SuperH™ (SH) 64-Bit RISC Series

SH-5 CPU Core, Volume 3: SHcompact

Last updated 22 February 2002

This publication contains proprietary information of SuperH, Inc., and is not to be copied in whole or part.

Issued by the SuperH Documentation Group on behalf of SuperH, Inc.

Information furnished is believed to be accurate and reliable. However, SuperH, Inc. 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 SuperH, Inc. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. SuperH, Inc. products are not authorized for use as critical components in life support devices or systems without the express written approval of SuperH, Inc.

is a registered trademark of SuperH, Inc.

SuperH is a registered trademark for products originally developed by Hitachi, Ltd. and is owned by Hitachi Ltd.
© 2001 SuperH, Inc. All Rights Reserved.

SuperH, Inc.
San Jose, U.S.A. - Bristol, United Kingdom - Tokyo, Japan

www.superh.com

Contents

Preface **xiii**

SuperH SH-5 document identification and control xiii

SuperH SH-5 CPU core documentation suite xiv

1 SHcompact specification 1

1.1 Overview 1

1.2 SHcompact architectural state 1

1.3 General-purpose registers 3

1.3.1 R0 To R15, GBR and PR 3

1.3.2 T-bit 7

1.3.3 MACL and MACH 8

1.3.4 Discussion 8

1.4 Floating-point registers 9

1.5 FPSCR, PR, SZ and FR 11

1.6 Delayed branches and delay slots 12

1.7 Scratch registers 14

1.8 Memory, cache and floating-point models 14

1.9 Abstract sequential model 14

1.9.1 Initial conditions 15

1.9.2 Instruction execution loop 15

1.9.3 Non-delayed and delayed state changes 16

2 SHcompact instruction set 19

2.1 Alphabetical list of instructions 19
ADD Rm, Rn 20
ADD #imm, Rn 21
ADDC Rm, Rn 22
ADDV Rm, Rn 23
AND Rm, Rn 24
AND #imm, R0 25
AND.B #imm, @(R0, GBR) 26
BF label 27
BF/S label 29
BRA label 31
BRAF Rn 32
BRK 33
BSR label 34
BSRF Rn 36
BT label 38
BT/S label 40
CLRMAC 42
CLRS 43
CLRT 44
CMP/EQ Rm, Rn 45
CMP/EQ #imm, R0 46
CMP/GE Rm, Rn 47
CMP/GT Rm, Rn 48
CMP/HI Rm, Rn 49
CMP/HS Rm, Rn 50
CMP/PL Rn 51

CMP/PZ Rn 52
CMP/STR Rm, Rn 53
DIV0S Rm, Rn 54
DIV0U 55
DIV1 Rm, Rn 56
DMULS.L Rm, Rn 57
DMULU.L Rm, Rn 58
DT Rn 59
EXTS.B Rm, Rn 60
EXTS.W Rm, Rn 61
EXTU.B Rm, Rn 62
EXTU.W Rm, Rn 63
FABS DRn 64
FABS FRn 65
FADD DRm, DRn 66
FADD FRm, FRn 67
FCMP/EQ DRm, DRn 69
FCMP/EQ FRm, FRn 70
FCMP/GT DRm, DRn 72
FCMP/GT FRm, FRn 73
FCNVDS DRm, FPUL 75
FCNVSD FPUL, DRn 76
FDIV DRm, DRn 78
FDIV FRm, FRn 79
FIPR FVm, FVn 82
FLDI0 FRn 85
FLDI1 FRn 86
FLDS FRm, FPUL 87

FLOAT FPUL, DRn 88
FLOAT FPUL, FRn 89
FMAC FR0, FRm, FRn 91
FMOV DRm, DRn 95
FMOV DRm, XDn 96
FMOV DRm, @Rn 97
FMOV DRm, @-Rn 98
FMOV DRm, @(R0, Rn) 99
FMOV FRm, FRn 100
FMOV.S FRm, @Rn 101
FMOV.S FRm, @-Rn 102
FMOV.S FRm, @(R0, Rn) 103
FMOV XDm, DRn 104
FMOV XDm, XDn 105
FMOV XDm, @Rn 106
FMOV XDm, @-Rn 107
FMOV XDm, @(R0, Rn) 108
FMOV @Rm, DRn 109
FMOV @Rm+, DRn 110
FMOV @(R0, Rm), DRn 111
FMOV.S @Rm, FRn 112
FMOV.S @Rm+, FRn 113
FMOV.S @(R0, Rm), FRn 114
FMOV @Rm, XDn 115
FMOV @Rm+, XDn 116
FMOV @(R0, Rm), XDn 117
FMUL DRm, DRn 118
FMUL FRm, FRn 119

FNEG DRn 121
FNEG FRn 122
FRCHG 123
FSCA FPUL, DRn 124
FSCHG 126
FSQRT DRn 127
FSQRT FRn 128
FSRRA FRn 130
FSTS FPUL, FRn 132
FSUB DRm, DRn 133
FSUB FRm, FRn 134
FTRC DRm, FPUL 136
FTRC FRm, FPUL 137
FTRV XMTRX, FVn 139
JMP @Rn 143
JSR @Rn 144
LDC Rm, GBR 146
LDC.L @Rm+, GBR 147
LDS Rm, FPSCR 148
LDS.L @Rm+, FPSCR 149
LDS Rm, FPUL 150
LDS.L @Rm+, FPUL 151
LDS Rm, MACH 152
LDS.L @Rm+, MACH 153
LDS Rm, MACL 154
LDS.L @Rm+, MACL 155
LDS Rm, PR 156
LDS.L @Rm+, PR 157

MAC.L @Rm+, @Rn+ 158
MAC.W @Rm+, @Rn+ 160
MOV Rm, Rn 162
MOV #imm, Rn 163
MOV.B Rm, @Rn 164
MOV.B Rm, @-Rn 165
MOV.B Rm, @(R0, Rn) 166
MOV.B R0, @(disp, GBR) 167
MOV.B R0, @(disp, Rn) 168
MOV.B @Rm, Rn 169
MOV.B @Rm+, Rn 170
MOV.B @(R0, Rm), Rn 171
MOV.B @(disp, GBR), R0 172
MOV.B @(disp, Rm), R0 173
MOV.L Rm, @Rn 174
MOV.L Rm, @-Rn 175
MOV.L Rm, @(R0, Rn) 176
MOV.L R0, @(disp, GBR) 177
MOV.L Rm, @(disp, Rn) 178
MOV.L @Rm, Rn 179
MOV.L @Rm+, Rn 180
MOV.L @(R0, Rm), Rn 181
MOV.L @(disp, GBR), R0 182
MOV.L @(disp, PC), Rn 183
MOV.L @(disp, Rm), Rn 184
MOV.W Rm, @Rn 185
MOV.W Rm, @-Rn 186
MOV.W Rm, @(R0, Rn) 187

MOV.W R0, @(disp, GBR) 188
MOV.W R0, @(disp, Rn) 189
MOV.W @Rm, Rn 190
MOV.W @Rm+, Rn 191
MOV.W @(R0, Rm), Rn 192
MOV.W @(disp, GBR), R0 193
MOV.W @(disp, PC), Rn 194
MOV.W @(disp, Rm), R0 195
MOVA @(disp, PC), R0 196
MOVCA.L R0, @Rn 197
MOVT Rn 199
MUL.L Rm, Rn 200
MULS.W Rm, Rn 201
MULU.W Rm, Rn 202
NEG Rm, Rn 203
NEGC Rm, Rn 204
NOP 205
NOT Rm, Rn 206
OCBI @Rn 207
OCBP @Rn 208
OCBWB @Rn 209
OR Rm, Rn 210
OR #imm, R0 211
OR.B #imm, @(R0, GBR) 212
PREF @Rn 213
ROTCL Rn 214
ROTCR Rn 215
ROTL Rn 216

ROTR Rn 217
RTS 218
SETS 219
SETT 220
SHAD Rm, Rn 221
SHAL Rn 222
SHAR Rn 223
SHLD Rm, Rn 224
SHLL Rn 225
SHLL2 Rn 226
SHLL8 Rn 227
SHLL16 Rn 228
SHLR Rn 229
SHLR2 Rn 230
SHLR8 Rn 231
SHLR16 Rn 232
STC GBR, Rn 233
STC.L GBR, @-Rn 234
STS FPSCR, Rn 235
STS.L FPSCR, @-Rn 236
STS FPUL, Rn 237
STS.L FPUL, @-Rn 238
STS MACH, Rn 239
STS.L MACH, @-Rn 240
STS MACL, Rn 241
STS.L MACL, @-Rn 242
STS PR, Rn 243
STS.L PR, @-Rn 244

SUB Rm, Rn 245
SUBC Rm, Rn 246
SUBV Rm, Rn 247
SWAP.B Rm, Rn 248
SWAP.W Rm, Rn 249
TAS.B @Rn 250
TRAPA #imm 252
TST Rm, Rn 253
TST #imm, R0 254
TST.B #imm, @(R0, GBR) 255
XOR Rm, Rn 256
XOR #imm, R0 257
XOR.B #imm, @(R0, GBR) 258
XTRCT Rm, Rn 259

A SHcompact instruction encoding 261

A.1 Formats 261
A.2 0 format 261
A.3 n format 262
A.4 m format 263
A.5 nm format 263
A.6 md format 264
A.7 nd4 format 265
A.8 nmd format 265
A.9 d format 266
A.10 d12 format 266
A.11 nd8 format 267
A.12 i format 267

A.13 ni format 268
A.14 Opcode assignment 268
A.15 Reserved instructions 269
A.16 Floating-point instructions 272

Index 273

Preface

This document is part of the SuperH SH-5 CPU core documentation suite detailed below. Comments on this or other books in the documentation suite should be made by contacting your local sales office or distributor.

SuperH SH-5 document identification and control

Each book in the documentation suite carries a unique identifier in the form:

05-CC-nnnnn Vx.x

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

Whenever making comments on a SuperH SH-5 document the complete identification 05-CC-1000n Vx.x should be quoted.

SuperH SH-5 CPU core documentation suite

The SuperH SH-5 CPU core documentation suite comprises the following volumes:

- SH-5 CPU Core, Volume 1: Architecture (05-CC-10001)
- SH-5 CPU Core, Volume 2: SHmedia (05-CC-10002)
- SH-5 CPU Core, Volume 3: SHcompact (05-CC-10003)
- SH-5 CPU Core, Volume 4: Implementation (05-CC-10004)

SHcompact specification

1

1.1 Overview

The SHcompact specification uses the language described in *Volume 2, Chapter 1: SHmedia specification*. This chapter describes additional details that are specific to the SHcompact specification.

1.2 SHcompact architectural state

SHcompact state is mapped on the same architectural state used by SHmedia. The architectural state and this mapping are described in *Volume 1, Chapter 2: Architectural state*.

SHcompact instructions are specified in terms of the full architectural state. This has the following implications:

- General-purpose registers are 64 bits wide in the architectural state. The specification language reads and writes all 64 bits as required to implement the 32-bit view of these registers seen by SHcompact instructions. Further details are given in *Section 1.3: General-purpose registers on page 3*.
- Floating-point registers are not banked in the architectural state. The specification language has to map the banked view of floating-point registers seen by SHcompact instructions onto the flat floating-point register set. Further details are given in *Section 1.4: Floating-point registers on page 9*.
- FPSCR is formatted as defined by the architectural state. The SHcompact view of FPSCR also includes 3 bits (PR, SZ and FR) which are copied from SR. The specification language has to map between these 2 views. Further details are given in *Section 1.5: FPSCR, PR, SZ and FR on page 11*.

- The SHcompact instruction set supported delayed branches and delay slots. Additional state notation is required to support this mechanism, and this is described in *Section 1.6: Delayed branches and delay slots on page 12*.

The view of architectural state used by the specification language is described in *Volume 2, Chapter 1: SHmedia specification*. SHcompact specification uses additional names for some state as described in *Table 1*.

Name	Architectural State	Description
GBR	R_{16}	Global base register
MACL	Lower 32 bits of R_{17}	Multiply-accumulate low
MACH	Upper 32 bits of R_{17}	Multiply-accumulate high
PR	R_{18}	Procedure link register
T	Bit 0 of R_{19}	Condition code flag
S	SR.S	Multiply-accumulate saturation flag
M	SR.M	Divide-step M flag
Q	SR.Q	Divide-step Q flag
FPSCR.PR	SR.PR	Floating-point precision of operation
FPSCR.SZ	SR.SZ	Floating-point size of data transfer
FPSCR.FR	SR.FR	Floating-point bank selection
FPUL	FR_{32}	FPU communication register

Table 1: Mapping from additional names to architectural state

1.3 General-purpose registers

General-purpose registers are 64 bits wide. These registers are visible as R_0 to R_{15}, GBR, MACL, MACH, PR and the T-bit to SHcompact instructions. Apart from MACL and MACH, SHcompact is only able to observe a subset of the bits contained within these general-purpose registers. For correct operation of SHcompact instructions, the non-observable bits must be maintained appropriately.

Table 2 shows the non-observable bits and the usual treatment applied while in SHcompact mode.

Architectural state	SHcompact name	Observable bits	Non-observable bits	Treatment for non-observable bits
R_0 to R_{15}	R_0 to R_{15}	[0, 31]	[32, 63]	Usual treatment: sign extension of bit 31
R_{16}	GBR	[0, 31]	[32, 63]	Usual treatment: sign extension of bit 31
R_{17}	MACL, MACH	[0, 63]	None	No special cases: all bits are observable
R_{18}	PR	[0. 31]	[32, 63]	Usual treatment: sign extension of bit 31
R_{19}	T	0	[1, 63]	Usual treatment: all set to zero (see *Section 1.3.2: T-bit on page 7*)

Table 2: General-purpose registers visible to SHcompact instructions

1.3.1 R_0 To R_{15}, GBR and PR

The architecture defines policies for the interpretation of non-observable bits in these registers when used as source operands to SHcompact instructions:

- 64-bit sources: the SHcompact instruction interprets the source as a 64-bit value. The instruction applies its operation to all 64 bits of the source operand.
- 32-bit sources: the SHcompact instruction discards the upper 32 bits of the source to leave a 32-bit value. The instruction applies its operation to these lower 32 bits of the operand.
- 32-bit sign-extended sources: the SHcompact instruction requires that the source operand has a value in the signed 32-bit integer range. If this condition is met, then the instruction applies its operation to the lower 32 bits of the operand. If this condition is not met, then the value of the source operand seen by that instruction is architecturally undefined.

These policies are applied to SHcompact instructions as follows:

1 General-purpose register to general-purpose register move operates at 64-bit width. There are no architecturally undefined cases.

2 Bit-wise AND, NOT, OR and XOR instructions operate at 64-bit width. There are no architecturally undefined cases.

3 Sign extension, zero extension, rotates, shifts, swaps and extract operate at 32-bit width. There are no architecturally undefined cases.

4 All instructions, apart from general-purpose register to general-purpose register move, that transfer a value out of a general-purpose register, GBR or PR read just the required bits of that register. This includes instructions such as general-purpose register stores, stores from system and control registers, and loads to system and control registers. There are no architecturally undefined cases.

5 All instructions that read general-purpose registers, other than the cases listed above, require that all non-observable bits are sign extensions of bit 31.

The policy in *5* is the usual treatment as described in *Table 2*. The other policies allow instructions that perform non-arithmetic data manipulation or data transfer to be used safely on 64-bit values. The specific cases where the usual treatment is relaxed are defined in the following tables.

Instruction	Operand interpretation	Instruction semantics
MOV Rm, Rn	R_m is a 64-bit source	64-bit move
AND Rm, Rn	R_m and R_n are 64-bit sources	64-bit bitwise AND
AND #imm, R0	R_0 is a 64-bit source	64-bit bitwise AND
NOT Rm, Rn	R_m is a 64-bit source	64-bit bitwise NOT
OR Rm, Rn	R_m and R_n are 64-bit sources	64-bit bitwise OR
OR #imm, R0	R_0 is a 64-bit source	64-bit bitwise OR
XOR Rm, Rn	R_m and R_n are 64-bit sources	64-bit bitwise XOR
XOR #imm, R0	R_0 is a 64-bit source	64-bit bitwise XOR

Table 3: SHcompact instructions with 64-bit source operands

Instruction	Operand interpretation	Instruction semantics
EXTS.B Rm, Rn EXTS.W Rm, Rn EXTU.B Rm, Rn EXTU.W Rm, Rn	R_m and R_n are 32-bit sources	Sign or zero extend to produce 32-bit result
ROTCL Rn ROTCR Rn	R_n is a 32-bit source	Rotate at 32-bit width with carry
ROTL Rn ROTR Rn	R_n is a 32-bit source	Rotate at 32 bit width
SHAD Rm, Rn SHLD Rm, Rn	R_m and R_n are 32-bit sources	Dynamic shift at 32-bit width
SHAL Rn SHAR Rn SHLL Rn SHLL16 Rn SHLL2 Rn SHLL8 Rn SHLR Rn SHLR16 Rn SHLR2 Rn SHLR8 Rn	R_n is a 32-bit source	Shift at 32-bit width
SWAP.B Rm, Rn SWAP.W Rm, Rn	R_m and R_n are 32-bit sources	Swap to produce 32-bit result
XTRCT Rm, Rn	R_m and R_n are 32-bit sources	Extract to produce 32-bit result

Table 4: SHcompact instructions with 32-bit source operands

Instruction	Operand interpretation	Instruction semantics
MOV.B R0, @(disp,GBR) MOV.L R0, @(disp,GBR) MOV.W R0, @(disp,GBR)	GBR is a 32-bit sign-extended source R_0 is a 32-bit source	8/16/32-bit data transfer
MOV.B R0, @(disp,Rn) MOV.W R0, @(disp,Rn) MOVCA.L R0, @Rn	R_n is a 32-bit sign-extended source If R_0 is a different register to R_n: R_0 is a 32-bit source Else: R_0 is a 32-bit sign-extended source	8/16/32-bit data transfer
MOV.B Rm, @-Rn MOV.B Rm, @Rn MOV.L Rm, @-Rn MOV.L Rm, @Rn MOV.L Rm, @(disp,Rn) MOV.W Rm, @-Rn MOV.W Rm, @Rn	R_n is a 32-bit sign-extended source If R_m is a different register to R_n: R_m is a 32-bit source Else: R_m is a 32-bit sign-extended source	8/16/32-bit data transfer
MOV.B Rm, @(R0,Rn) MOV.L Rm, @(R0,Rn) MOV.W Rm, @(R0,Rn)	R_0 is a 32-bit sign-extended source R_n is a 32-bit sign-extended source If R_m is a different register to R_0 and R_n: R_m is a 32-bit source Else: R_m is a 32-bit sign-extended source	8/16/32-bit data transfer
STC GBR, Rn	GBR is a 32-bit source	32-bit data transfer
STC.L GBR, @-Rn	R_n is a 32-bit sign-extended source GBR is a 32-bit source	32-bit data transfer
STS PR, Rn	PR is a 32-bit source	32-bit data transfer
STS.L PR, @-Rn	R_n is a 32-bit sign-extended source PR is a 32-bit source	32-bit data transfer

Table 5: SHcompact instructions that transfer data from a 32-bit source operand

Instruction	Operand interpretation	Instruction semantics

LDC Rm, GBR
LDS Rm, FPSCR
LDS Rm, FPUL
LDS Rm, MACH
LDS Rm, MACL
LDS Rm, PR
	R_m is a 32-bit source	32-bit data transfer

Table 5: SHcompact instructions that transfer data from a 32-bit source operand

Software must ensure that non-observable bits in general-purpose register source operands are correct when an SHcompact instruction is executed:

- If this condition is met, then the SHcompact instruction has the behavior defined by the architecture.
- If this condition is not met, then the values of each incorrectly-formed source operand seen by that instruction is architecturally undefined. The instruction will complete execution, though the results can be unpredictable due to the undefinedness of its data. This can result in an unexpected exception (for example, due to an incorrect effective address calculation) or propagation of architecturally undefined values into destination registers.

Each SHcompact instruction that operates at 64-bit width (see *Table 3*) has an important property. If all source operands of that instruction are 32-bit sign-extended sources, then the result will also be in a 32-bit sign-extended representation. This is not a requirement for the execution of these instructions, but it does mean that these instructions have the obvious behavior where software is using the usual treatment for non-observable bits.

1.3.2 T-bit

The T-bit follows similar policies to those described in *Section 1.3.1: R0 To R15, GBR and PR on page 3* except that only the lowest bit of the T-bit is observable. The upper 63 bits of the T-bit are non-observable, and the allowed values for the T-bit are only 0 and 1.

SHcompact instructions that read the T-bit require that all non-observable bits of the T-bit are 0. If this condition is met, then the instruction observes the T-bit equal to 0 or 1 as expected. If this condition is not met, then the value of the T-bit seen by that instruction is architecturally undefined.

Providing that all necessary conditions are met on source operands, SHcompact instructions that write to the T-bit will write the T-bit as 0 or 1. However, if

conditions on the source operands are not met, then the behavior is already architecturally undefined and the value written to the T-bit could be neither 0 nor 1.

Observation of inappropriate values for the T-bit can be avoided by software convention. Two example strategies are:

- Software could ensure that R_{19} has a value of 0 or 1 on all mode switches from SHmedia to SHcompact.
- Software could ensure, following a mode switch from SHmedia to SHcompact, that the T-bit is written before there are any reads of the T-bit.

Consider the following instruction sequence:

1 An SHmedia instruction sets R_{19} to a value other than 0 or 1.

2 Mode switch from SHmedia to SHcompact.

3 The SHcompact sequence neither reads from nor writes to the T-bit.

4 Mode switch from SHcompact to SHmedia.

5 An SHmedia instruction observes the value of R_{19}.

A consequence of the architecture is that the R_{19} value read in step 5 is guaranteed to be the value written to R_{19} in step 1. Additionally, the behavior of this sequence is architecturally defined since the T-bit is never read while it contains an inappropriate value.

1.3.3 MACL and MACH

MACL and MACH occupy all of R17. Since all bits of R17 are observable from SHcompact mode, special care is not required.

1.3.4 Discussion

In general, general-purpose registers that are visible to SHcompact instructions should have their non-observable bits set according to the usual treatment defined in *Table 2*. If this treatment is in effect at a particular instance, then any subsequent sequence of SHcompact instructions will continue to uphold this treatment.

The net effect of these rules is that SHcompact instructions execute correctly, providing that the necessary conditions are met at interfaces with SHmedia code. Software conventions are typically used to ensure the SHcompact visible registers have correctly formed values at mode switches from SHmedia to SHcompact.

The special cases defined in *Section 1.3.1* are specifically designed to allow software to relax the usual treatment in situations such as the following:

- A general-purpose register contains an uninitialized value.
- A general-purpose register contains a temporary that can be safely discarded.
- A general-purpose register deliberately contains a 64-bit value (for example, a parameter).

In these cases, the general-purpose register is not guaranteed to be in a signed 32-bit range. The relaxed treatment allows software to stay within the architecture.

Functions are used in the specification language to denote source operands where a value is expected to be in a certain signed or unsigned range.

Function	Description
$SignExpect_n$(value)	If value is in $[-2^{n-1}, 2^{n-1})$, returns value If value is not in this range, returns an architecturally undefined value
$ZeroExpect_n$(value)	If value is in $[0, 2^{n})$, returns value If value is not in this range, returns an architecturally undefined value

Table 6: Support functions for sign and zero expectancy

1.4 Floating-point registers

The specification language maps from the banked view of floating-point registers seen by SHcompact instruction to the flat architectural floating-point register set.

Two additional variable names are used to support this mapping.

Name	Value	Description
FRONT	If SR.FR is 0, FRONT is 0 If SR.FR is 1, FRONT is 16	First register index in the regular bank of floating-point registers
BACK	If SR.FR is 0, BACK is 16 If SR.FR is 1, BACK is 0	First register index in the extended bank of floating-point registers

Table 7: Variables to support bank selection

Additionally, SHcompact instructions use the DR notation to refer to pairs of single-precision floating-point registers. This is mapped onto the correct FP notation in the instruction specifications. The full set of mappings are given in *Table 8*.

Names of SHcompact state	Description of state	Architectural state name
FR_i where i is in [0, 15]	Single-precision registers	$FR_{FRONT+i}$
DR_{2i} where i is in [0, 7]	Double-precision registers	$DR_{FRONT+2i}$
	Single-precision register pairs	$FP_{FRONT+2i}$
FV_{4i} where i is in [0, 3]	Single-precision vector	$FV_{FRONT+4i}$
XF_i where i is in [0, 15]	Single-precision extended registers	FR_{BACK+i}
XD_{2i} where i is in [0, 7]	Double-precision extended registers	$DR_{BACK+2i}$
	Single-precision extended register pairs	$FP_{BACK+2i}$
XMTRX	Single-precision extended register matrix	$MTRX_{BACK}$

Table 8: Mapping of banked SHcompact floating-point state

1.5 FPSCR, PR, SZ and FR

The specification language has to map the SHcompact view of FPSCR onto the architectural state. When an SHcompact instruction reads from FPSCR, the specification has to pack FPSCR, SR.PR, SR.SZ and SR.FR into a single 32-bit value. When an SHcompact instruction writes to FPSCR, the specification has to unpack the 32-bit value into FPSCR, SR.PR, SR.SZ and SR.FR.

Two functions are used in the specification language to denote this packing and unpacking.

Function	Description
value ← PackFPSCR(fpscr, pr, sz, fr)	This function packs the given parameters into a single FPSCR value as seen in SHcompact.
fpscr, pr, sz, fr ← UnpackFPSCR(value)	This function unpacks the single FPSCR value (as seen in SHcompact) into the given results list.

Table 9: Support functions for FPSCR packing and unpacking

These 3 bits have the following effects on the SHcompact instruction specification:

- SR.PR selects the precision of operation: 0 indicates single-precision and 1 indicates double-precision. Some floating-point instructions are only available when SR.PR has a certain value. These requirements are shown in the instruction specification.
- SR.SZ selects the width of data-transfer for floating-point loads and stores: 0 indicates transfers of 32-bit registers and 1 indicates transfers of pairs of 32-bit registers (64 bits). Some floating-point instructions are only available when SR.SZ has a certain value. These requirements are shown in the instruction specification.
- SR.FR determines which bank is viewed using the regular floating-point register names and which as the extended bank: the banking arrangement is described in *Section 1.4: Floating-point registers on page 9*.

1.6 Delayed branches and delay slots

SHcompact supports delayed branches. The instruction immediately following a delayed branch in memory is called a delay slot instruction. For a delayed branch, the delay slot is executed before the branch is effected. There are special rules and notations for the modelling of this mechanism.

The delayed branch instructions are listed in the following table.

Instruction	Summary
BF/S label	delayed branch if false
BRA label	delayed branch
BRAF Rn	delayed branch far
BSR label	delayed branch to subroutine
BSRF Rn	delayed branch to subroutine far
BT/S label	delayed branch if true
JMP @Rn	delayed jump
JSR @Rn	delayed jump to subroutine
RTS	delayed return from subroutine

Table 10: Delayed branch instructions

Any instruction can be placed in a delay slot apart from those listed in *Table 11*. If any of these instructions are executed in a delay slot, an ILLSLOT exception is raised.

Instruction	Summary
BF/S, BRA, BRAF, BSR, BSRF, BT/S, JMP, JSR, RTS	Any delayed branch instruction (see *Table 10*)
BF label	branch if false
BT label	branch if true
MOV.L @(disp, PC), Rn	load 32-bits from PC with displacement

Table 11: Illegal delay slot instructions

Instruction	Summary
MOV.W @(disp, PC), Rn	load 16-bits from PC with displacement
MOVA @(disp, PC), R0	move PC-relative address
TRAPA #imm	trap always

Table 11: Illegal delay slot instructions

Any floating-point instruction can be placed in a delay slot. When the FPU is disabled, the execution of a floating-point instruction normally leads to an FPUDIS exception. However, when the FPU is disabled and a floating-point instruction in a delay slot is executed, a SLOTFPUDIS exception is raised instead. This approach simplifies software emulation of floating-point instructions.

The following additional notation is used:

- PC' refers to the PC value after this instruction has executed.
- PR' refers to the PR value after this instruction has executed.
- ISA' refers to the ISA value after this instruction has executed.
- PC" refers to the PC value after this and the next instruction have executed.
- PR" refers to the PR value after this and the next instruction have executed.
- ISA" refers to the ISA value after this and the next instruction have executed.

The execution model described in *Section 1.9: Abstract sequential model on page 14* uses this state to model delayed branches.

A function is used to indicate whether an instruction is executing in a delay slot.

Function	Description
IsDelaySlot()	If instruction is executing in a delay slot, returns true If instruction is not executing in a delay slot, returns false

Table 12: Support function to distinguish delay slots

1.7 Scratch registers

Volume 1, Chapter 2: Architectural state defines a set of scratch registers that are used as scratch state during the execution of SHcompact instructions. Scratch registers are not explicitly modeled in the specification language.

The scratch registers are summarized in *Table 13*.

Scratch register	Becomes architecturally undefined:
R_{20} to R_{23} inclusive	when any SHcompact instruction is executed (even if the instruction causes an exception).
TR_0 to TR_3 inclusive	when any SHcompact instruction is executed (even if the instruction causes an exception).
FR_{33}	when any SHcompact floating-point instruction is executed (even if the instruction causes an exception).

Table 13: Scratch registers

1.8 Memory, cache and floating-point models

SHcompact specification uses the same models of memory, cache and floating-point operation as SHmedia, *Volume 2, Chapter 1: SHmedia specification*. A subset of these support functions are used in the SHcompact specifications.

1.9 Abstract sequential model

The abstract sequential model of SHcompact instruction execution is largely similar to its SHmedia counterpart. The model is modified to accommodate the 2-byte instructions in SHcompact and the delayed branching mechanism.

Section 1.9.1 describes the initial conditions that are initialized upon a mode switch from SHmedia to SHcompact. No special actions are required upon a mode switch from SHcompact to SHmedia. *Section 1.9.2* describes the steps taken to execute each SHcompact instruction in the abstract sequential model. *Section 1.9.3* describes the mechanisms used to model delayed branching.

1.9.1 Initial conditions

The abstract model described here maintains hidden internal state in the variables PC”, PR” and ISA” to keep track of delayed state changes. These values are automatically set to appropriate initial conditions at the beginning of a sequence of SHcompact instructions. The beginning of an SHcompact instruction sequence occurs when the previous instruction is an SHmedia instruction that mode switches to SHcompact. The initial state is set as follows:

- PC” is set to PC+2
- PR” is set to the same value as PR
- ISA” is set to 0

1.9.2 Instruction execution loop

If ISA is 1, the instruction is executed in SHmedia mode as described in *Volume 2, Chapter 1: SHmedia specification*. Otherwise, the instruction is executed in SHcompact mode. The steps associated with executing each SHcompact 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. For SHcompact, 2 bytes need to be fetched for each instruction.
4. Calculate the default values of PC’, PR’ and ISA’. PC’ is set to the value of PC”, PR’ is set to the value of PR” and ISA’ is set to the value of ISA”.
5. Calculate the default values of PC”, PR” and ISA” assuming continued sequential execution without procedure call or mode switch. For SHcompact, PC” is PC’+2, while PR” and ISA” are 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. Special case handling of SHcompact events is described in *Volume 1, Chapter 16: Event handling*.

The execution of an instruction can update the PC, PR and ISA 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", PR" and ISA" to achieve a branch, procedure call or mode-switch after the next instruction has completed.

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

7 If the value of PC' is outside of the implemented part of the effective address space, then the behavior becomes architecturally undefined.

8 Set the current program counter (PC) to the value of the next program counter (PC'). Similarly, set PR to the value of PR' and set ISA to the value of ISA'.

The actions associated with the handling of asynchronous and synchronous events are described in *Volume 1, Chapter 16: Event handling*. 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. This is described further in *Volume 1, Chapter 3: Data representation*.

1.9.3 Non-delayed and delayed 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, PR and ISA 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", PR" or ISA" to cause a delayed state change:

- A non-delayed state change is achieved by updating PC' or PR' to over-ride their default values. There is no mechanism to update ISA' as the result of instruction execution. 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", PR" or ISA" to override their default values. After the execution of this instruction, PC", PR" or ISA" get copied to PC', PR' and ISA' respectively. After the execution of the next instruction, PC', PR' and ISA' get copied to PC, PR and ISA respectively, and then influence instruction execution. Hence, there is a delay slot before the values of PC", PR" and ISA" propagate through to PC, PR and ISA.

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:

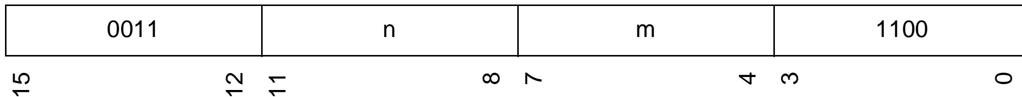
- The case of a delayed state change to PC immediately followed by a read of PC does not occur. This is because delay slot instructions that read from PC are illegal and cause an ILLSLOT exception.
- The case of a delayed state change to PR immediately followed by a read from PR can occur. The ambiguous 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).

2 SHcompact instruction set

2.1 Alphabetical list of instructions

ADD Rm, Rn

ADD Rm, Rn



0011	n	m	1100
15 12	11 8	7 4	3 0

op1 ← $SignExpect_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
op2 ← op2 + op1;
R_n ← $Register(SignExtend_{32}(op2))$;

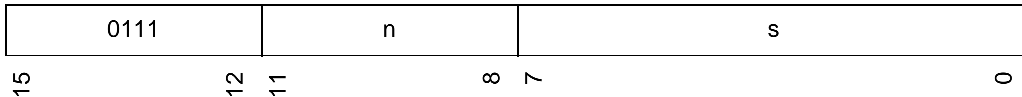
Description:

This instruction adds R_m to R_n and places the result in R_n.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

ADD #imm, Rn



imm ← $SignExtend_8$(s);
op2 ← $SignExpect_{32}$(R_n);
op2 ← op2 + imm;
R_n ← Register($SignExtend_{32}$(op2));

Description:

This instruction adds R_n to the sign-extended 8-bit immediate s and places the result in R_n.

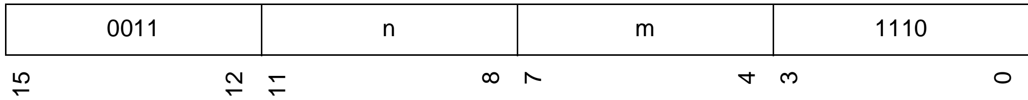
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The '#imm' in the assembly syntax represents the immediate s after sign extension.

ADDC Rm, Rn

ADDC Rm, Rn



$t \leftarrow ZeroExpect_1(T);$
$op1 \leftarrow ZeroExtend_{32}(SignExpect_{32}(R_m));$
$op2 \leftarrow ZeroExtend_{32}(SignExpect_{32}(R_n));$
$op2 \leftarrow (op2 + op1) + t;$
$t \leftarrow op2_{< 32 \text{ FOR } 1 >};$
$R_n \leftarrow Register(SignExtend_{32}(op2));$
$T \leftarrow Bit(t);$

Description:

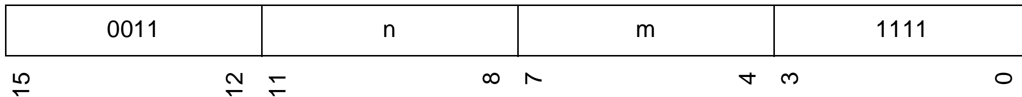
This instruction adds R_m, R_n and the T-bit. The result of the addition is placed in R_n. and the carry-out from the addition is placed in the T-bit.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation. The T-bit source is required to have a 0 or 1 value.

ADDV Rm, Rn

ADDV Rm, Rn



op1 ← $SignExpect_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
op2 ← op2 + op1;
t ← INT ((op2 < (- 2^{31})) OR (op2 ≥ 2^{31}));
R_n ← Register($SignExtend_{32}$(op2));
T ← Bit(t);

Description:

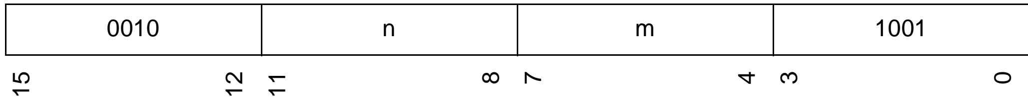
This instruction adds R_m to R_n and places the result in R_n. The T-bit is set to 1 if the addition result is outside the 32-bit signed range, otherwise the T-bit is set to 0.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

AND Rm, Rn

AND Rm, Rn



$op1 \leftarrow ZeroExtend_{64}(R_m);$
$op2 \leftarrow ZeroExtend_{64}(R_n);$
$op2 \leftarrow op2 \wedge op1;$
$R_n \leftarrow Register(op2);$

Description:

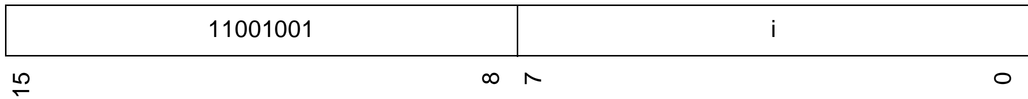
This instruction performs bitwise AND of R_m with R_n and places the result in R_n.

Notes:

This instruction performs a 64-bit bitwise AND. The R_m and R_n sources are not required to have their upper 32 bits as sign-extensions. However, if both source values have a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

AND #imm, R0

AND #imm, R0



$r0 \leftarrow ZeroExtend_{64}(R_0);$
$imm \leftarrow ZeroExtend_8(i);$
$r0 \leftarrow r0 \wedge imm;$
$R_0 \leftarrow Register(r0);$

Description:

This instruction performs bitwise AND of R_0 with the zero-extended 8-bit immediate i and places the result in R_0.

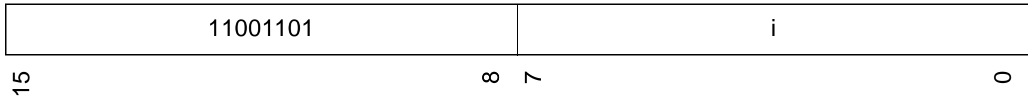
Notes:

This instruction performs a 64-bit bitwise AND. The R_0 source is not required to have its upper 32 bits as sign-extensions. However, if the R_0 source value has a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

The ‘#imm’ in the assembly syntax represents the immediate i after zero extension.

AND.B #imm, @(R0, GBR)

AND.B #imm, @(R0, GBR)




```
r0 ← SignExpect32(R0);
gbr ← SignExpect32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend64(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ∧ imm;
WriteMemory8(address, value);
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT, WRITEPROT

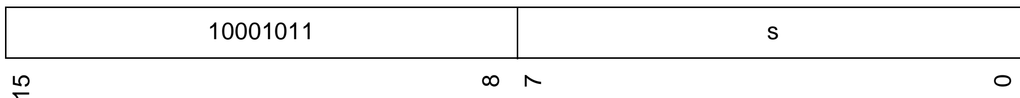
Notes:

The R_0 and GBR sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

BF label




```
t ← ZeroExpect1(T);
pc ← SignExpect32(PC);
newpc ← SignExpect32(PC');
delayedpc ← SignExpect32(PC");
offset ← SignExtend8(s) << 1;
label ← (pc + 4) + offset;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (MalformedAddress(label))
    THROW IADDERR, label;
IF (t = 0)
{
    newpc ← label;
    delayedpc ← label + 2;
}
PC' ← Register(SignExtend32(newpc));
PC" ← Register(SignExtend32(delayedpc));
```

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.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

The T-bit source is required to have a 0 or 1 value.

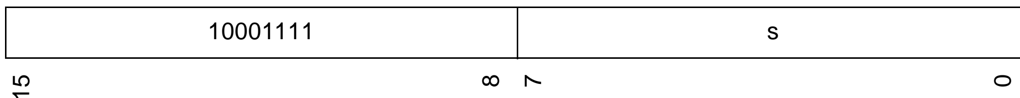
The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space. The exception check on the

target address is performed regardless of whether the conditional branch is taken or not-taken.

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 absolute address of the target SHcompact instruction.

BF/S label




```
t ← ZeroExpect1(T);
pc ← SignExpect32(PC);
delayedpc ← SignExpect32(PC”);
offset ← SignExtend8(s) << 1;
label ← (pc + 4) + offset;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (MalformedAddress(label))
    THROW IADDERR, label;
IF (t = 0)
    delayedpc ← label;
PC” ← Register(SignExtend32(delayedpc));
```

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.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

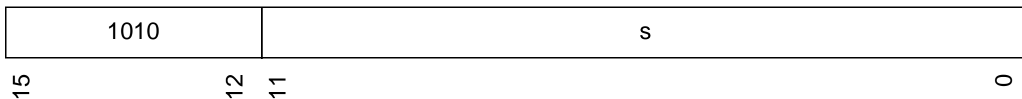
The T-bit source is required to have a 0 or 1 value.

The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space. The exception check on the target address is performed regardless of whether the conditional branch is taken or not-taken.

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 absolute address of the target SHcompact instruction.

BRA label




```
pc ← SignExpect32(PC);
offset ← SignExtend12(s) << 1;
label ← (pc + 4) + offset;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (MalformedAddress(label))
    THROW IADDERR, label;
delayedpc ← label;
PC” ← Register(SignExtend32(delayedpc));
```

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.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

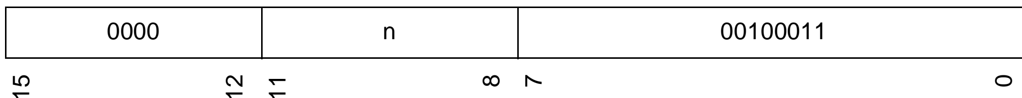
The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

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 absolute address of the target SHcompact instruction.

BRAF Rn

BRAF Rn




```
pc ← SignExpect32(PC);
op1 ← SignExpect32(Rn);
IF (IsDelaySlot())
   THROW ILLSLOT;
target ← (pc + 4) + op1;
IF (MalformedAddress(target) OR ((target ∧ 0x3) = 0x3))
   THROW IADDERR, target;
delayedisa ← target ∧ 0x1;
delayedpc ← target ∧ (~ 0x1);
PC” ← Register(SignExtend32(delayedpc));
ISA” ← Bit(delayedisa);
```

Description:

This instruction is a delayed unconditional branch. The target address is calculated by adding R_n to PC+4. If the last two bits of the target address are both set, an IADDERR exception is raised. Otherwise, the delay slot is executed in SHcompact. Bit zero of the target address gives the new value of the ISA mode for the next instruction. The least significant bit of the target address is cleared, and this value is copied to the PC.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

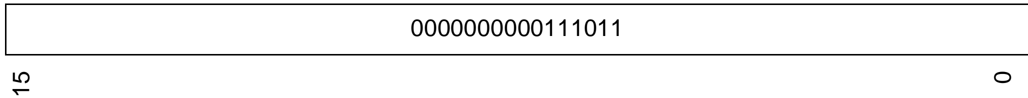
The R_n source is required to have a 32-bit sign-extended representation.

The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The delay slot is executed before branching and before ISA is updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

BRK

BRK




```
THROW BREAK;
```

Description:

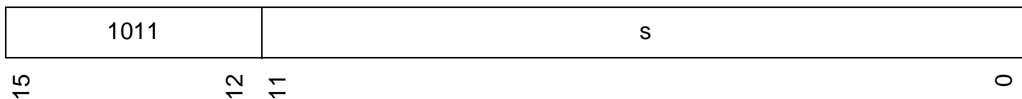
The BRK instruction causes a pre-execution BREAK exception. This exception is generated even if BRK is executed in a delay slot. The BRK instruction is typically reserved for use by the debugger.

Possible exceptions:

BREAK

BSR label

BSR label




```
pc ← SignExpect32(PC);
offset ← SignExtend12(s) << 1;
delayedpr ← pc + 4;
label ← (pc + 4) + offset;
IF (IsDelaySlot())
   THROW ILLSLOT;
IF (MalformedAddress(label))
   THROW IADDERR, label;
delayedpc ← label;
PR" ← Register(SignExtend32(delayedpr));
PC" ← Register(SignExtend32(delayedpc));
```

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.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

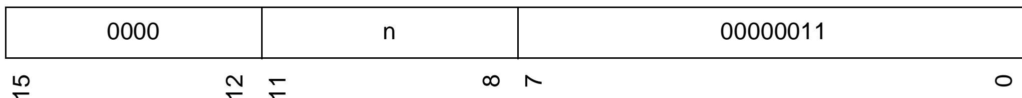
The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

If this instruction does not raise an exception then PR will be updated regardless of whether the delay slot instruction raises an exception. 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 absolute address of the target SHcompact instruction.

BSRF Rn

BSRF Rn




```
pc ← SignExpect32(PC);
op1 ← SignExpect32(Rn);
IF (IsDelaySlot())
   THROW ILLSLOT;
delayedpr ← pc + 4;
target ← (pc + 4) + op1;
IF (MalformedAddress(target) OR ((target ∧ 0x3) = 0x3))
   THROW IADDERR, target;
delayedisa ← target ∧ 0x1;
delayedpc ← target ∧ (~ 0x1);
PR” ← Register(SignExtend32(delayedpr));
PC” ← Register(SignExtend32(delayedpc));
ISA” ← Bit(delayedisa);
```

Description:

This instruction is a delayed unconditional branch used for branching to a far subroutine. The target address is calculated by adding R_n to PC+4. If the last two bits of the target address are both set, an IADDERR exception is raised. Otherwise, the delay slot is executed in SHcompact. Bit zero of the target address gives the new value of the ISA mode for the next instruction. The least significant bit of the target address is cleared, and this value is copied to the PC. The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

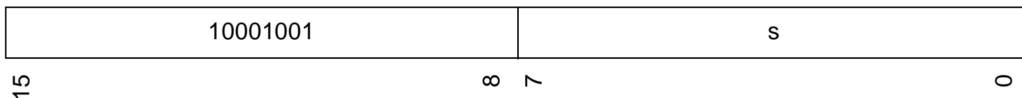
The R_n source is required to have a 32-bit sign-extended representation.

The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

If this instruction does not raise an exception then PR will be updated regardless of whether the delay slot instruction raises an exception. The delay slot is executed before branching and before ISA and PR are updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

BT label

BT label




```
t ← ZeroExpect1(T);
pc ← SignExpect32(PC);
newpc ← SignExpect32(PC’);
delayedpc ← SignExpect32(PC”);
offset ← SignExtend8(s) << 1;
label ← (pc + 4) + offset;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (MalformedAddress(label))
    THROW IADDERR, label;
IF (t = 1)
{
    newpc ← label;
    delayedpc ← label + 2;
}
PC’ ← Register(SignExtend32(newpc));
PC” ← Register(SignExtend32(delayedpc));
```

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.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

The T-bit source is required to have a 0 or 1 value.

The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space. The exception check on the

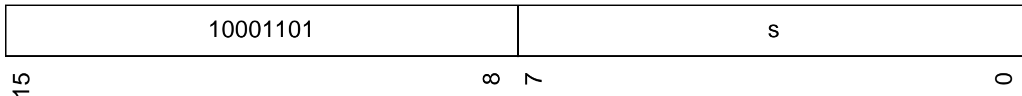
target address is performed regardless of whether the conditional branch is taken or not-taken.

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 absolute address of the target SHcompact instruction.

BT/S label

BT/S label




```
t ← ZeroExpect_1(T);
pc ← SignExpect_32(PC);
delayedpc ← SignExpect_32(PC”);
offset ← SignExtend_8(s) << 1;
label ← (pc + 4) + offset;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (MalformedAddress(label))
    THROW IADDERR, label;
IF (t = 1)
    delayedpc ← label;
PC” ← Register(SignExtend_32(delayedpc));
```

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.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

The T-bit source is required to have a 0 or 1 value.

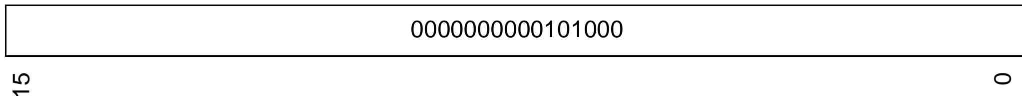
The target address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space. The exception check on the target address is performed regardless of whether the conditional branch is taken or not-taken.

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 absolute address of the target SHcompact instruction.

CLRMAC

CLRMAC



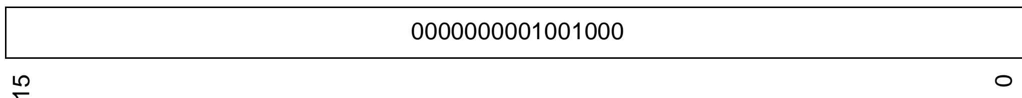

```
macl ← 0;
mach ← 0;
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);
```

Description:

This instruction clears MACL and MACH.

CLRS

CLRS



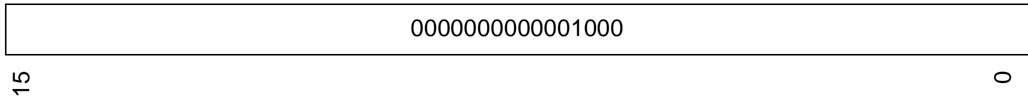

```
s ← 0;
S ← Bit(s);
```

Description:

This instruction clears the S-bit.

CLRT

CLRT



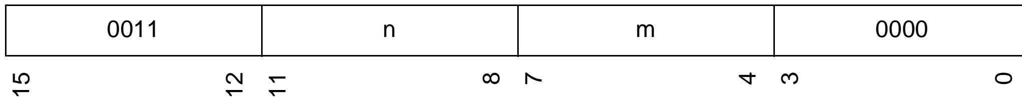
t ← 0;
T ← Bit(t);

Description:

This instruction clears the T-bit.

CMP/EQ Rm, Rn

CMP/EQ Rm, Rn



op1 ← $SignExpect_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
t ← INT (op2 = op1);
T ← Bit(t);

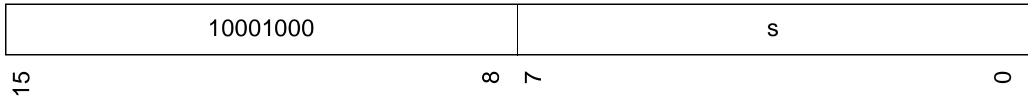
Description:

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.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

CMP/EQ #imm, R0



r0 ← $\text{SignExpect}_{32}(R_0)$;
imm ← $\text{SignExtend}_8(s)$;
t ← INT (r0 = imm);
T ← Bit(t);

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.

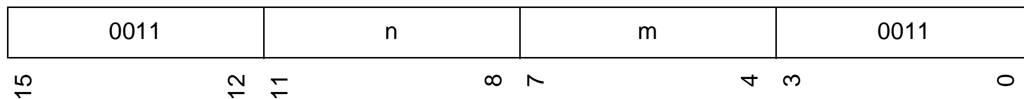
Notes:

The R_0 source is required to have a 32-bit sign-extended representation.

The '#imm' in the assembly syntax represents the immediate s after sign extension.

CMP/GE Rm, Rn

CMP/GE Rm, Rn



op1 ← $\text{SignExpect}_{32}(R_m)$;
op2 ← $\text{SignExpect}_{32}(R_n)$;
t ← INT (op2 ≥ op1);
T ← Bit(t);

Description:

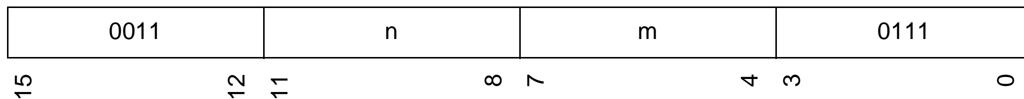
This instruction sets the T-bit if the signed value of R_n is greater than or equal to the signed value of R_m, otherwise it clears the T-bit.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

CMP/GT Rm, Rn

CMP/GT Rm, Rn




```
op1 ← SignExpect32(Rm);
op2 ← SignExpect32(Rn);
t ← INT (op2 > op1);
T ← Bit(t);
```

Description:

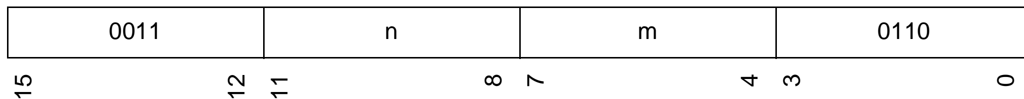
This instruction sets the T-bit if the signed value of R_n is greater than the signed value of R_m, otherwise it clears the T-bit.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

CMP/HI Rm, Rn

CMP/HI Rm, Rn



op1 ← ZeroExtend$_{32}$(SignExpect$_{32}$(R_m));
op2 ← ZeroExtend$_{32}$(SignExpect$_{32}$(R_n));
t ← INT (op2 > op1);
T ← Bit(t);

Description:

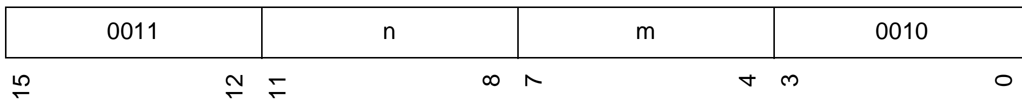
This instruction sets the T-bit if the unsigned value of R_n is greater than the unsigned value of R_m, otherwise it clears the T-bit.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

CMP/HS Rm, Rn

CMP/HS Rm, Rn



$op1 \leftarrow ZeroExtend_{32}(SignExpect_{32}(R_m));$
$op2 \leftarrow ZeroExtend_{32}(SignExpect_{32}(R_n));$
$t \leftarrow INT\ (op2 \geq op1);$
$T \leftarrow Bit(t);$

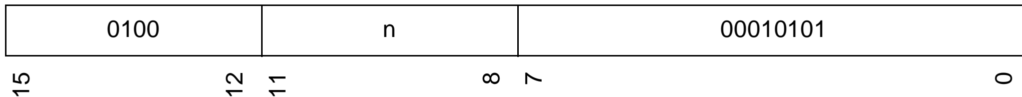
Description:

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.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

CMP/PL Rn



op1 ← $SignExpect_{32}(R_n)$;
t ← INT (op1 > 0);
T ← Bit(t);

Description:

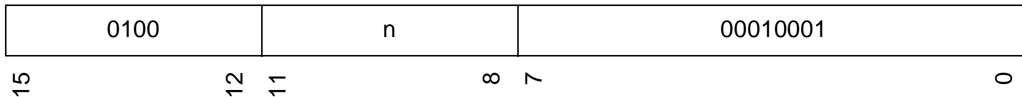
This instruction sets the T-bit if the signed value of R_n is greater than 0, otherwise it clears the T-bit.

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

CMP/PZ Rn

CMP/PZ Rn



0100	n	00010001

15 12 11 8 7 0

op1 ← $SignExpect_{32}(R_n)$;
t ← INT (op1 ≥ 0);
T ← Bit(t);

Description:

This instruction sets the T-bit if the signed value of R_n is greater than or equal to 0, otherwise it clears the T-bit.

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

CMP/STR Rm, Rn

CMP/STR Rm, Rn

0010	n	m	1100
15 12	11 8	7 4	3 0

```
op1 ← SignExpect32(Rm);
op2 ← SignExpect32(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);
```

Description:

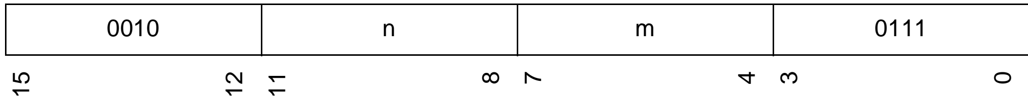
This instruction sets the T-bit if any byte in R_n has the same value as the corresponding byte in R_m, otherwise it clears the T-bit.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

DIV0S Rm, Rn

DIV0S Rm, Rn



op1 $\leftarrow$ $\text{SignExpect}_{32}(R_m)$;
op2 $\leftarrow$ $\text{SignExpect}_{32}(R_n)$;
q $\leftarrow$ $\text{op2}_{< 31 \text{ FOR } 1 >}$;
m $\leftarrow$ $\text{op1}_{< 31 \text{ FOR } 1 >}$;
t $\leftarrow$ m $\oplus$ q;
Q $\leftarrow$ Bit(q);
M $\leftarrow$ Bit(m);
T $\leftarrow$ Bit(t);

Description:

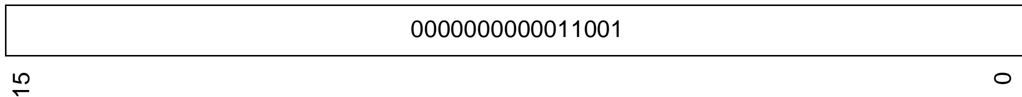
This instruction initializes the divide-step state for a signed division. The Q-bit is initialized with the sign-bit of the dividend, and the M-bit with the sign-bit of the divisor. The T-bit is initialized to 0 if the Q-bit and the M-bit are the same, otherwise it is initialized to 1.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

DIV0U

DIV0U



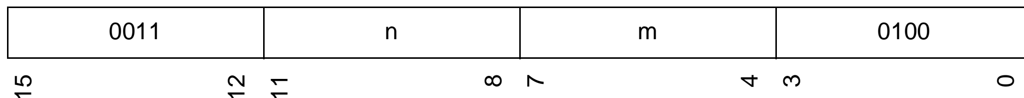

```
q ← 0;
m ← 0;
t ← 0;
Q ← Bit(q);
M ← Bit(m);
T ← Bit(t);
```

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.

DIV1 Rm, Rn

DIV1 Rm, Rn



0011	n	m	0100
15 – 12	11 – 8	7 – 4	3 – 0

```
q ← ZeroExtend_1(Q);
m ← ZeroExtend_1(M);
t ← ZeroExpect_1(T);
op1 ← ZeroExtend_32(SignExpect_32(R_m));
op2 ← ZeroExtend_32(SignExpect_32(R_n));
oldq ← q;
q ← op2_< 31 FOR 1 >;
op2 ← ZeroExtend_32(op2 << 1) ∨ t;
IF (oldq = m)
    op2 ← op2 - op1;
ELSE
    op2 ← op2 + op1;
q ← (q ⊕ m) ⊕ op2_< 32 FOR 1 >;
t ← 1 - (q ⊕ m);
R_n ← Register(SignExtend_32(op2));
Q ← Bit(q);
T ← Bit(t);
```

Description:

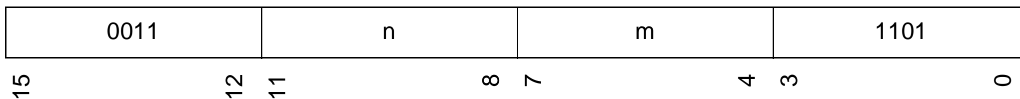
This instruction is used to perform a single-bit divide-step for the division of a dividend held in R_n by a divisor held in R_m. The Q-bit, M-bit and T-bit are used to hold additional state through a divide-step sequence. Each DIV1 consumes 1 bit of the dividend from R_n, and produces 1 bit of result. The divide initialization and step instructions do not detect divide-by-zero nor overflow. If required, these cases should be checked using additional instructions.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation. The T-bit source is required to have a 0 or 1 value.

DMULS.L Rm, Rn

DMULS.L Rm, Rn




```
op1 ← SignExpect32(Rm);
op2 ← SignExpect32(Rn);
mac ← op2 × op1;
macl ← mac;
mach ← mac >> 32;
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);
```

Description:

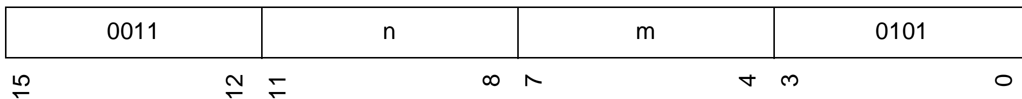
This instruction multiplies the signed 32-bit value held in R_m with the signed 32-bit value held in R_n to give a full 64-bit result. The lower half of the result is placed in MACL and the upper half in MACH.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

DMULU.L Rm, Rn

DMULU.L Rm, Rn



op1 ← $ZeroExtend_{32}(SignExpect_{32}(R_m))$;
op2 ← $ZeroExtend_{32}(SignExpect_{32}(R_n))$;
mac ← op2 × op1;
macl ← mac;
mach ← mac >> 32;
MACL ← $ZeroExtend_{32}$(macl);
MACH ← $ZeroExtend_{32}$(mach);

Description:

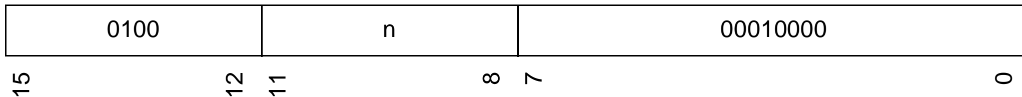
This instruction multiplies the unsigned 32-bit value held in R_m with the unsigned 32-bit value held in R_n to give a full 64-bit result. The lower half of the result is placed in MACL and the upper half in MACH.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

DT Rn

DT Rn




```
op1 ← SignExpect32(Rn);
op1 ← op1 - 1;
t ← INT (op1 = 0);
Rn ← Register(SignExtend32(op1));
T ← Bit(t);
```

Description:

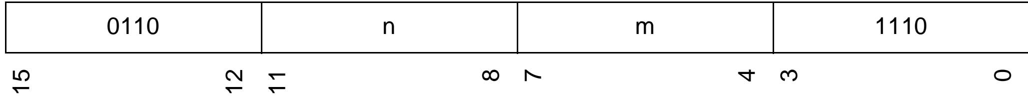
This instruction subtracts 1 from R_n and placed the result in R_n. The T-bit is set if the result is zero, otherwise the T-bit is cleared.

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

EXTS.B Rm, Rn

EXTS.B Rm, Rn



op1 ← $\text{SignExtend}_8(R_m)$;   op2 ← op1; R_n ← $\text{Register}(\text{SignExtend}_{32}(\text{op2}))$;

Description:

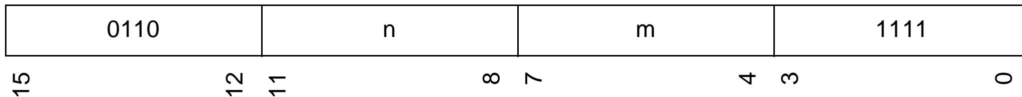
This instruction reads the 8 least significant bits of R_m, sign-extends, and places the result in R_n.

Notes:

The R_m source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

EXTS.W Rm, Rn

EXTS.W Rm, Rn



$op1 \leftarrow SignExtend_{16}(R_m);$
$op2 \leftarrow op1;$
$R_n \leftarrow Register(SignExtend_{32}(op2));$

Description:

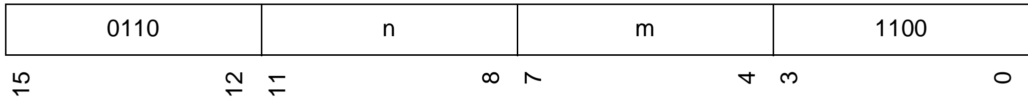
This instruction reads the 16 least significant bits of R_m, sign-extends, and places the result in R_n.

Notes:

The R_m source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

EXTU.B Rm, Rn

EXTU.B Rm, Rn



op1 ← ZeroExtend$_8$(R$_m$);
op2 ← op1;
R$_n$ ← Register(SignExtend$_{32}$(op2));

Description:

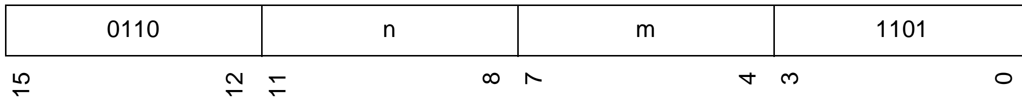
This instruction reads the 8 least significant bits of R_m, zero-extends, and places the result in R_n.

Notes:

The R_m source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

EXTU.W Rm, Rn

EXTU.W Rm, Rn



op1 ← $ZeroExtend_{16}(R_m)$; op2 ← op1; R_n ← Register($SignExtend_{32}$(op2));

Description:

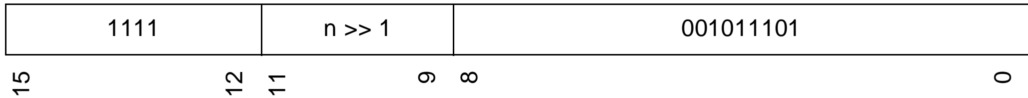
This instruction reads the 16 least significant bits of R_m, zero-extends, and places the result in R_n.

Notes:

The R_m source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

FABS DRn

FABS DRn



Available only when PR=1 and SZ=0

sr ← ZeroExtend$_{64}$(SR);
op1 ← FloatValue$_{64}$(DR$_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
op1 ← FABS_D(op1);
DR$_{FRONT+n}$ ← FloatRegister$_{64}$(op1);

Description:

This floating-point instruction computes the absolute value of a double-precision floating-point number. It reads DR_n, clears the sign bit and places the result in DR_n.

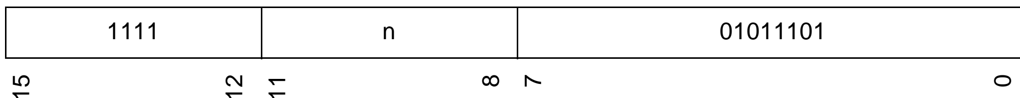
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FABS FRn

FABS FRn



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValue_{32}$($FR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← FABS_S(op1);
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op1);

Description:

This floating-point instruction computes the absolute value of a single-precision floating-point number. It reads FR_n, clears the sign bit and places the result in FR_n.

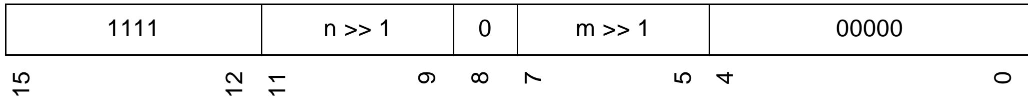
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FADD DRm, DRn

FADD DRm, DRn



Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}$($DR_{FRONT+m}$);
op2 ← $FloatValue_{64}$($DR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FADD FRm, FRn

FADD FRm, FRn

1111	n	m	0000
15 12	11 8	7 4	3 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $FloatValue_{32}$($FR_{FRONT+n}$);
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;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction performs a single-precision floating-point addition. It adds FR_m to FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Possible 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.

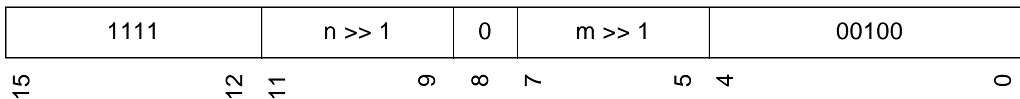
op1 → ↓ op2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	ADD	op2	op2	+INF	-INF	n/a	qNaN	qNaN
+0	op1	+0	+0	+INF	-INF	n/a	qNaN	qNaN
-0	op1	+0	-0	+INF	-INF	n/a	qNaN	qNaN
+INF	+INF	+INF	+INF	+INF	qNaN	n/a	qNaN	qNaN
-INF	-INF	-INF	-INF	qNaN	-INF	n/a	qNaN	qNaN
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a	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 DRm, DRn

FCMP/EQ DRm, DRn



Available only when PR=1 and SZ=0

sr ← ZeroExtend$_{64}$(SR);
fps ← ZeroExtend$_{32}$(FPSCR);
op1 ← FloatValue$_{64}$(DR$_{FRONT+m}$);
op2 ← FloatValue$_{64}$(DR$_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
t, fps ← FCMPEQ_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
FPSCR ← ZeroExtend$_{32}$(fps);
T ← Bit(t);

Description:

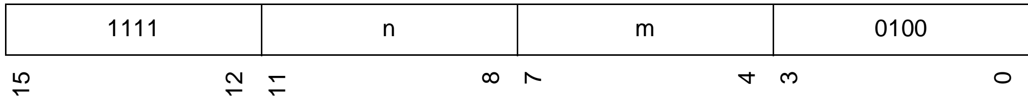
This floating-point instruction performs a double-precision floating-point equality comparison. It sets the T-bit to 1 if DR_m is equal to DR_n, and otherwise sets the T-bit to 0.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FCMP/EQ FRm, FRn

FCMP/EQ FRm, FRn



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $FloatValue_{32}$($FR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPEQ_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← $ZeroExtend_{32}$(fps);
T ← Bit(t);

Description:

This floating-point instruction performs a single-precision floating-point equality comparison. It sets the T-bit to 1 if FR_m is equal to FR_n, and otherwise sets the T-bit to 0.

Possible 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.

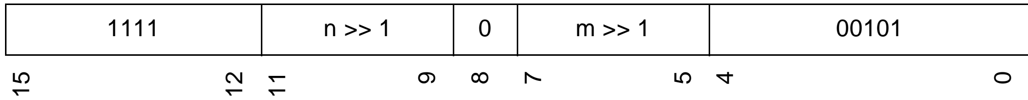
op1 → ↓ op2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	CMPEQ	false	false	false	false	false	false	false
+0	false	true	true	false	false	false	false	false
-0	false	true	true	false	false	false	false	false
+INF	false	false	false	true	false	false	false	false
-INF	false	false	false	false	true	false	false	false
+, -DNRM	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 DRm, DRn

FCMP/GT DRm, DRn



Available only when PR=1 and SZ=0

sr ← ZeroExtend$_{64}$(SR);
fps ← ZeroExtend$_{32}$(FPSCR);
op1 ← FloatValue$_{64}$(DR$_{FRONT+m}$);
op2 ← FloatValue$_{64}$(DR$_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPGT_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend$_{32}$(fps);
T ← Bit(t);

Description:

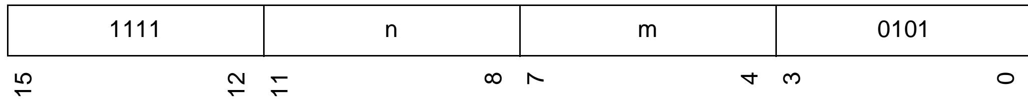
This floating-point instruction performs a double-precision floating-point greater-than comparison. It sets the T-bit to 1 if DR_n is greater than DR_m, and otherwise sets the T-bit to 0.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FCMP/GT FRm, FRn

FCMP/GT FRm, FRn



1111	n	m	0101
15 ... 12	11 ... 8	7 ... 4	3 ... 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}(FR_{FRONT+m})$;
op2 ← $FloatValue_{32}(FR_{FRONT+n})$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
t, fps ← FCMPGT_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
FPSCR ← $ZeroExtend_{32}$(fps);
T ← Bit(t);

Description:

This floating-point instruction performs a single-precision floating-point greater-than comparison. It sets the T-bit to 1 if FR_n is greater than FR_m, and otherwise sets the T-bit to 0.

Possible 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.

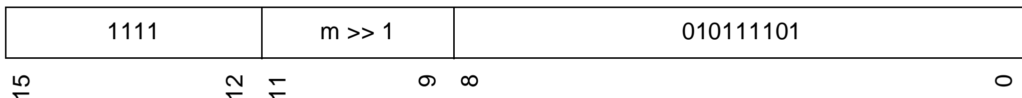
op2 → ↓ op1	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	CMPGT	CMPGT	CMPGT	true	false	CMPGT	false	false
+0	CMPGT	false	false	true	false	CMPGT	false	false
-0	CMPGT	true	false	true	false	CMPGT	false	false
+INF	false	false	false	false	false	false	false	false
-INF	true	true	true	true	false	true	false	false
+, -DNRM	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 DRm, FPUL

FCNVDS DRm, FPUL



Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}$($DR_{FRONT+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(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 ← $ZeroExtend_{32}$(fps);
FPUL ← $ZeroExtend_{32}$(fpul);

Description:

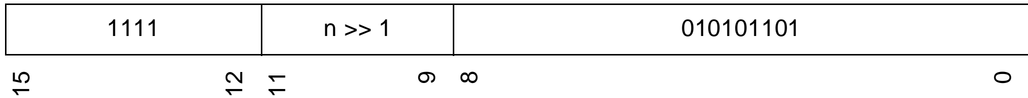
This floating-point instruction performs a double-precision to single-precision floating-point conversion. It reads a double-precision value from DR_m, converts it to single-precision and places the result in FPUL. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FCNVSD FPUL, DRn

FCNVSD FPUL, DRn



1111	n >> 1	010101101
15 12	11 9	8 0

Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
fpul ← $SignExtend_{32}$(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
op1, fps ← FCNV_SD(fpul, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op1);
FPSCR ← $ZeroExtend_{32}$(fps);

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_n. FPSCR.RM has no effect since the conversion is exact.

Possible 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.

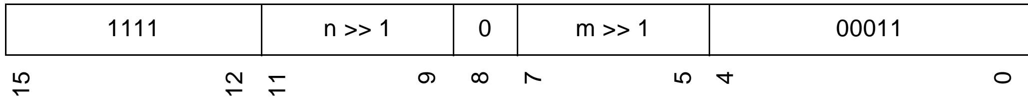
op1 →	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
	CNV	+0	-0	+INF	-INF	n/a	qNaN	qNaN

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

FDIV DRm, DRn



1111	n >> 1	0	m >> 1	00011
15 – 12	11 – 9	8	7 – 5	4 – 0

Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}$($DR_{FRONT+m}$);
op2 ← $FloatValue_{64}$($DR_{FRONT+n}$);
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;
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FDIV FRm, FRn

FDIV FRm, FRn

1111	n	m	0011
15 12	11 8	7 4	3 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $FloatValue_{32}$($FR_{FRONT+n}$);
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;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction performs a single-precision floating-point division. It divides FR_n by FR_m and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Possible 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	+DNRM, -DNRM	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
+, -DNRM	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 FVm, FVn

FIPR FVm, FVn

1111	n >> 2	m >> 2	11101101
15 … 12	11 … 10	9 … 8	7 … 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValueVector_{32}$($FV_{FRONT+m}$);
op2 ← $FloatValueVector_{32}$($FV_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
op2[3], fps ← FIPR_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
THROW FPUEXC, fps;
$FV_{FRONT+n}$ ← $FloatRegisterVector_{32}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction computes dot-product of two vectors, FV_m and FV_n, and places the result in element 3 of FV_n. Each vector contains four single-precision floating-point values. The dot-product is specified as:

$$FR_{n+3} = \sum_{i=0}^{3} FR_{n+i} \times FR_{m+i}$$

This is an approximate computation. The specified error in the result value is defined in *Volume 1, Chapter 13: SHcompact floating-point*.

Possible 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 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. In the following tables, invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown. Inexact is signaled in the 'FIPRADD' case.

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:

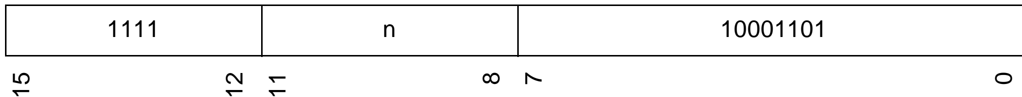
op1 → ↓ op2	+,-NORM, +,-DNRM	+0	-0	+INF	-INF	qNaN	sNaN
+,-NORM and +,-DNRM	FIPRMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
qNaN	qNaN	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 temp0 to temp3) are summed:

↓ temp2	temp0 → / temp1→ ↓ temp3	FIPRMUL, +0, -0			+INF			-INF		
		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

FLDI0 FRn

FLDI0 FRn



1111	n	10001101
15 … 12	11 … 8	7 … 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
op1 ← 0x00000000;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op1);

Description:

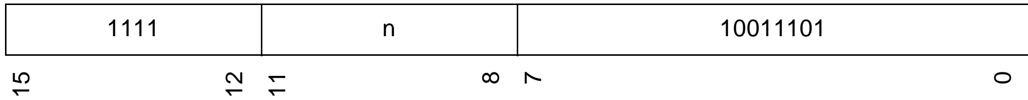
This floating-point instruction loads a constant representing the single-precision floating-point value of 0.0 into FR_n.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FLDI1 FRn

FLDI1 FRn



1111	n	10011101
15 … 12	11 … 8	7 … 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
op1 ← 0x3F800000;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op1);

Description:

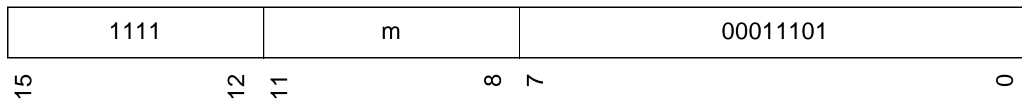
This floating-point instruction loads a constant representing the single-precision floating-point value of 1.0 into FR_n.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FLDS FRm, FPUL

FLDS FRm, FPUL




```
sr ← ZeroExtend64(SR);
op1 ← FloatValue32(FRFRONT+m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul ← op1;
FPUL ← ZeroExtend32(fpul);
```

Description:

This floating-point instruction copies FR_m to FPUL.

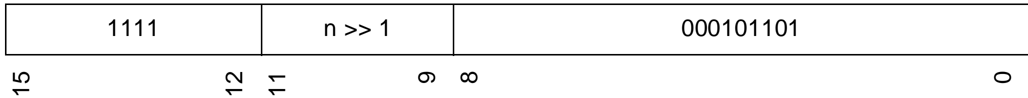
This instruction is not considered an arithmetic operation, and it does not signal invalid operations.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FLOAT FPUL, DRn

FLOAT FPUL, DRn



Available only when PR=1 and SZ=0

fpul ← $SignExtend_{32}$(FPUL);
sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
op1, fps ← FLOAT_LD(fpul, fps);
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op1);

Description:

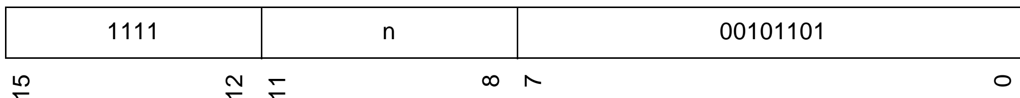
This floating-point instruction performs a signed 32-bit integer to double-precision floating-point conversion. It reads a signed 32-bit integer value from FPUL, converts it to a double-precision range and places the result in DR_n. In all cases the provided integer value will be exactly represented in the destination floating-point format. FPSCR.RM has no effect since the conversion is exact.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FLOAT FPUL, FRn

FLOAT FPUL, FRn



1111	n	00101101
15 – 12	11 – 8	7 – 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
fpul ← $SignExtend_{32}$(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
op1, fps ← FLOAT_LS(fpul, fps);
IF (FpuEnableI(fps))
THROW FPUEXC, fps;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op1);
FPSCR ← $ZeroExtend_{32}$(fps);

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_n. In cases where the integer value cannot be exactly represented in the destination floating-point format, the rounding mode is determined by FPSCR.RM.

Possible 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.

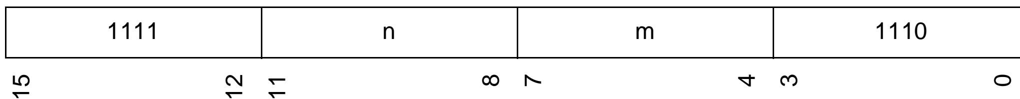
1 Disabled: an exception is raised if the FPU is disabled.

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

FMAC FR0, FRm, FRn



Available only when PR=0

sr ← ZeroExtend_{64}(SR);
fps ← ZeroExtend_{32}(FPSCR);
fr0 ← $\text{FloatValue}_{32}(\text{FR}_{\text{FRONT+0}})$;
op1 ← $\text{FloatValue}_{32}(\text{FR}_{\text{FRONT+m}})$;
op2 ← $\text{FloatValue}_{32}(\text{FR}_{\text{FRONT+n}})$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$\text{FR}_{\text{FRONT+n}}$ ← $\text{FloatRegister}_{32}$(op2);
FPSCR ← ZeroExtend_{32}(fps);

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.

Possible 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. In these 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.

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

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:

↓ op1, fr0 →	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 →	other		qNaN	
↓ op2, op1 →	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 →	+other				-other				+INF				-INF			
op1 → ↓ op2	+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		qNaN

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	+DNRM, -DNRM
+,-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
+, -DNRM	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.

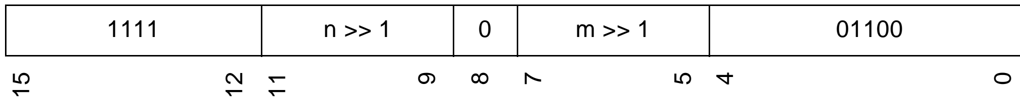
In the 'FULLMUL', +0, -0, +INF and -INF cases, the 2 addition operands (fr0*op1 and op2) are summed:

(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
+, -DNRM	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.

FMOV DRm, DRn

FMOV DRm, DRn



Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValuePair_{32}$($FP_{FRONT+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
op2 ← op1;
$FP_{FRONT+n}$ ← $FloatRegisterPair_{32}$(op2);

Description:

This floating-point instruction reads a pair of single-precision floating-point values from DR_m and copies them to DR_n. This is a bit-by-bit copy with no interpretation or conversion of the values.

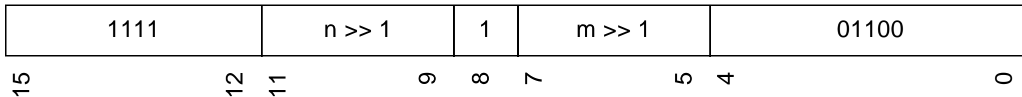
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FMOV DRm, XDn

FMOV DRm, XDn



1111	n >> 1	1	m >> 1	01100
15 – 12	11 – 9	8	7 – 5	4 – 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValuePair_{32}$($FP_{FRONT+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
op2 ← op1;
FP_{BACK+n} ← $FloatRegisterPair_{32}$(op2);

Description:

This floating-point instruction reads a pair of single-precision floating-point values from DR_m and copies them to XD_n. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FMOV DRm, @Rn

FMOV DRm, @Rn

1111	n	m >> 1	01010
15 … 12	11 … 8	7 … 5	4 … 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValuePair_{32}$($FP_{FRONT+m}$);
op2 ← $SignExpect_{32}$(R_n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(op2);
$WriteMemoryPair_{32}$(address, op1);

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.

Possible exceptions:

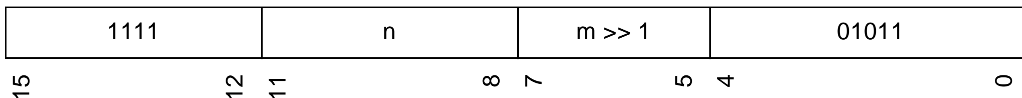
SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

FMOV DRm, @-Rn

FMOV DRm, @-Rn



1111	n	m >> 1	01011
15 … 12	11 … 8	7 … 5	4 … 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValuePair_{32}(FP_{FRONT+m})$;
op2 ← $SignExpect_{32}(R_n)$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(op2 - 8);
$WriteMemoryPair_{32}$(address, op1);
op2 ← address;
R_n ← Register($SignExtend_{32}$(op2));

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

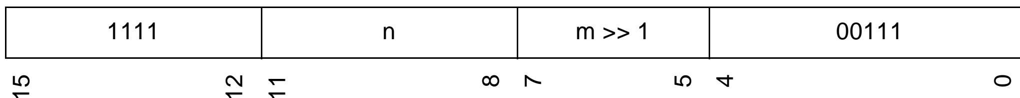
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV DRm, @(R0, Rn)

FMOV DRm, @(R0, Rn)



Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
r0 ← $SignExpect_{32}(R_0)$;
op1 ← $FloatValuePair_{32}(FP_{FRONT+m})$;
op2 ← $SignExpect_{32}(R_n)$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(r0 + op2);
$WriteMemoryPair_{32}$(address, op1);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

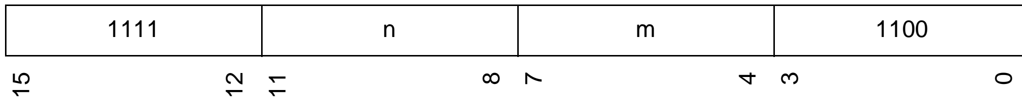
Notes:

The R_0 and R_n sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV FRm, FRn

FMOV FRm, FRn



Available only when SZ=0

sr ← ZeroExtend_{64}(SR);
op1 ← $\text{FloatValue}_{32}(\text{FR}_{\text{FRONT+m}})$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
$\text{FR}_{\text{FRONT+n}}$ ← $\text{FloatRegister}_{32}$(op2);

Description:

This floating-point instruction reads a single-precision floating-point value from FR_m and copies it to FR_n. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FMOV.S FRm, @Rn

FMOV.S FRm, @Rn

1111	n	m	1010
15 12	11 8	7 4	3 0

Available only when SZ=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $SignExpect_{32}$(R_n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(op2);
$WriteMemory_{32}$(address, op1);

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_m is written 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.

Possible exceptions:

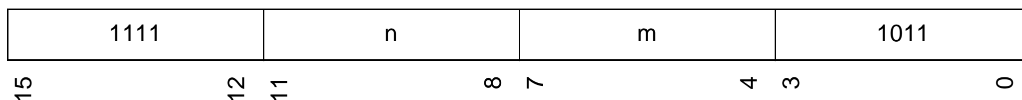
SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

FMOV.S FRm, @-Rn

FMOV.S FRm, @-Rn



Available only when SZ=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $SignExpect_{32}$(R_n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(op2 - 4);
$WriteMemory_{32}$(address, op1);
op2 ← address;
R_n ← Register($SignExtend_{32}$(op2));

Description:

This floating-point instruction stores a single-precision floating-point register 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 FR_m 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

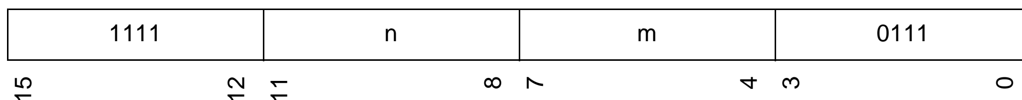
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV.S FRm, @(R0, Rn)

FMOV.S FRm, @(R0, Rn)



1111	n	m	0111
15 12	11 8	7 4	3 0

Available only when SZ=0

sr ← $ZeroExtend_{64}$(SR);
r0 ← $SignExpect_{32}$(R_0);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $SignExpect_{32}$(R_n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(r0 + op2);
$WriteMemory_{32}$(address, op1);

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_0 to R_n. The 32-bit value of FR_m 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

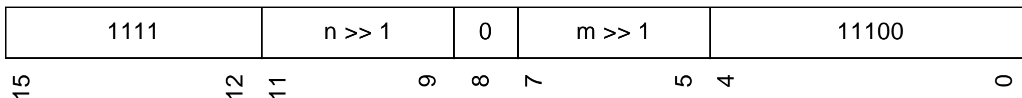
Notes:

The R_0 and R_n sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV XDm, DRn

FMOV XDm, DRn



Available only when PR=0 and SZ=1

sr ← ZeroExtend$_{64}$(SR);
op1 ← FloatValuePair$_{32}$(FP$_{BACK+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
FP$_{FRONT+n}$ ← FloatRegisterPair$_{32}$(op2);

Description:

This floating-point instruction reads a pair of single-precision floating-point values from XD_m and copies them to DR_n. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FMOV XDm, XDn

FMOV XDm, XDn

1111	n >> 1	1	m >> 1	11100
15 – 12	11 – 9	8	7 – 5	4 – 0

Available only when PR=0 and SZ=1

sr ← ZeroExtend$_{64}$(SR);
op1 ← FloatValue$_{64}$(DR$_{BACK+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
DR$_{BACK+n}$ ← FloatRegister$_{64}$(op2);

Description:

This floating-point instruction reads a pair of single-precision floating-point values from XD_m and copies them to XD_n. This is a bit-by-bit copy with no interpretation or conversion of the values.

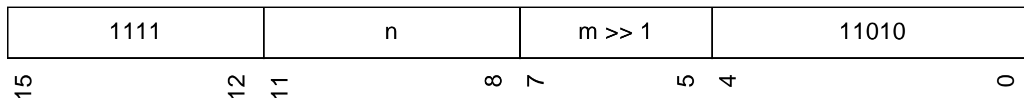
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FMOV XDm, @Rn

FMOV XDm, @Rn



1111	n	m >> 1	11010
15 … 12	11 … 8	7 … 5	4 … 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValuePair_{32}$(FP_{BACK+m});
op2 ← $SignExpect_{32}$(R_n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(op2);
$WriteMemoryPair_{32}$(address, op1);

Description:

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with zero-displacement addressing. XD_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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

FMOV XDm, @-Rn

FMOV XDm, @-Rn

1111	n	m >> 1	11011
15 — 12	11 — 8	7 — 5	4 — 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValuePair_{32}$(FP_{BACK+m});
op2 ← $SignExpect_{32}$(R_n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(op2 - 8);
$WriteMemoryPair_{32}$(address, op1);
op2 ← address;
R_n ← Register($SignExtend_{32}$(op2));

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

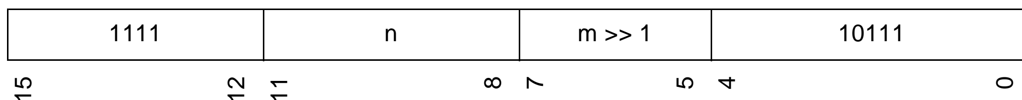
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV XDm, @(R0, Rn)

FMOV XDm, @(R0, Rn)



Available only when PR=0 and SZ=1

sr ← ZeroExtend$_{64}$(SR);
r0 ← SignExpect$_{32}$(R$_0$);
op1 ← FloatValuePair$_{32}$(FP$_{BACK+m}$);
op2 ← SignExpect$_{32}$(R$_n$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← ZeroExtend$_{64}$(r0 + op2);
WriteMemoryPair$_{32}$(address, op1);

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. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

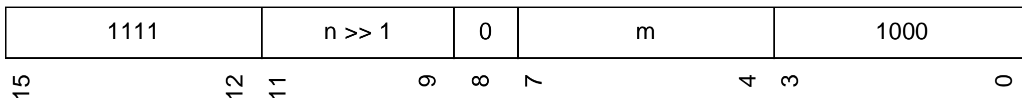
Notes:

The R_0 and R_n sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV @Rm, DRn

FMOV @Rm, DRn



Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $SignExpect_{32}$(R_m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(op1);
op2 ← $ReadMemoryPair_{32}$(address);
$FP_{FRONT+n}$ ← $FloatRegisterPair_{32}$(op2);

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.

Possible exceptions:

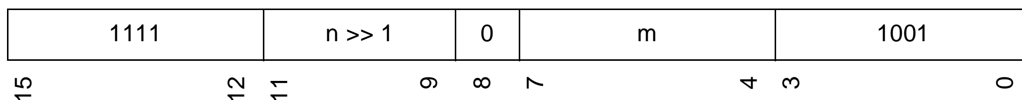
SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

FMOV @Rm+, DRn

FMOV @Rm+, DRn



1111	n >> 1	0	m	1001
15 12	11 9	8	7 4	3 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $SignExpect_{32}$(R_m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(op1);
op2 ← $ReadMemoryPair_{32}$(address);
op1 ← op1 + 8;
R_m ← Register($SignExtend_{32}$(op1));
$FP_{FRONT+n}$ ← $FloatRegisterPair_{32}$(op2);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

FMOV @(R0, Rm), DRn

FMOV @(R0, Rm), DRn

1111	n >> 1	0	m	0110
15 12	11 9	8	7 4	3 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
r0 ← $SignExpect_{32}(R_0)$;
op1 ← $SignExpect_{32}(R_m)$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(r0 + op1);
op2 ← $ReadMemoryPair_{32}$(address);
$FP_{FRONT+n}$ ← $FloatRegisterPair_{32}$(op2);

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_m. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

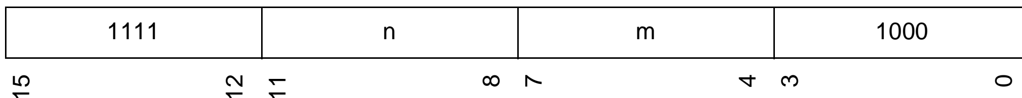
Notes:

The R_0 and R_m sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV.S @Rm, FRn

FMOV.S @Rm, FRn



Available only when SZ=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $SignExpect_{32}$(R_m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(op1);
op2 ← $ReadMemory_{32}$(address);
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);

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.

Possible exceptions:

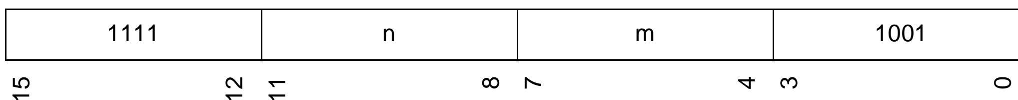
SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

FMOV.S @Rm+, FRn

FMOV.S @Rm+, FRn



Available only when SZ=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $SignExpect_{32}$(R_m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(op1);
op2 ← $ReadMemory_{32}$(address);
op1 ← op1 + 4;
R_m ← Register($SignExtend_{32}$(op1));
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

FMOV.S @(R0, Rm), FRn

FMOV.S @(R0, Rm), FRn

1111	n	m	0110
15 12	11 8	7 4	3 0

Available only when SZ=0

sr ← $ZeroExtend_{64}$(SR);
r0 ← $SignExpect_{32}(R_0)$;
op1 ← $SignExpect_{32}(R_m)$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
address ← $ZeroExtend_{64}$(r0 + op1);
op2 ← $ReadMemory_{32}$(address);
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);

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_m. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

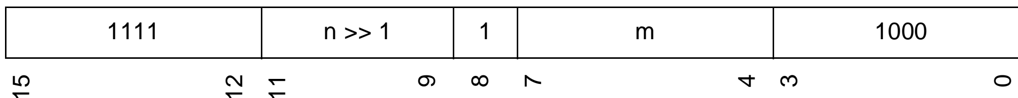
Notes:

The R_0 and R_m sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMOV @Rm, XDn

FMOV @Rm, XDn



Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
op1 ← $SignExpect_{32}(R_m)$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(op1);
op2 ← $ReadMemoryPair_{32}$(address);
FP_{BACK+n} ← $FloatRegisterPair_{32}$(op2);

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.

Possible exceptions:

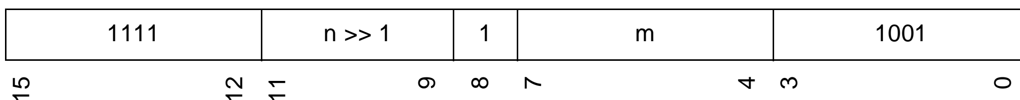
SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

FMOV @Rm+, XDn

FMOV @Rm+, XDn



1111	n >> 1	1	m	1001
15 … 12	11 … 9	8	7 … 4	3 … 0

Available only when PR=0 and SZ=1

sr ← ZeroExtend$_{64}$(SR);
op1 ← SignExpect$_{32}$(R$_m$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend$_{64}$(op1);
op2 ← ReadMemoryPair$_{32}$(address);
op1 ← op1 + 8;
R$_m$ ← Register(SignExtend$_{32}$(op1));
FP$_{BACK+n}$ ← FloatRegisterPair$_{32}$(op2);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

FMOV @(R0, Rm), XDn

FMOV @(R0, Rm), XDn

1111	n >> 1	1	m	0110
15 – 12	11 – 9	8	7 – 4	3 – 0

Available only when PR=0 and SZ=1

sr ← $ZeroExtend_{64}$(SR);
r0 ← $SignExpect_{32}$(R_0);
op1 ← $SignExpect_{32}$(R_m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← $ZeroExtend_{64}$(r0 + op1);
op2 ← $ReadMemoryPair_{32}$(address);
FP_{BACK+n} ← $FloatRegisterPair_{32}$(op2);

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_m. 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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_0 and R_m sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

FMUL DRm, DRn

FMUL DRm, DRn

1111	n >> 1	0	m >> 1	00010
15 … 12	11 … 9	8	7 … 5	4 … 0

Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}(DR_{FRONT+m})$;
op2 ← $FloatValue_{64}(DR_{FRONT+n})$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

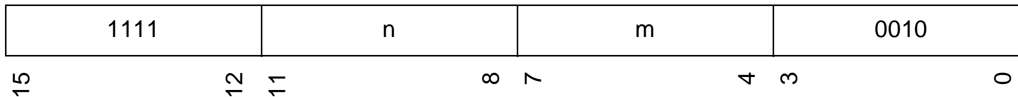
This floating-point instruction performs a double-precision floating-point multiplication. It multiplies DR_m by DR_n and places the result in DR_n. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FMUL FRm, FRn

FMUL FRm, FRn



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}(FR_{FRONT+m})$;
op2 ← $FloatValue_{32}(FR_{FRONT+n})$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction performs a single-precision floating-point multiplication. It multiplies FR_m by FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Possible 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.

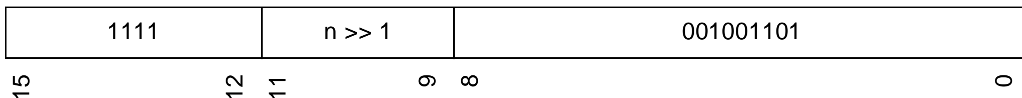
op1 → ↓ op2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	MUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	n/a	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	n/a	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	n/a	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	n/a	qNaN	qNaN
+, -DNRM	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 DRn

FNEG DRn



Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValue_{64}$($DR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← FNEG_D(op1);
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op1);

Description:

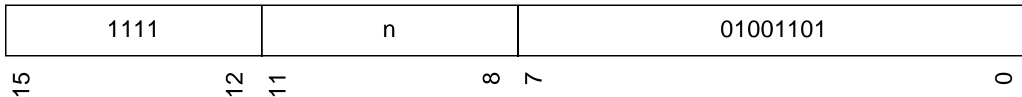
This floating-point instruction computes the negated value of a double-precision floating-point number. It reads DR_n, inverts the sign bit and places the result in DR_n.

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FNEG FRn



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
op1 ← $FloatValue_{32}$($FR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
op1 ← FNEG_S(op1);
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op1);

Description:

This floating-point instruction computes the negated value of a single-precision floating-point number. It reads FR_n, inverts the sign bit and places the result in FR_n.

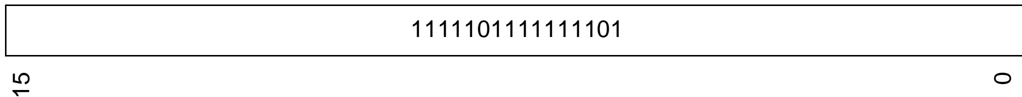
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FRCHG

FRCHG



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fr ← $ZeroExtend_{1}$(SR.FR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fr ← fr ⊕ 1;
SR.FR ← Bit(fr);

Description:

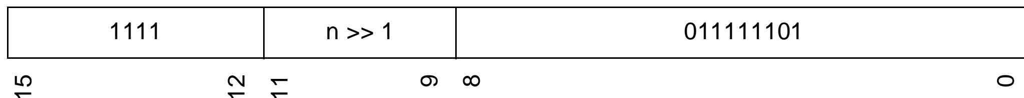
This floating-point instruction toggles the FPSCR.FR bit. This has the effect of switching the basic and extended banks of the floating-point register file.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FSCA FPUL, DRn

FSCA FPUL, DRn



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
fpul ← $SignExtend_{32}$(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
op1[0], fps ← FSINA_S(fpul, fps);
op1[1], fps ← FCOSA_S(fpul, fps);
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
$FP_{FRONT+n}$ ← $FloatRegisterPair_{32}$(op1);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction computes the sine and cosine of an angle stored in FPUL. The lower register in DR_n returns the sine of the angle in single-precision floating-point format. The upper register in DR_n returns the cosine of the angle in single-precision floating-point format. The input angle is the amount of rotation expressed as a signed fixed-point number in a 2's complement representation. The value 1 represents an angle of $360^o/2^{16}$. The upper 16 bits indicate the number of full rotations and the lower 16 bits indicate the remainder angle between 0° and 360°. This is an approximate computation. The specified error in the result value is:

spec_error = 2^{-21}.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FSCA special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

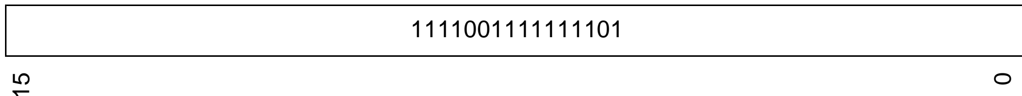
1 Disabled: an exception is raised if the FPU is disabled.

2 Inexact: this is an approximate instruction and inexact is always signaled. 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, the instruction computes an approximate result using an implementation-dependent algorithm.

FSCHG

FSCHG



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
sz ← $ZeroExtend_{1}$(SR.SZ);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
sz ← sz ⊕ 1;
SR.SZ ← Bit(sz);

Description:

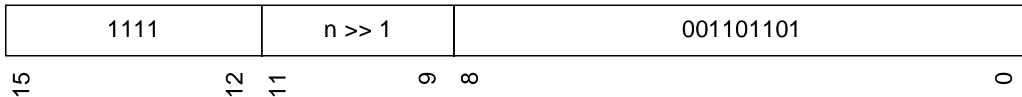
This floating-point instruction toggles the FPSCR.SZ bit. This has the effect of changing the size of the data transfer for subsequent floating-point loads, stores and moves. Two transfer sizes are available: FPSCR.SZ = 0 indicates 32-bit transfer and FPSCR.SZ = 1 indicates 64-bit transfer.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FSQRT DRn

FSQRT DRn



1111	n >> 1	001101101
15 – 12	11 – 9	8 – 0

Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}$($DR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op1);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

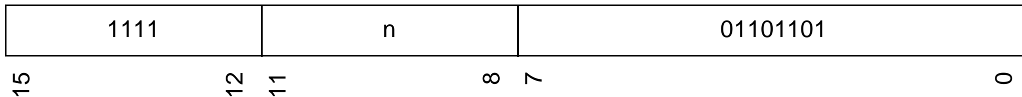
This floating-point instruction performs a double-precision floating-point square root. It extracts the square root of DR_n and places the result in DR_n. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FSQRT FRn

FSQRT FRn



Available only when PR=0

sr ← ZeroExtend$_{64}$(SR);
fps ← ZeroExtend$_{32}$(FPSCR);
op1 ← FloatValue$_{32}$(FR$_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
FR$_{FRONT+n}$ ← FloatRegister$_{32}$(op1);
FPSCR ← ZeroExtend$_{32}$(fps);

Description:

This floating-point instruction performs a single-precision floating-point square root. It extracts the square root of FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Possible 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.

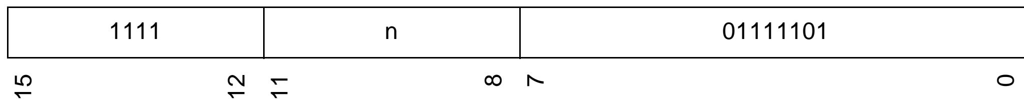
op1 →	+NORM	-NORM	+0	-0	+INF	-INF	+DNRM	-DNRM	qNaN	sNaN
	SQRT	qNaN	+0	-0	+INF	qNaN	n/a	qNaN	qNaN	qNaN

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.

FSRRA FRn

FSRRA FRn



Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}$($FR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
op1, fps ← FSRRA_S(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))
THROW FPUEXC, fps;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op1);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction computes the reciprocal of the square root of the value stored in FR_n and places the result in FR_n. This is an approximate computation. The specified error in the result value is:

spec_error = 2^{E-21}, where E = unbiased exponent value of the result.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FSRRA 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 reciprocal square root of a number less than zero (including negative infinity and negative normalized/denormalized numbers, but excluding negative zero).

3 Divide-by-zero: a divide-by-zero is signaled if this is a reciprocal square root of zero (regardless of the sign of the zero).

4 Error: an FPU error is signaled if FPSCR.DN is 0 and the input is a positive denormalized number.

5 Inexact: this is an approximate instruction and inexact is signaled if this is a reciprocal square root of a positive normalized non-zero finite number. Inexact is not signaled if the input is a negative normalized number, a zero, an infinity, a denormalized number or a NaN. 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. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

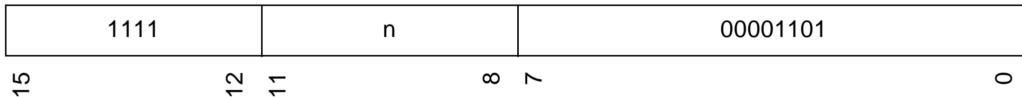
op1 →	+NORM	-NORM	+0	-0	+INF	-INF	+DNRM	-DNRM	qNaN	sNaN
	SRRA	qNaN	+INF	-INF	+0	qNaN	n/a	qNaN	qNaN	qNaN

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 and inexact cases are not shown.

The normal 'SRRA' case uses an implementation-specific algorithm to calculate an approximation of the reciprocal square root of op1.

FSTS FPUL, FRn

FSTS FPUL, FRn




```
sr ← ZeroExtend64(SR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← fpul;
FRFRONT+n ← FloatRegister32(op1);
```

Description:

This floating-point instruction copies FPUL to FR_n.

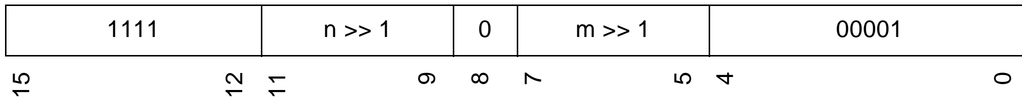
This instruction is not considered an arithmetic operation, and it does not signal invalid operations.

Possible exceptions:

SLOTFPUDIS, FPUDIS

FSUB DRm, DRn

FSUB DRm, DRn



Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}$($DR_{FRONT+m}$);
op2 ← $FloatValue_{64}$($DR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$DR_{FRONT+n}$ ← $FloatRegister_{64}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FSUB FRm, FRn

FSUB FRm, FRn

1111	n	m	0001
15 12	11 8	7 4	3 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
op2 ← $FloatValue_{32}$($FR_{FRONT+n}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$FR_{FRONT+n}$ ← $FloatRegister_{32}$(op2);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction performs a single-precision floating-point subtraction. It subtracts FR_m from FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Possible 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.

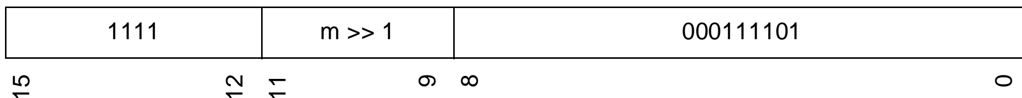
op2 → ↓ op1	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	SUB	SUB	SUB	+INF	-INF	n/a	qNaN	qNaN
+0	op2	+0	-0	+INF	-INF	n/a	qNaN	qNaN
-0	op2	+0	+0	+INF	-INF	n/a	qNaN	qNaN
+INF	-INF	-INF	-INF	qNaN	-INF	n/a	qNaN	qNaN
-INF	+INF	+INF	+INF	+INF	qNaN	n/a	qNaN	qNaN
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a	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 DRm, FPUL

FTRC DRm, FPUL



Available only when PR=1 and SZ=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{64}$($DR_{FRONT+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul, fps ← FTRC_DL(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPUL ← $ZeroExtend_{32}$(fpul);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

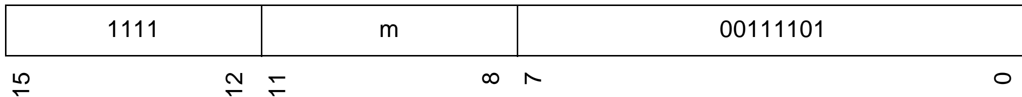
This floating-point instruction performs a double-precision floating-point to signed 32-bit integer conversion. It reads a double-precision value from DR_m, converts it to a signed 32-bit integral range and places the result in FPUL. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

Possible exceptions:

SLOTFPUDIS, FPUDIS, FPUEXC

FTRC FRm, FPUL

FTRC FRm, FPUL



1111	m	00111101
15 — 12	11 — 8	7 — 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
op1 ← $FloatValue_{32}$($FR_{FRONT+m}$);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
THROW FPUDIS;
fpul, fps ← FTRC_SL(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
THROW FPUEXC, fps;
FPUL ← $ZeroExtend_{32}$(fpul);
FPSCR ← $ZeroExtend_{32}$(fps);

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_m, 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.

Possible 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, -NORM (in range)	+0	-0	+INF or +NORM (out of range)	-INF or -NORM (out of range)	+DNRM, -DNRM	qNaN	sNaN
	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

FTRV XMTRX, FVn

1111	n >> 2	0111111101
15 … 12	11 … 10	9 … 0

Available only when PR=0

sr ← $ZeroExtend_{64}$(SR);
fps ← $ZeroExtend_{32}$(FPSCR);
xmtrx ← $FloatValueMatrix_{32}(MTRX_{BACK})$;
op1 ← $FloatValueVector_{32}(FV_{FRONT+n})$;
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(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;
$FV_{FRONT+n}$ ← $FloatRegisterVector_{32}$(op1);
FPSCR ← $ZeroExtend_{32}$(fps);

Description:

This floating-point instruction multiplies the matrix, XMTRX, with a vector, FV_n, and places the resulting vector in FV_n. 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:

$$FR_{n+0} = \sum_{i=0}^{3} XF_{i \times 4} \times FR_{n+i}$$

$$FR_{n+1} = \sum_{i=0}^{3} XF_{1+i \times 4} \times FR_{n+i}$$

$$FR_{n+2} = \sum_{i=0}^{3} XF_{2+i\times4} \times FR_{n+i}$$

$$FR_{n+3} = \sum_{i=0}^{3} XF_{3+i\times4} \times FR_{n+i}$$

This is an approximate computation. The specified error in the result value is defined in *Volume 1, Chapter 13: SHcompact floating-point*.

Possible 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 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 does not check all of its inputs for invalid operations and then raise an exception accordingly. If invalid operation exceptions are requested by the user, this instruction always raises that exception. If this exception is not

requested by the user, then each of the four inner-products is checked separately for an invalid operation (as described above) and the appropriate result is set to qNaN for each inner-product that is invalid.

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. 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, +,-DNRM	+0	-0	+INF	-INF	qNaN	sNaN
+,-NORM and +,-DNRM	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

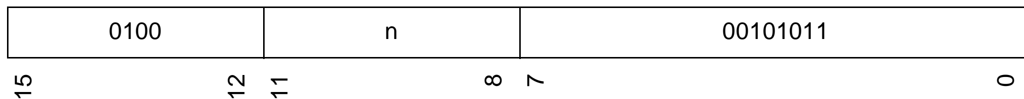
If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the ‘FTRVMUL’, +0, -0, +INF and -INF cases, the 4 addition operands (labelled temp0 to temp3) are summed:

	temp0 →	FTRVMUL, +0, -0			+INF			-INF		
↓ temp2	temp1→ ↓ temp3	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

JMP @Rn




```
op1 ← SignExpect32(Rn);
IF (IsDelaySlot())
   THROW ILLSLOT;
target ← op1;
IF ((target ∧ 0x3) = 0x3)
   THROW IADDERR, target;
delayedisa ← target ∧ 0x1;
delayedpc ← target ∧ (~ 0x1);
PC” ← Register(SignExtend32(delayedpc));
ISA” ← Bit(delayedisa);
```

Description:

This instruction is a delayed unconditional branch used for jumping to the target address specified in R_n. If the last two bits of the target address are both set, an IADDERR exception is raised. Otherwise, the delay slot is executed in SHcompact. Bit zero of the target address gives the new value of the ISA mode for the next instruction. The least significant bit of the target address is cleared, and this value is copied to the PC.

Possible exceptions:

ILLSLOT, IADDERR

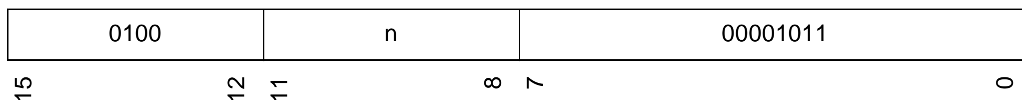
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The delay slot is executed before branching and before ISA is updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

JSR @Rn

JSR @Rn




```
pc ← SignExpect32(PC);
op1 ← SignExpect32(Rn);
IF (IsDelaySlot())
   THROW ILLSLOT;
delayedpr ← pc + 4;
target ← op1;
IF ((target ∧ 0x3) = 0x3)
   THROW IADDERR, target;
delayedisa ← target ∧ 0x1;
delayedpc ← target ∧ (~ 0x1);
PR” ← Register(SignExtend32(delayedpr));
PC” ← Register(SignExtend32(delayedpc));
ISA” ← Bit(delayedisa);
```

Description:

This instruction is a delayed unconditional branch used for jumping to the subroutine starting at the target address specified in R_n. If the last two bits of the target address are both set, an IADDERR exception is raised. Otherwise, the delay slot is executed in SHcompact. Bit zero of the target address gives the new value of the ISA mode for the next instruction. The least significant bit of the target address is cleared, and this value is copied to the PC. The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

Possible exceptions:

ILLSLOT, IADDERR

Notes:

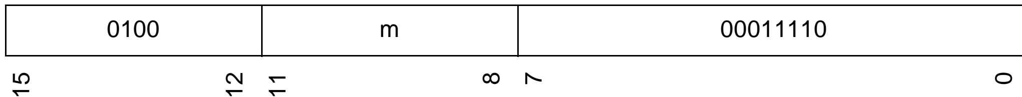
The R_n source is required to have a 32-bit sign-extended representation.

If this instruction does not raise an exception then PR will be updated regardless of whether the delay slot instruction raises an exception. The delay slot is executed

before branching and before ISA and PR are updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

LDC Rm, GBR

LDC Rm, GBR



op1 ← $SignExtend_{32}(R_m)$;
gbr ← op1;
GBR ← Register($SignExtend_{32}$(gbr));

Description:

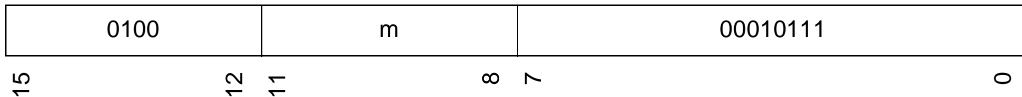
This instruction copies R_m to GBR.

Notes:

The R_m source is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

LDC.L @Rm+, GBR

LDC.L @Rm+, GBR



$op1 \leftarrow SignExpect_{32}(R_m);$
$address \leftarrow ZeroExtend_{64}(op1);$
$gbr \leftarrow SignExtend_{32}(ReadMemory_{32}(address));$
$op1 \leftarrow op1 + 4;$
$R_m \leftarrow Register(SignExtend_{32}(op1));$
$GBR \leftarrow Register(SignExtend_{32}(gbr));$

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.

Possible exceptions:

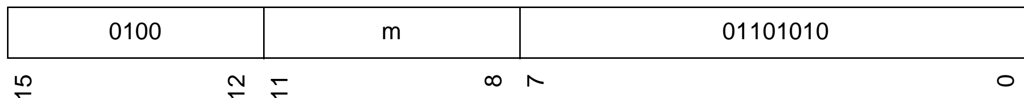
RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

LDS Rm, FPSCR

LDS Rm, FPSCR




```
sr ← ZeroExtend64(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fps, pr, sz, fr ← UnpackFPSCR(op1);
FPSCR ← ZeroExtend32(fps);
SR.PR ← Bit(pr);
SR.SZ ← Bit(sz);
SR.FR ← Bit(fr);
```

Description:

This floating-point instruction copies R_m to FPSCR. The setting of FPSCR does not cause any floating-point exceptional conditions to be signaled.

Possible exceptions:

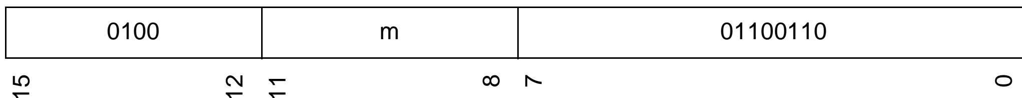
SLOTFPUDIS, FPUDIS

Notes:

The R_m source is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

LDS.L @Rm+, FPSCR

LDS.L @Rm+, FPSCR



0100	m	01100110

15 12 11 8 7 0

```
sr ← ZeroExtend64(SR);
op1 ← SignExpect32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend64(op1);
value ← ReadMemory32(address);
fps, pr, sz, fr ← UnpackFPSCR(value);
op1 ← op1 + 4;
Rm ← Register(SignExtend32(op1));
FPSCR ← ZeroExtend32(fps);
SR.PR ← Bit(pr);
SR.SZ ← Bit(sz);
SR.FR ← Bit(fr);
```

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.

Possible exceptions:

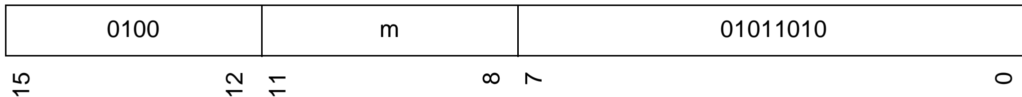
SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

LDS Rm, FPUL

LDS Rm, FPUL




```
sr ← ZeroExtend64(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul ← op1;
FPUL ← ZeroExtend32(fpul);
```

Description:

This floating-point instruction copies R_m to FPUL.

Possible exceptions:

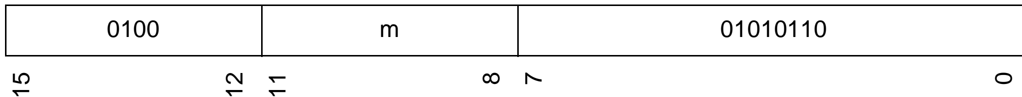
SLOTFPUDIS, FPUDIS

Notes:

The R_m source is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

LDS.L @Rm+, FPUL

LDS.L @Rm+, FPUL




```
sr ← ZeroExtend64(SR);
op1 ← SignExpect32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend64(op1);
fpul ← ReadMemory32(address);
op1 ← op1 + 4;
Rm ← Register(SignExtend32(op1));
FPUL ← ZeroExtend32(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.

Possible exceptions:

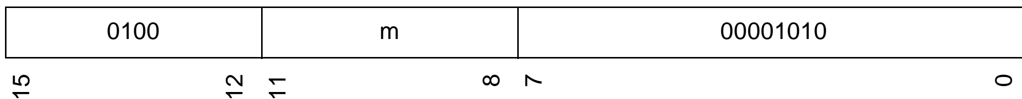
SLOTFPUDIS, FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

LDS Rm, MACH

LDS Rm, MACH



op1 ← $\text{SignExtend}_{32}(R_m)$;
mach ← op1;
MACH ← $\text{ZeroExtend}_{32}(\text{mach})$;

Description:

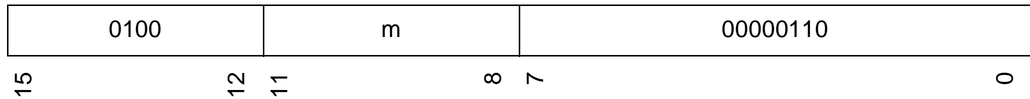
This instruction copies R_m to MACH.

Notes:

The R_m source is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

LDS.L @Rm+, MACH

LDS.L @Rm+, MACH



op1 ← SignExpect$_{32}$(R$_m$);
address ← ZeroExtend$_{64}$(op1);
mach ← SignExtend$_{32}$(ReadMemory$_{32}$(address));
op1 ← op1 + 4;
R$_m$ ← Register(SignExtend$_{32}$(op1));
MACH ← ZeroExtend$_{32}$(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.

Possible exceptions:

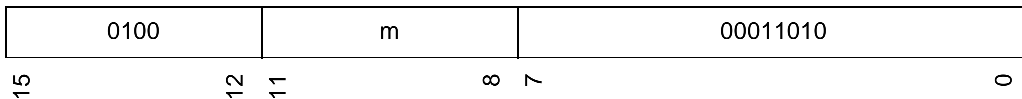
RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

LDS Rm, MACL

LDS Rm, MACL



op1 ← $SignExtend_{32}(R_m)$;
macl ← op1;
MACL ← $ZeroExtend_{32}$(macl);

Description:

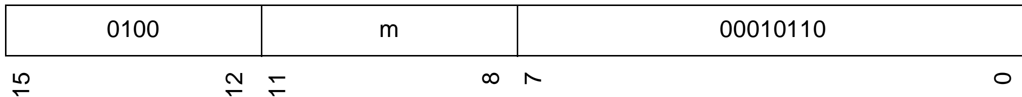
This instruction copies R_m to MACL.

Notes:

The R_m source is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

LDS.L @Rm+, MACL

LDS.L @Rm+, MACL



op1 ← SignExpect$_{32}$(R_m);
address ← ZeroExtend$_{64}$(op1);
macl ← SignExtend$_{32}$(ReadMemory$_{32}$(address));
op1 ← op1 + 4;
R_m ← Register(SignExtend$_{32}$(op1));
MACL ← ZeroExtend$_{32}$(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.

Possible exceptions:

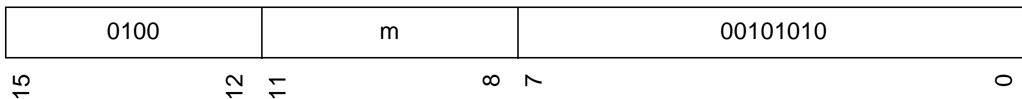
RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

LDS Rm, PR

LDS Rm, PR



$op1 \leftarrow SignExtend_{32}(R_m);$
$newpr \leftarrow op1;$
$delayedpr \leftarrow newpr;$
$PR' \leftarrow Register(SignExtend_{32}(newpr));$
$PR'' \leftarrow Register(SignExtend_{32}(delayedpr));$

Description:

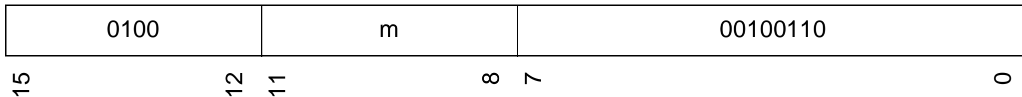
This instruction copies R_m to PR.

Notes:

The R_m source is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

LDS.L @Rm+, PR

LDS.L @Rm+, PR



op1 ← $SignExpect_{32}(R_m)$;
address ← $ZeroExtend_{64}$(op1);
newpr ← $SignExtend_{32}(ReadMemory_{32}$(address));
delayedpr ← newpr;
op1 ← op1 + 4;
R_m ← Register($SignExtend_{32}$(op1));
PR’ ← Register($SignExtend_{32}$(newpr));
PR” ← Register($SignExtend_{32}$(delayedpr));

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

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.

Possible exceptions:

All exception checks on the R_n operand are performed before any of the exception checks on the R_m operand. The exception checks for each operand are in the usual precedence order. However, the overall order for the MAC.L exceptions is:

RADDERR, RTLBMISS, READPROT (for R_n access)

followed by:

RADDERR, RTLBMISS, READPROT (for R_m access)

which differs from the usual precedence order.

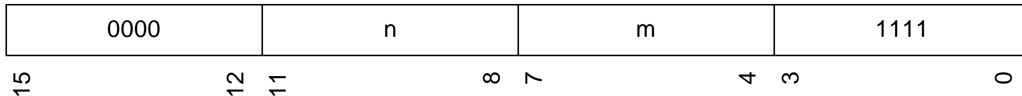
Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

If R_m and R_n refer to the same register (that is, m = n), then this register will be post-incremented twice. The instruction will read two long-words from consecutive memory locations.

MAC.L @Rm+, @Rn+



macl ← $ZeroExtend_{32}$(MACL);
mach ← $ZeroExtend_{32}$(MACH);
s ← $ZeroExtend_{1}$(S);
m_field ← $ZeroExtend_{4}$(m);
n_field ← $ZeroExtend_{4}$(n);
m_address ← $SignExpect_{32}(R_m)$;
n_address ← $SignExpect_{32}(R_n)$;
value2 ← $SignExtend_{32}$($ReadMemory_{32}$($ZeroExtend_{64}$(n_address)));
n_address ← n_address + 4;
IF (n_field = m_field)
{
    m_address ← m_address + 4;
    n_address ← n_address + 4;
}
value1 ← $SignExtend_{32}$($ReadMemory_{32}$($ZeroExtend_{64}$(m_address)));
m_address ← m_address + 4;
mul ← value2 × value1;
mac ← (mach << 32) + macl;
result ← mac + mul;
IF (s = 1)
    IF $(((result \oplus mac) \wedge (result \oplus mul))_{< 63 \text{ FOR } 1 >} = 1)$
        IF $(mac_{< 63 \text{ FOR } 1 >} = 0)$
            result ← $2^{47} - 1$;
        ELSE
            result ← $- 2^{47}$;
    ELSE
        result ← $SignedSaturate_{48}$(result);
macl ← result;
mach ← result >> 32;
R_m ← Register($SignExtend_{32}$(m_address));
R_n ← Register($SignExtend_{32}$(n_address));
MACL ← $ZeroExtend_{32}$(macl);
MACH ← $ZeroExtend_{32}$(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.

Possible exceptions:

All exception checks on the R_n operand are performed before any of the exception checks on the R_m operand. The exception checks for each operand are in the usual precedence order. However, the overall order for the MAC.W exceptions is:

RADDERR, RTLBMISS, READPROT (for R_n access)

followed by:

RADDERR, RTLBMISS, READPROT (for R_m access)

which differs from the usual precedence order.

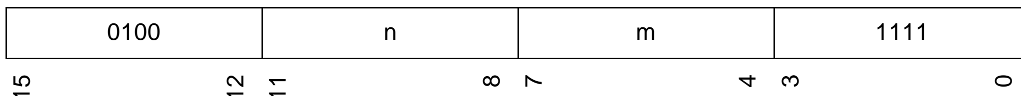
Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

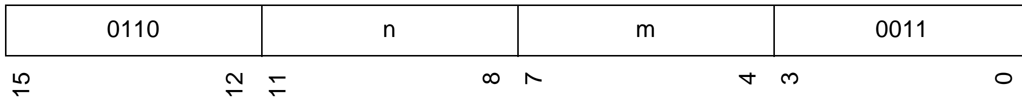
If R_m and R_n refer to the same register (that is, m = n), then this register will be post-incremented twice. The instruction will read two words from consecutive memory locations.

MAC.W @Rm+, @Rn+




```
macl ← ZeroExtend32(MACL);
mach ← ZeroExtend32(MACH);
s ← ZeroExtend1(S);
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
m_address ← SignExpect32(Rm);
n_address ← SignExpect32(Rn);
value2 ← SignExtend16(ReadMemory16(ZeroExtend64(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(ZeroExtend64(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(SignExtend32(m_address));
Rn ← Register(SignExtend32(n_address));
MACL ← ZeroExtend32(macl);
MACH ← ZeroExtend32(mach);
```

MOV Rm, Rn



op1 ← $ZeroExtend_{64}(R_m)$;
op2 ← op1;
R_n ← Register(op2);

Description:

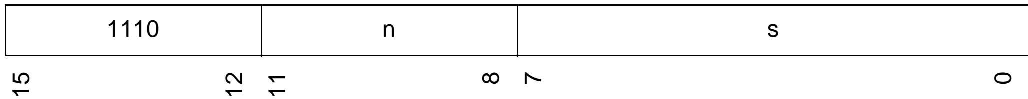
This instruction copies the value of R_m to R_n.

Notes:

This instruction performs a 64-bit copy. The source is not required to have its upper 32 bits as sign-extensions. However, if the source value has a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

MOV #imm, Rn

MOV #imm, Rn



imm ← $SignExtend_8$(s);
op2 ← imm;
R_n ← Register($SignExtend_{32}$(op2));

Description:

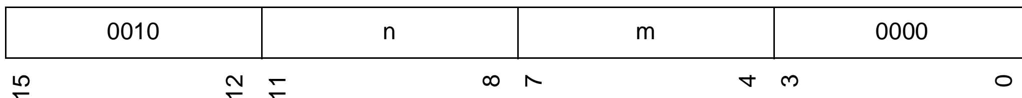
This instruction sign-extends the 8-bit immediate s and places the result in R_n.

Notes:

The '#imm' in the assembly syntax represents the immediate s after sign extension.

MOV.B Rm, @Rn

MOV.B Rm, @Rn



op1 ← $SignExtend_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
address ← $ZeroExtend_{64}(op2)$;
$WriteMemory_8(address, op1)$;

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.

Possible exceptions:

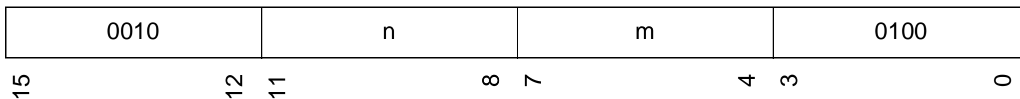
WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

MOV.B Rm, @-Rn

MOV.B Rm, @-Rn



op1 ← $SignExtend_{32}(R_m)$; op2 ← $SignExpect_{32}(R_n)$; address ← $ZeroExtend_{64}$(op2 - 1); $WriteMemory_8$(address, op1); op2 ← address; R_n ← Register($SignExtend_{32}$(op2));

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

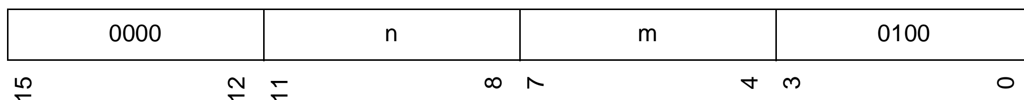
Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.B Rm, @(R0, Rn)

MOV.B Rm, @(R0, Rn)



r0 ← $\text{SignExpect}_{32}(R_0)$;
op1 ← $\text{SignExtend}_{32}(R_m)$;
op2 ← $\text{SignExpect}_{32}(R_n)$;
address ← ZeroExtend_{64}(r0 + op2);
WriteMemory_8(address, op1);

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

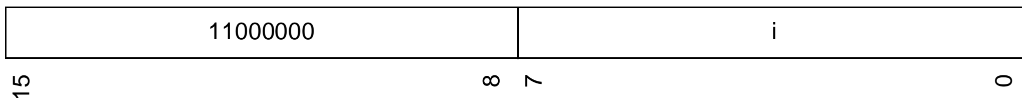
Notes:

The R_0 and R_n sources are required to have a 32-bit sign-extended representation.

If R_m is a different register to both R_0 and R_n, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m is the same register as either of R_0 or R_n, then the R_m source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.B R0, @(disp, GBR)



gbr ← SignExpect$_{32}$(GBR);
r0 ← SignExtend$_{32}$(R_0);
disp ← ZeroExtend$_8$(i);
address ← ZeroExtend$_{64}$(disp + gbr);
WriteMemory$_8$(address, r0);

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

Notes:

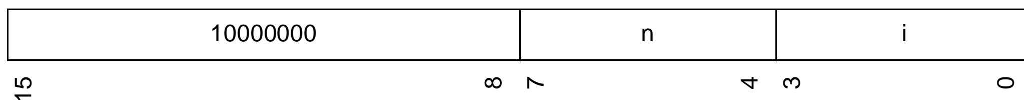
The GBR source is required to have a 32-bit sign-extended representation. The R_0 source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_0 are ignored.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The ‘disp’ in the assembly syntax represents the immediate i after zero extension.

MOV.B R0, @(disp, Rn)

MOV.B R0, @(disp, Rn)




```
r0 ← SignExtend32(R0);
disp ← ZeroExtend4(i);
op2 ← SignExpect32(Rn);
address ← ZeroExtend64(disp + op2);
WriteMemory8(address, r0);
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

Notes:

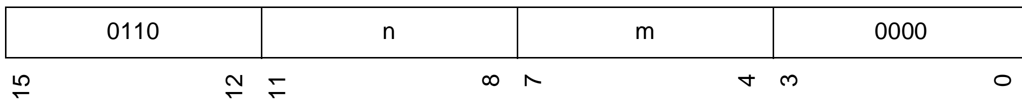
The R_n source is required to have a 32-bit sign-extended representation. If R_0 and R_n are different registers, then the R_0 source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_0 are ignored. However, if R_0 and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension.

MOV.B @Rm, Rn

MOV.B @Rm, Rn



op1 ← $\text{SignExpect}_{32}(R_m)$;
address ← ZeroExtend_{64}(op1);
op2 ← $\text{SignExtend}_8(\text{ReadMemory}_8(\text{address}))$;
R_n ← $\text{Register}(\text{SignExtend}_{32}(\text{op2}))$;

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.

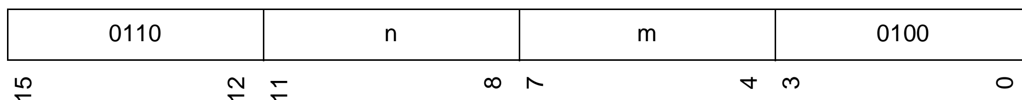
Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

MOV.B @Rm+, Rn




```
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExpect32(Rm);
address ← ZeroExtend64(op1);
op2 ← SignExtend8(ReadMemory8(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 1;
Rm ← Register(SignExtend32(op1));
Rn ← Register(SignExtend32(op2));
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

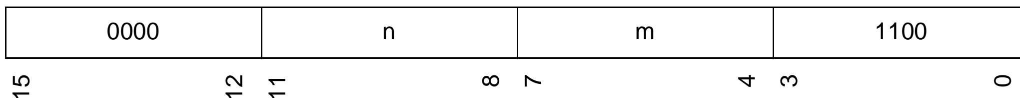
Notes:

The R_m source is required to have a 32-bit sign-extended representation.

If R_m and R_n refer to the same register (that is, m = n), the result placed in this register will be the sign-extended byte loaded from memory.

MOV.B @(R0, Rm), Rn

MOV.B @(R0, Rm), Rn



$r0 \leftarrow SignExpect_{32}(R_0);$
$op1 \leftarrow SignExpect_{32}(R_m);$
$address \leftarrow ZeroExtend_{64}(r0 + op1);$
$op2 \leftarrow SignExtend_8(ReadMemory_8(address));$
$R_n \leftarrow Register(SignExtend_{32}(op2));$

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.

Possible exceptions:

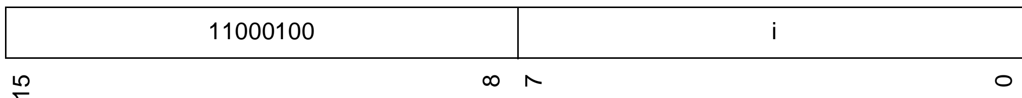
RADDERR, RTLBMISS, READPROT

Notes:

The R_0 and R_m sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.B @(disp, GBR), R0



gbr ← $SignExpect_{32}$(GBR);
disp ← $ZeroExtend_8$(i);
address ← $ZeroExtend_{64}$(disp + gbr);
r0 ← $SignExtend_8$($ReadMemory_8$(address));
R_0 ← Register($SignExtend_{32}$(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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

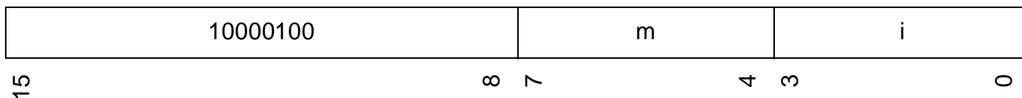
The GBR source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The ‘disp’ in the assembly syntax represents the immediate i after zero extension.

MOV.B @(disp, Rm), R0

MOV.B @(disp, Rm), R0



disp ← $ZeroExtend_4$(i);
op2 ← $SignExpect_{32}$(R_m);
address ← $ZeroExtend_{64}$(disp + op2);
r0 ← $SignExtend_8$($ReadMemory_8$(address));
R_0 ← Register($SignExtend_{32}$(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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

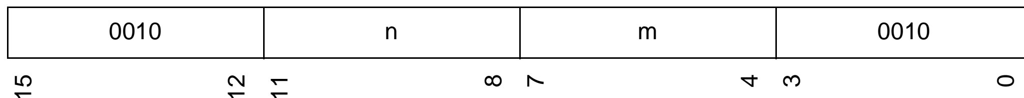
The R_m source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension.

MOV.L Rm, @Rn

MOV.L Rm, @Rn



op1 ← $SignExtend_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
address ← $ZeroExtend_{64}(op2)$;
$WriteMemory_{32}(address, op1)$;

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.

Possible exceptions:

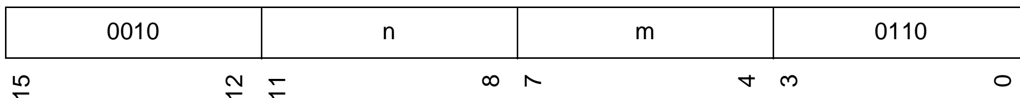
WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

MOV.L Rm, @-Rn

MOV.L Rm, @-Rn




```
op1 ← SignExtend32(Rm);
op2 ← SignExpect32(Rn);
address ← ZeroExtend64(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(SignExtend32(op2));
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

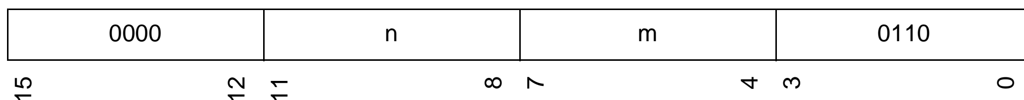
Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.L Rm, @(R0, Rn)

MOV.L Rm, @(R0, Rn)




```
r0 ← SignExpect32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExpect32(Rn);
address ← ZeroExtend64(r0 + op2);
WriteMemory32(address, op1);
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

Notes:

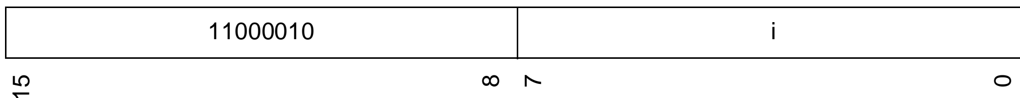
The R_0 and R_n sources are required to have a 32-bit sign-extended representation.

If R_m is a different register to both R_0 and R_n, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m is the same register as either of R_0 or R_n, then the R_m source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.L R0, @(disp, GBR)

MOV.L R0, @(disp, GBR)




```
gbr ← SignExpect32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i) << 2;
address ← ZeroExtend64(disp + gbr);
WriteMemory32(address, r0);
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

Notes:

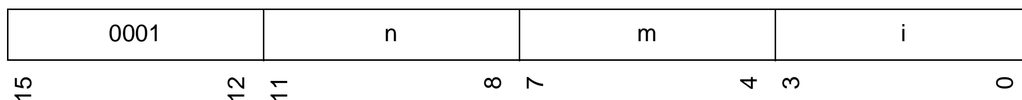
The GBR source is required to have a 32-bit sign-extended representation. The R_0 source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_0 are ignored.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The ‘disp’ in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L Rm, @(disp, Rn)

MOV.L Rm, @(disp, Rn)




```
op1 ← SignExtend32(Rm);
disp ← ZeroExtend4(i) << 2;
op3 ← SignExpect32(Rn);
address ← ZeroExtend64(disp + op3);
WriteMemory32(address, op1);
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

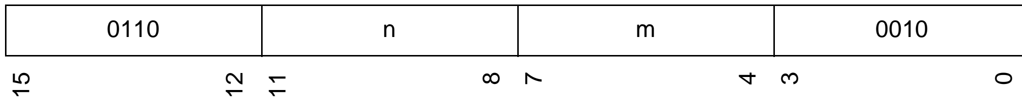
Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L @Rm, Rn



op1 ← $\text{SignExpect}_{32}(R_m)$;
address ← ZeroExtend_{64}(op1);
op2 ← $\text{SignExtend}_{32}(\text{ReadMemory}_{32}$(address));
R_n ← Register(SignExtend_{32}(op2));

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.

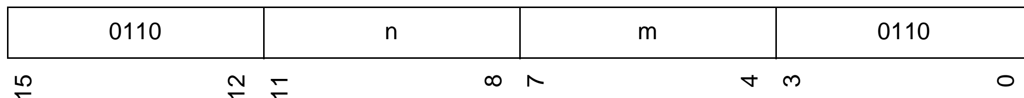
Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

MOV.L @Rm+, Rn




```
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExpect32(Rm);
address ← ZeroExtend64(op1);
op2 ← SignExtend32(ReadMemory32(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 4;
Rm ← Register(SignExtend32(op1));
Rn ← Register(SignExtend32(op2));
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

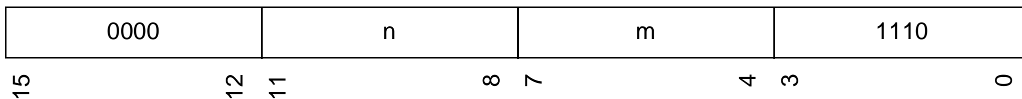
Notes:

The R_m source is required to have a 32-bit sign-extended representation.

If R_m and R_n refer to the same register (that is, m = n), the result placed in this register will be the long-word loaded from memory.

MOV.L @(R0, Rm), Rn

MOV.L @(R0, Rm), Rn



r0 ← $SignExpect_{32}(R_0)$;
op1 ← $SignExpect_{32}(R_m)$;
address ← $ZeroExtend_{64}(r0 + op1)$;
op2 ← $SignExtend_{32}(ReadMemory_{32}(address))$;
R_n ← $Register(SignExtend_{32}(op2))$;

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.

Possible exceptions:

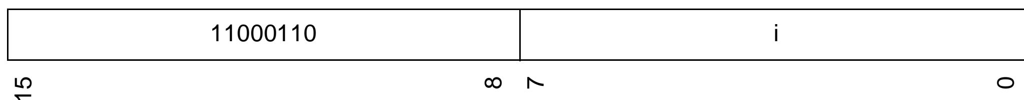
RADDERR, RTLBMISS, READPROT

Notes:

The R_0 and R_m sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.L @(disp, GBR), R0




```
gbr ← SignExpect32(GBR);
disp ← ZeroExtend8(i) << 2;
address ← ZeroExtend64(disp + gbr);
r0 ← SignExtend32(ReadMemory32(address));
R0 ← Register(SignExtend32(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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

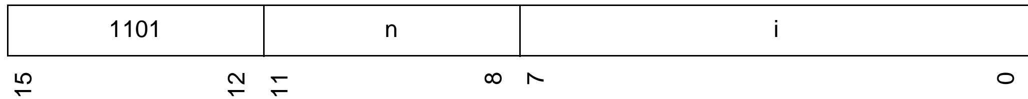
The GBR source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L @(disp, PC), Rn

MOV.L @(disp, PC), Rn




```
pc ← SignExpect32(PC);
disp ← ZeroExtend8(i) << 2;
IF (IsDelaySlot())
   THROW ILLSLOT;
address ← SignExtend32(disp + ((pc + 4) ∧ (~ 0x3)));
op2 ← SignExtend32(ReadMemory32(address));
Rn ← Register(SignExtend32(op2));
```

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. The effective address is then converted to a sign-extended 32-bit range. The long-word is loaded from this effective address and placed in R_n.

The address calculation ensures that the effective address is correctly aligned for a long-word access regardless of the PC alignment. Additionally, the calculation cannot generate an address outside the sign-extended 32-bit address space. The RADDERR exception is therefore not possible for this instruction.

Possible exceptions:

ILLSLOT, RTLBMISS, READPROT

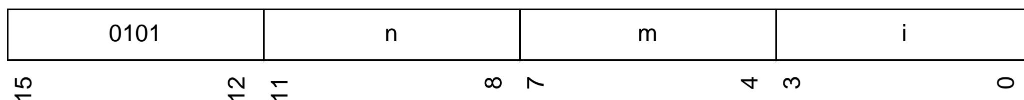
Notes:

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

MOV.L @(disp, Rm), Rn




```
disp ← ZeroExtend4(i) << 2;
op2 ← SignExpect32(Rm);
address ← ZeroExtend64(disp + op2);
op3 ← SignExtend32(ReadMemory32(address));
Rn ← Register(SignExtend32(op3));
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

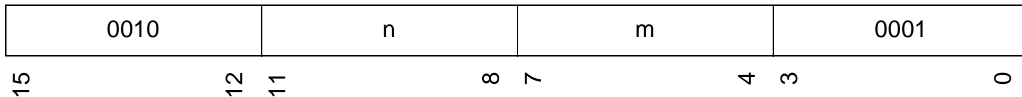
The R_m source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The ‘disp’ in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W Rm, @Rn

MOV.W Rm, @Rn



$op1 \leftarrow SignExtend_{32}(R_m);$
$op2 \leftarrow SignExpect_{32}(R_n);$
$address \leftarrow ZeroExtend_{64}(op2);$
$WriteMemory_{16}(address, op1);$

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.

Possible exceptions:

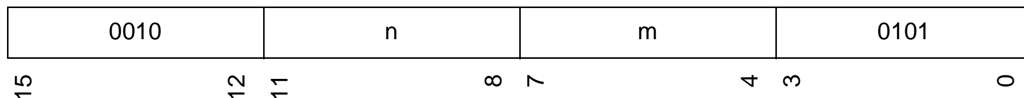
WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

MOV.W Rm, @-Rn

MOV.W Rm, @-Rn



op1 ← $SignExtend_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
address ← $ZeroExtend_{64}$(op2 - 2);
$WriteMemory_{16}$(address, op1);
op2 ← address;
R_n ← Register($SignExtend_{32}$(op2));

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.

Possible exceptions:

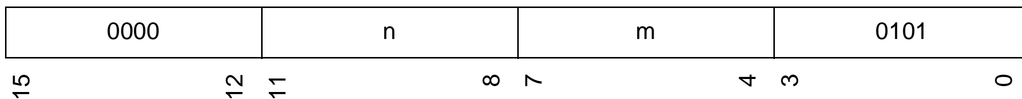
WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_m and R_n are different registers, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.W Rm, @(R0, Rn)




```
r0 ← SignExpect32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExpect32(Rn);
address ← ZeroExtend64(r0 + op2);
WriteMemory16(address, op1);
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

Notes:

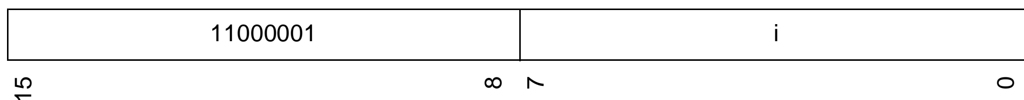
The R_0 and R_n sources are required to have a 32-bit sign-extended representation.

If R_m is a different register to both R_0 and R_n, then the R_m source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_m are ignored. However, if R_m is the same register as either of R_0 or R_n, then the R_m source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.W R0, @(disp, GBR)

MOV.W R0, @(disp, GBR)




```
gbr ← SignExpect32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i) << 1;
address ← ZeroExtend64(disp + gbr);
WriteMemory16(address, r0);
```

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_0.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

Notes:

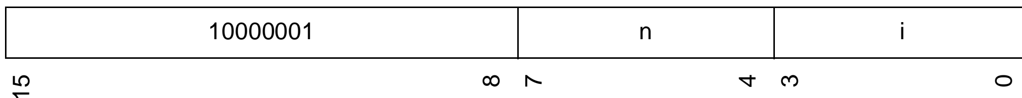
The GBR source is required to have a 32-bit sign-extended representation. The R_0 source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_0 are ignored.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W R0, @(disp, Rn)

MOV.W R0, @(disp, Rn)



r0 ← $SignExtend_{32}(R_0)$; disp ← $ZeroExtend_4$(i) << 1; op2 ← $SignExpect_{32}(R_n)$; address ← $ZeroExtend_{64}$(disp + op2); $WriteMemory_{16}$(address, r0);

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_0.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

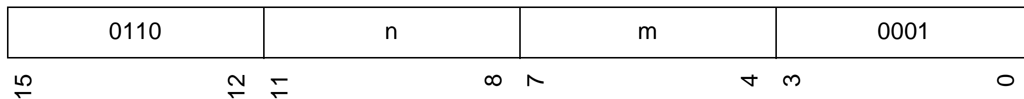
Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_0 and R_n are different registers, then the R_0 source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_0 are ignored. However, if R_0 and R_n are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W @Rm, Rn



op1 ← $SignExpect_{32}(R_m)$;
address ← $ZeroExtend_{64}$(op1);
op2 ← $SignExtend_{16}(ReadMemory_{16}$(address));
R_n ← Register($SignExtend_{32}$(op2));

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

MOV.W @Rm+, Rn

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 ← SignExpect32(Rm);
address ← ZeroExtend64(op1);
op2 ← SignExtend16(ReadMemory16(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 2;
Rm ← Register(SignExtend32(op1));
Rn ← Register(SignExtend32(op2));
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

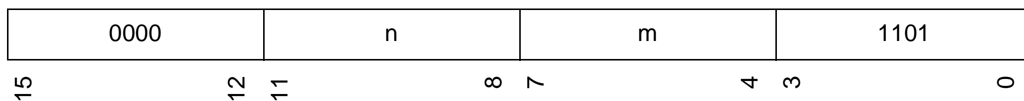
Notes:

The R_m source is required to have a 32-bit sign-extended representation.

If R_m and R_n refer to the same register (that is, m = n), the result placed in this register will be the sign-extended word loaded from memory.

MOV.W @(R0, Rm), Rn

MOV.W @(R0, Rm), Rn



r0 ← $\text{SignExpect}_{32}(R_0)$;
op1 ← $\text{SignExpect}_{32}(R_m)$;
address ← $\text{ZeroExtend}_{64}(\text{r0} + \text{op1})$;
op2 ← $\text{SignExtend}_{16}(\text{ReadMemory}_{16}(\text{address}))$;
R_n ← $\text{Register}(\text{SignExtend}_{32}(\text{op2}))$;

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

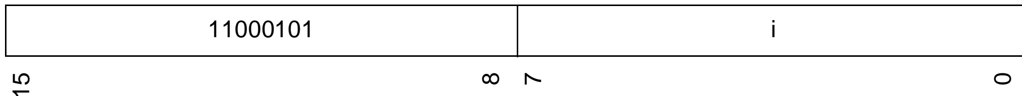
Notes:

The R_0 and R_m sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

MOV.W @(disp, GBR), R0

MOV.W @(disp, GBR), R0




```
gbr ← SignExpect32(GBR);
disp ← ZeroExtend8(i) << 1;
address ← ZeroExtend64(disp + gbr);
r0 ← SignExtend16(ReadMemory16(address));
R0 ← Register(SignExtend32(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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

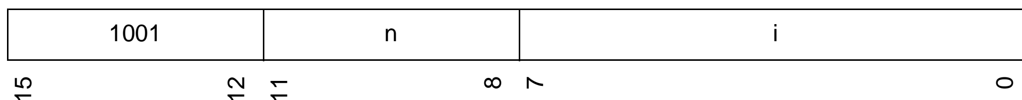
The GBR source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W @(disp, PC), Rn

MOV.W @(disp, PC), Rn




```
pc ← SignExpect32(PC);
disp ← ZeroExtend8(i) << 1;
IF (IsDelaySlot())
   THROW ILLSLOT;
address ← SignExtend32(disp + (pc + 4));
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(SignExtend32(op2));
```

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 effective address is then converted to a sign-extended 32-bit range. The word is loaded from this effective address, sign-extended and placed in R_n.

The address calculation ensures that the effective address is correctly aligned for a word access. Additionally, the calculation cannot generate an address outside the sign-extended 32-bit address space. The RADDERR exception is therefore not possible for this instruction.

Possible exceptions:

ILLSLOT, RTLBMISS, READPROT

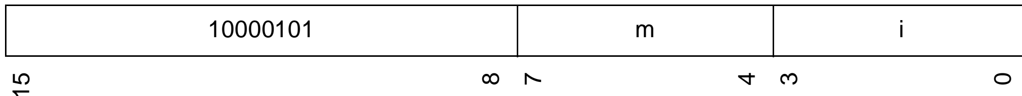
Notes:

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

MOV.W @(disp, Rm), R0



disp ← $ZeroExtend_4$(i) << 1;
op2 ← $SignExpect_{32}(R_m)$;
address ← $ZeroExtend_{64}$(disp + op2);
r0 ← $SignExtend_{16}(ReadMemory_{16}$(address));
R_0 ← Register($SignExtend_{32}$(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_0.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

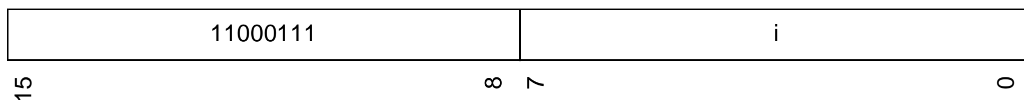
The R_m source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOVA @(disp, PC), R0

MOVA @(disp, PC), R0




```
pc ← SignExpect32(PC);
disp ← ZeroExtend8(i) << 2;
IF (IsDelaySlot())
   THROW ILLSLOT;
r0 ← disp + ((pc + 4) ∧ (~ 0x3));
R0 ← Register(SignExtend32(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 then converted to a sign-extended 32-bit range. The effective address is placed in R_0.

Possible exceptions:

ILLSLOT

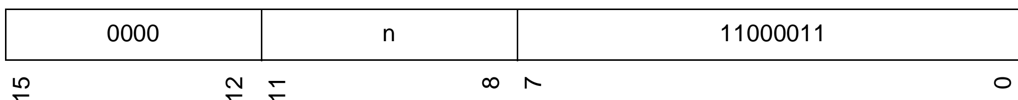
Notes:

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

MOVCA.L R0, @Rn




```
r0 ← SignExtend32(R0);
op1 ← SignExpect32(Rn);
IF (MalformedAddress(op1) OR ((op1 ∧ 0x3) ≠ 0))
    THROW WADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW WTLBMISS, op1;
IF (MMU() AND WriteProhibited(op1))
    THROW WRITEPROT, op1;
ALLOCO(op1);
address ← ZeroExtend64(op1);
WriteMemory32(op1, r0);
```

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.

Possible exceptions:

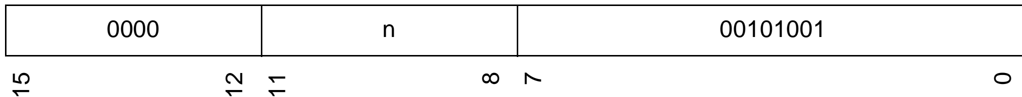
WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation. If R_0 and R_n are different registers, then the R_0 source value is not required to have a 32-bit sign-extended representation and the upper 32 bits of R_0 are ignored. However, if R_0 and R_m are the same register, then this register's source value is required to have a 32-bit sign-extended representation.

MOVT Rn

MOVT Rn



$t \leftarrow ZeroExpect_1(T);$
$op1 \leftarrow t;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$

Description:

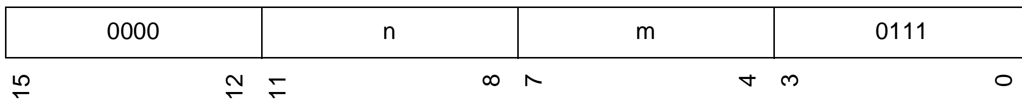
This instruction copies the T-bit to R_n.

Notes:

The T-bit source is required to have a 0 or 1 value.

MUL.L Rm, Rn

MUL.L Rm, Rn



op1 ← $\text{SignExpect}_{32}(R_m)$;
op2 ← $\text{SignExpect}_{32}(R_n)$;
macl ← op1 × op2;
MACL ← $\text{ZeroExtend}_{32}(\text{macl})$;

Description:

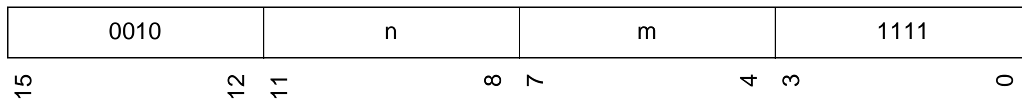
This instruction multiplies the 32-bit value in R_m by the 32-bit value in R_n, and places the least significant 32 bits of the result in MACL. The most significant 32 bits of the result are not provided, and MACH is not modified.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

MULS.W Rm, Rn

MULS.W Rm, Rn



op1 $\leftarrow$ SignExtend$_{16}$(SignExpect$_{32}$(R_m));
op2 $\leftarrow$ SignExtend$_{16}$(SignExpect$_{32}$(R_n));
macl $\leftarrow$ op1 $\times$ op2;
MACL $\leftarrow$ ZeroExtend$_{32}$(macl);

Description:

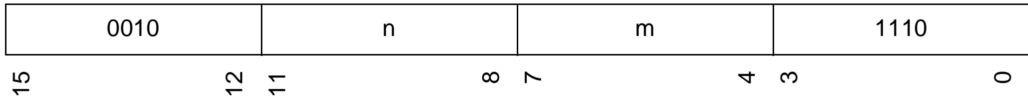
This instruction multiplies the signed lowest 16 bits of R_m by the signed lowest 16 bits of R_n, and places the full 32-bit result in MACL. MACH is not modified.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

MULU.W Rm, Rn

MULU.W Rm, Rn



op1 ← $ZeroExtend_{16}(SignExpect_{32}(R_m))$;
op2 ← $ZeroExtend_{16}(SignExpect_{32}(R_n))$;
macl ← op1 × op2;
MACL ← $ZeroExtend_{32}(macl)$;

Description:

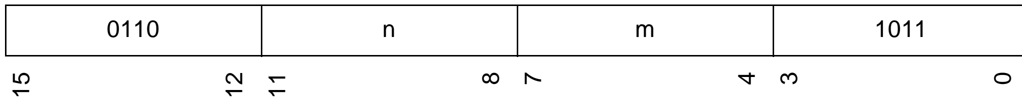
This instruction multiplies the unsigned lowest 16 bits of R_m by the unsigned lowest 16 bits of R_n, and places the full 32-bit result in MACL. MACH is not modified.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

NEG Rm, Rn

NEG Rm, Rn



0110	n	m	1011


```
op1 ← SignExpect32(Rm);
op2 ← - op1;
Rn ← Register(SignExtend32(op2));
```

Description:

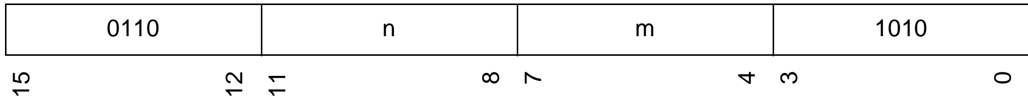
This instruction subtracts R_m from zero and places the result in R_n.

Notes:

The R_m source is required to have a 32-bit sign-extended representation.

NEGC Rm, Rn

NEGC Rm, Rn



t ← ZeroExpect_1(T);
op1 ← $\text{ZeroExtend}_{32}(\text{SignExpect}_{32}(R_m))$;
op2 ← (- op1) - t;
t ← $\text{op2}_{< 32\ \text{FOR}\ 1 >}$;
R_n ← $\text{Register}(\text{SignExtend}_{32}(\text{op2}))$;
T ← Bit(t);

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.

Notes:

The R_m source is required to have a 32-bit sign-extended representation. The T-bit source is required to have a 0 or 1 value.

NOP

NOP

0000000000001001

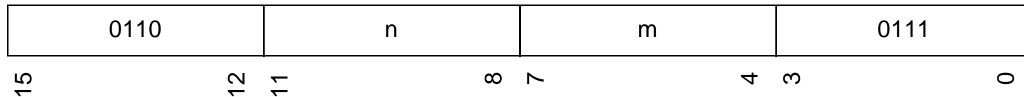
15 0

Description:

This instruction performs no operation.

NOT Rm, Rn

NOT Rm, Rn



$op1 \leftarrow ZeroExtend_{64}(R_m);$
$op2 \leftarrow \sim op1;$
$R_n \leftarrow Register(op2);$

Description:

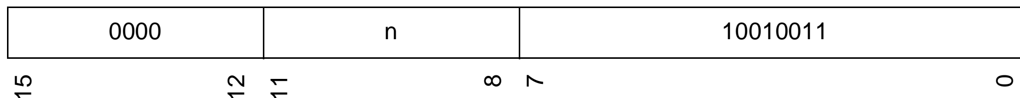
This instruction performs a bitwise NOT on R_m and places the result in R_n.

Notes:

This instruction performs a 64-bit bitwise NOT. The R_m source is not required to have its upper 32 bits as sign-extensions. However, if the source value has a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

OCBI @Rn

OCBI @Rn




```
op1 ← SignExpect32(Rn);
IF (MalformedAddress(op1))
   THROW WADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
   THROW WTLBMISS, op1;
IF (MMU() AND WriteProhibited(op1))
   THROW WRITEPROT, op1;
OCBI(op1);
```

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 any operand or unified cache.

There is no misalignment check on this instruction, and the specified effective address can be any byte address. The effective address specified in R_n is automatically aligned downwards to the nearest exact multiple of the cache block size.The effective address 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 checks for address error, translation miss and protection exception cases.

Possible exceptions:

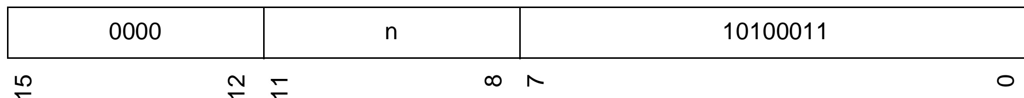
WADDERR, WTLBMISS, WRITEPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation. OCBI invalidates an implementation-dependent amount of data. For compatibility with other implementations, software must exercise care when using OCBI.

OCBP @Rn

OCBP @Rn




```
op1 ← SignExpect32(Rn);
IF (MalformedAddress(op1))
   THROW RADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
   THROW RTLBMISS, op1;
IF (MMU() AND (ReadProhibited(op1) AND WriteProhibited(op1)))
   THROW READPROT, op1;
OCBP(op1);
```

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 any operand or unified cache.

There is no misalignment check on this instruction, and the specified effective address can be any byte address. The effective address specified in R_n is automatically aligned downwards to the nearest exact multiple of the cache block size.The effective address 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.

Possible exceptions:

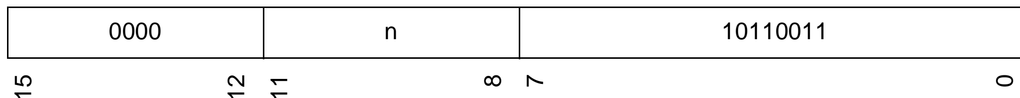
RADDERR, RTLBMISS, READPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

OCBWB @Rn

OCBWB @Rn



0000	n	10110011

15 12 11 8 7 0

```
op1 ← SignExpect32(Rn);
IF (MalformedAddress(op1))
   THROW RADDERR, op1;
IF (MMU() AND DataAccessMiss(op1))
   THROW RTLBMISS, op1;
IF (MMU() AND (ReadProhibited(op1) AND WriteProhibited(op1)))
   THROW READPROT, op1;
OCBWB(op1);
```

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 any operand or unified cache.

There is no misalignment check on this instruction, and the specified effective address can be any byte address. The effective address specified in R_n is automatically aligned downwards to the nearest exact multiple of the cache block size.The effective address 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.

Possible exceptions:

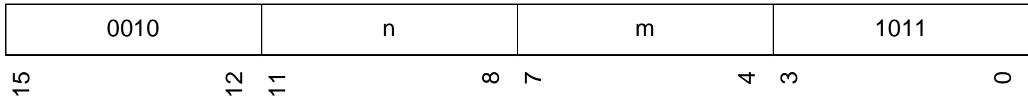
RADDERR, RTLBMISS, READPROT

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

OR Rm, Rn

OR Rm, Rn



0010	n	m	1011
15 12	11 8	7 4	3 0

$op1 \leftarrow ZeroExtend_{64}(R_m);$
$op2 \leftarrow ZeroExtend_{64}(R_n);$
$op2 \leftarrow op2 \vee op1;$
$R_n \leftarrow Register(op2);$

Description:

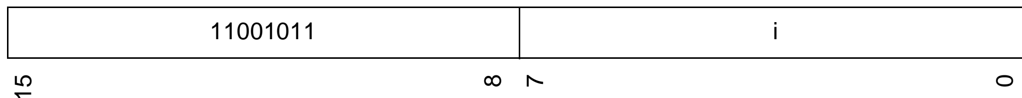
This instruction performs a bitwise OR of R_m with R_n and places the result in R_n.

Notes:

This instruction performs a 64-bit bitwise OR. The R_m and R_n sources are not required to have their upper 32 bits as sign-extensions. However, if both source values have a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

OR #imm, R0

OR #imm, R0



r0 ← ZeroExtend$_{64}$(R$_0$); imm ← ZeroExtend$_8$(i); r0 ← r0 ∨ imm; R$_0$ ← Register(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.

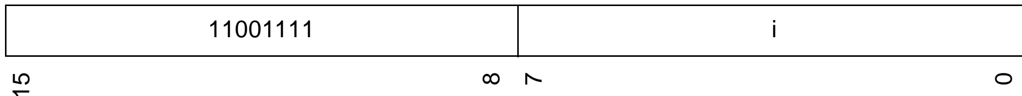
Notes:

This instruction performs a 64-bit bitwise OR. The R_0 source is not required to have its upper 32 bits as sign-extensions. However, if the R_0 source value has a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

OR.B #imm, @(R0, GBR)

OR.B #imm, @(R0, GBR)




```
r0 ← SignExpect32(R0);
gbr ← SignExpect32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend64(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ∨ imm;
WriteMemory8(address, value);
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT, WRITEPROT

Notes:

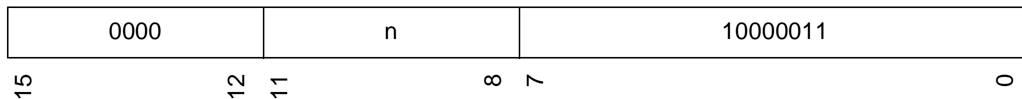
The R_0 and GBR sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The ‘#imm’ in the assembly syntax represents the immediate i after zero extension.

PREF @Rn

PREF @Rn




```
op1 ← SignExpect32(Rn);
IF (NOT MalformedAddress(op1))
   IF (NOT (MMU() AND DataAccessMiss(op1)))
      IF (NOT (MMU() AND ReadProhibited(op1)))
         PREFO(op1);
```

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.

There is no misalignment check on this instruction, and the specified effective address can be any byte address. The effective address specified in R_n is automatically aligned downwards to the nearest exact multiple of the cache block size.The effective address 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.

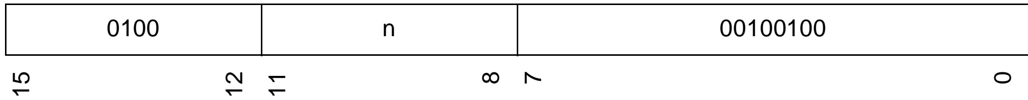
In exceptional cases, no exception is raised and the prefetch has no effect.

Notes:

The R_n source is required to have a 32-bit sign-extended representation.

ROTCL Rn

ROTCL Rn



$t \leftarrow \text{ZeroExpect}_1(T);$
$op1 \leftarrow \text{ZeroExtend}_{32}(R_n);$
$op1 \leftarrow (op1 << 1) \vee t;$
$t \leftarrow op1_{< 32 \text{ FOR } 1 >};$
$R_n \leftarrow \text{Register}(\text{SignExtend}_{32}(op1));$
$T \leftarrow \text{Bit}(t);$

Description:

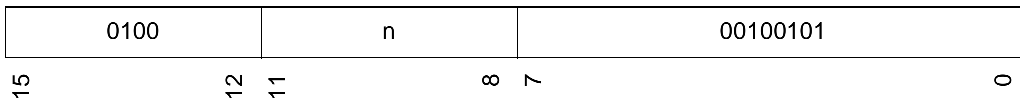
This instruction performs a one-bit left rotation of the bits held in R_n and the T-bit. The 32-bit value in R_n is shifted one bit to the left, the least significant bit is given the old value of the T-bit, and the bit that is shifted out is moved to the T-bit.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored. The T-bit source is required to have a 0 or 1 value.

ROTCR Rn

ROTCR Rn



$t \leftarrow ZeroExpect_1(T);$
$op1 \leftarrow ZeroExtend_{32}(R_n);$
$oldt \leftarrow t;$
$t \leftarrow op1_{< 0 \text{ FOR } 1 >};$
$op1 \leftarrow (op1 >> 1) \vee (oldt << 31);$
$R_n \leftarrow Register(SignExtend_{32}(op1));$
$T \leftarrow Bit(t);$

Description:

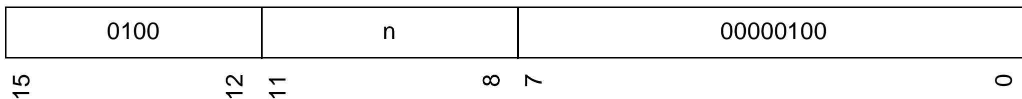
This instruction performs a one-bit right rotation of the bits held in R_n and the T-bit. The 32-bit value in R_n is shifted one bit to the right, the most significant bit is given the old value of the T-bit, and the bit that is shifted out is moved to the T-bit.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored. The T-bit source is required to have a 0 or 1 value.

ROTL Rn

ROTL Rn



$op1 \leftarrow ZeroExtend_{32}(R_n);$
$t \leftarrow op1_{< 31 \text{ FOR } 1 >};$
$op1 \leftarrow (op1 << 1) \vee t;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$
$T \leftarrow Bit(t);$

Description:

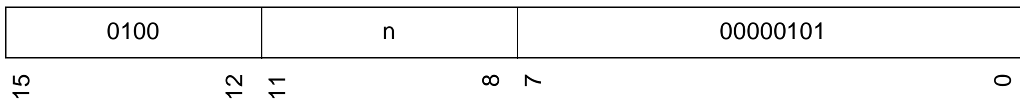
This instruction performs a one-bit left rotation of the bits held in R_n. The 32-bit value in R_n is shifted one bit to the left, and the least significant bit is given the value of the bit that is shifted out. The bit that is shifted out of the operand is also copied to the T-bit.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

ROTR Rn

ROTR Rn




```
op1 ← ZeroExtend32(Rn);
t ← op1< 0 FOR 1 >;
op1 ← (op1 >> 1) ∨ (t << 31);
Rn ← Register(SignExtend32(op1));
T ← Bit(t);
```

Description:

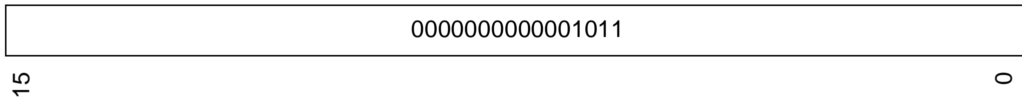
This instruction performs a one-bit right rotation of the bits held in R_n. The 32-bit value in R_n is shifted one bit to the right, and the most significant bit is given the value of the bit that is shifted out. The bit that is shifted out of the operand is also copied to the T-bit.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

RTS

RTS




```
pr ← SignExpect32(PR);
IF (IsDelaySlot())
   THROW ILLSLOT;
target ← pr;
IF ((target ∧ 0x3) = 0x3)
   THROW IADDERR, target;
delayedisa ← target ∧ 0x1;
delayedpc ← target ∧ (~ 0x1);
PC” ← Register(SignExtend32(delayedpc));
ISA” ← Bit(delayedisa);
```

Description:

This instruction is a delayed unconditional branch used for returning from a subroutine. The value in PR specifies the target address. If the last two bits of the target address are both set, an IADDERR exception is raised. Otherwise, the delay slot is executed in SHcompact. Bit zero of the target address gives the new value of the ISA mode for the next instruction. The least significant bit of the target address is cleared, and this value is copied to the PC.

Possible exceptions:

ILLSLOT, IADDERR

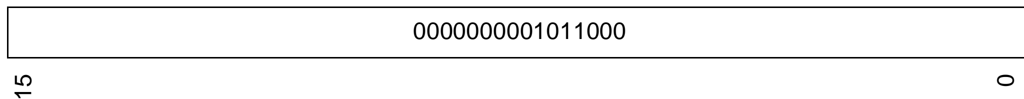
Notes:

The PR source is required to have a 32-bit sign-extended representation.

Since this is a delayed branch instruction, the delay slot is executed before branching and before ISA is updated. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

SETS

SETS

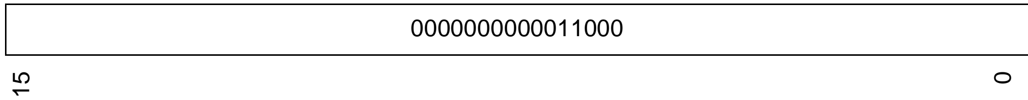



```
s ← 1;
S ← Bit(s);
```

Description:

This instruction sets the S-bit to 1.

SETT



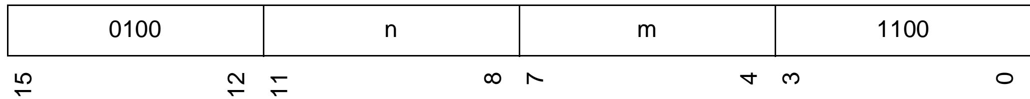

```
t ← 1;
T ← Bit(t);
```

Description:

This instruction sets the T-bit to 1.

SHAD Rm, Rn

SHAD Rm, Rn




```
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(SignExtend32(op2));
```

Description:

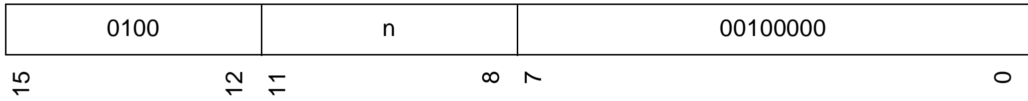
This instruction performs an arithmetic shift of R_n, with the dynamic shift direction and shift amount indicated by R_m, and places the result in R_n. If R_m is zero, no shift is performed. If R_m is greater than zero, this is a left shift and the shift amount is given by the least significant 5 bits of R_m. If R_m is less than zero, this is an arithmetic 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 an arithmetic right shift by 32, the result is filled with copies of the sign-bit of the original R_n.

Notes:

The R_m and R_n source values are not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m and R_n are ignored.

SHAL Rn

SHAL Rn



$op1 \leftarrow SignExtend_{32}(R_n);$
$t \leftarrow op1_{< 31 \text{ FOR } 1 >};$
$op1 \leftarrow op1 << 1;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$
$T \leftarrow Bit(t);$

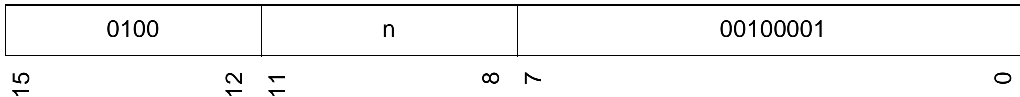
Description:

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.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHAR Rn




```
op1 ← SignExtend32(Rn);
t ← op1< 0 FOR 1 >;
op1 ← op1 >> 1;
Rn ← Register(SignExtend32(op1));
T ← Bit(t);
```

Description:

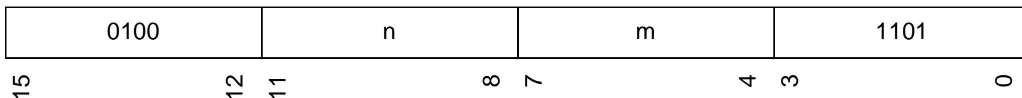
Arithmetically shifts R_n to the right by one bit and places the result in R_n. The bit that is shifted out of the operand is moved to T-bit.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLD Rm, Rn

SHLD Rm, Rn




```
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(SignExtend32(op2));
```

Description:

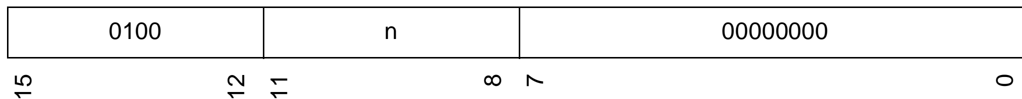
This instruction performs a logical shift of R_n, with the dynamic shift direction and shift amount indicated by R_m, and places the result in R_n. If R_m is zero, no shift is performed. If R_m is greater than zero, this is a left shift and the shift amount is given by the least significant 5 bits of 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.

Notes:

The R_m and R_n source values are not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m and R_n are ignored.

SHLL Rn

SHLL Rn



$op1 \leftarrow ZeroExtend_{32}(R_n);$
$t \leftarrow op1_{<31\ FOR\ 1>};$
$op1 \leftarrow op1 << 1;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$
$T \leftarrow Bit(t);$

Description:

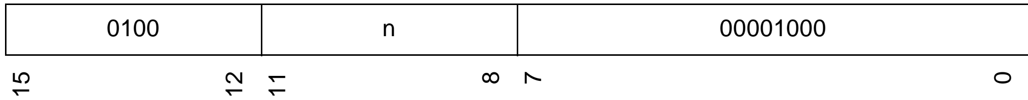
This instruction performs a logical left shift of R_n by 1 bit and places the result in R_n. The bit that is shifted out is moved to the T-bit.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLL2 Rn

SHLL2 Rn



0100	n	00001000

Bits: 15–12, 11–8, 7–0

```
op1 ← ZeroExtend32(Rn);
op1 ← op1 << 2;
Rn ← Register(SignExtend32(op1));
```

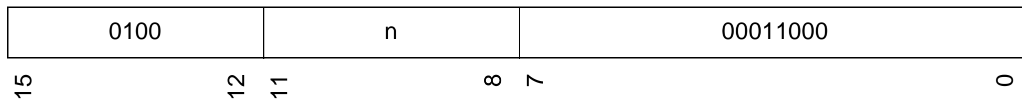
Description:

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.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLL8 Rn



op1 ← $ZeroExtend_{32}(R_n)$;
op1 ← op1 << 8;
R_n ← $Register(SignExtend_{32}(op1))$;

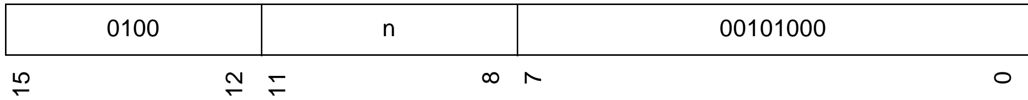
Description:

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.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLL16 Rn



op1 ← $ZeroExtend_{32}(R_n)$;
op1 ← op1 << 16;
R_n ← $Register(SignExtend_{32}(op1))$;

Description:

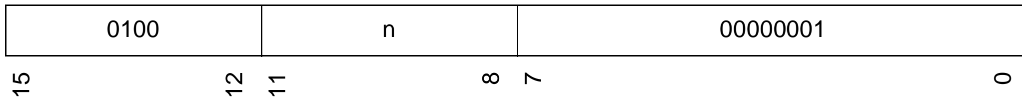
This instruction performs a logical left shift of R_n by 16 bits and places the result in R_n. The bits that are shifted out are discarded.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLR Rn

SHLR Rn




```
op1 ← ZeroExtend32(Rn);
t ← op1< 0 FOR 1 >;
op1 ← op1 >> 1;
Rn ← Register(SignExtend32(op1));
T ← Bit(t);
```

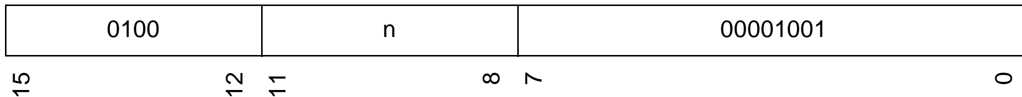
Description:

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.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLR2 Rn




```
op1 ← ZeroExtend32(Rn);
op1 ← op1 >> 2;
Rn ← Register(SignExtend32(op1));
```

Description:

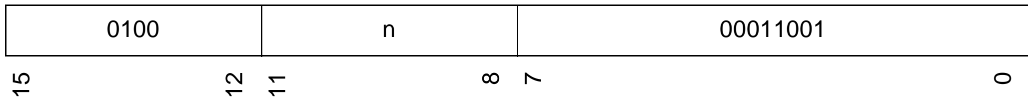
This instruction performs a logical right shift of R_n by 2 bits and places the result in R_n. The bits that are shifted out are discarded.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLR8 Rn

SHLR8 Rn



op1 ← $ZeroExtend_{32}(R_n)$;
op1 ← op1 >> 8;
R_n ← $Register(SignExtend_{32}(op1))$;

Description:

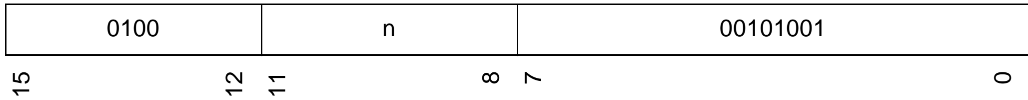
This instruction performs a logical right shift of R_n by 8 bits and places the result in R_n. The bits that are shifted out are discarded.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

SHLR16 Rn

SHLR16 Rn




```
op1 ← ZeroExtend32(Rn);
op1 ← op1 >> 16;
Rn ← Register(SignExtend32(op1));
```

Description:

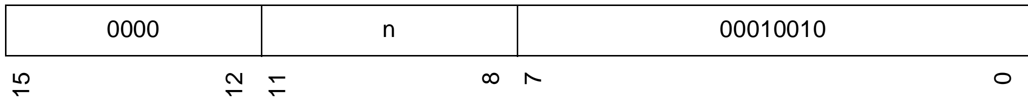
This instruction performs a logical right shift of R_n by 16 bits and places the result in R_n. The bits that are shifted out are discarded.

Notes:

The R_n source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_n are ignored.

STC GBR, Rn

STC GBR, Rn



$gbr \leftarrow SignExtend_{32}(GBR);$
$op1 \leftarrow gbr;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$

Description:

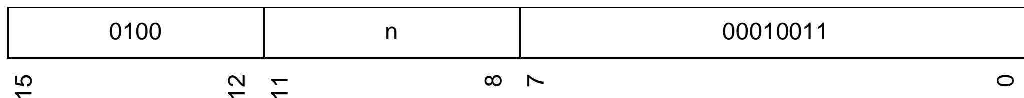
This instruction copies GBR to R_n.

Notes:

The GBR source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of GBR are ignored.

STC.L GBR, @-Rn

STC.L GBR, @-Rn



gbr ← $SignExtend_{32}$(GBR);
op1 ← $SignExpect_{32}(R_n)$;
address ← $ZeroExtend_{64}$(op1 - 4);
$WriteMemory_{32}$(address, gbr);
op1 ← address;
R_n ← Register($SignExtend_{32}$(op1));

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

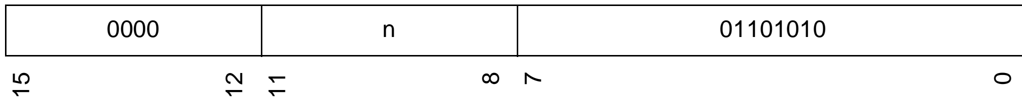
Notes:

The R_n source is required to have a 32-bit sign-extended representation. The GBR source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of GBR are ignored.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

STS FPSCR, Rn

STS FPSCR, Rn




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
pr ← ZeroExtend1(SR.PR);
sz ← ZeroExtend1(SR.SZ);
fr ← ZeroExtend1(SR.FR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← PackFPSCR(fps, pr, sz, fr);
Rn ← Register(SignExtend32(op1));
```

Description:

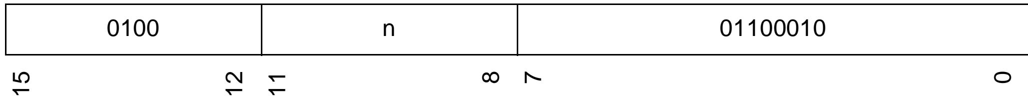
This floating-point instruction copies FPSCR to R_n.

Possible exceptions:

SLOTFPUDIS, FPUDIS

STS.L FPSCR, @-Rn

STS.L FPSCR, @-Rn




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
pr ← ZeroExtend1(SR.PR);
sz ← ZeroExtend1(SR.SZ);
fr ← ZeroExtend1(SR.FR);
op1 ← SignExpect32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
   THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
value ← PackFPSCR(fps, pr, sz, fr);
address ← ZeroExtend64(op1 - 4);
WriteMemory32(address, value);
op1 ← address;
Rn ← Register(SignExtend32(op1));
```

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

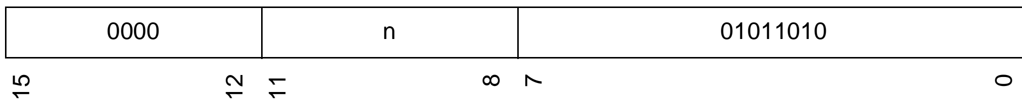
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

STS FPUL, Rn

STS FPUL, Rn



0000	n	01011010


```
sr ← ZeroExtend64(SR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← fpul;
Rn ← Register(SignExtend32(op1));
```

Description:

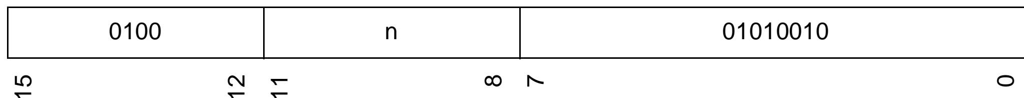
This floating-point instruction copies FPUL to R_n.

Possible exceptions:

SLOTFPUDIS, FPUDIS

STS.L FPUL, @-Rn

STS.L FPUL, @-Rn




```
sr ← ZeroExtend64(SR);
fpul ← SignExtend32(FPUL);
op1 ← SignExpect32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend64(op1 - 4);
WriteMemory32(address, fpul);
op1 ← address;
Rn ← Register(SignExtend32(op1));
```

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.

Possible exceptions:

SLOTFPUDIS, FPUDIS, WADDERR, WTLBMISS, WRITEPROT

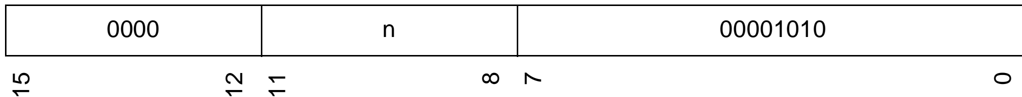
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

STS MACH, Rn

STS MACH, Rn



0000	n	00001010

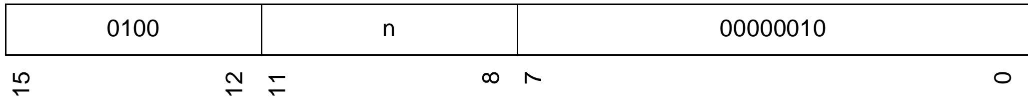
mach ← $SignExtend_{32}$(MACH);
op1 ← mach;
R_n ← Register($SignExtend_{32}$(op1));

Description:

This instruction copies MACH to R_n.

STS.L MACH, @-Rn

STS.L MACH, @-Rn




```
mach ← SignExtend32(MACH);
op1 ← SignExpect32(Rn);
address ← ZeroExtend64(op1 - 4);
WriteMemory32(address, mach);
op1 ← address;
Rn ← Register(SignExtend32(op1));
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

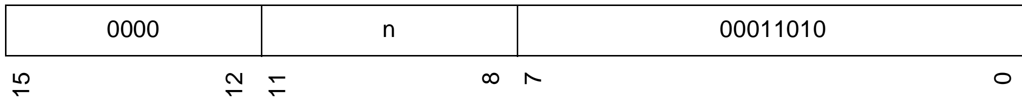
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

STS MACL, Rn

STS MACL, Rn



0000	n	00011010

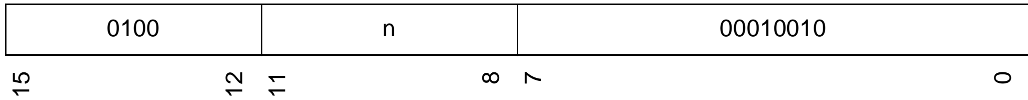
15 12 11 8 7 0

$macl \leftarrow SignExtend_{32}(MACL);$
$op1 \leftarrow macl;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$

Description:

This instruction copies MACL to R_n.

STS.L MACL, @-Rn




```
macl ← SignExtend32(MACL);
op1 ← SignExpect32(Rn);
address ← ZeroExtend64(op1 - 4);
WriteMemory32(address, macl);
op1 ← address;
Rn ← Register(SignExtend32(op1));
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

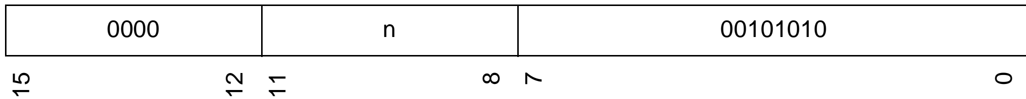
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

STS PR, Rn

STS PR, Rn



$pr \leftarrow SignExtend_{32}(PR');$
$op1 \leftarrow pr;$
$R_n \leftarrow Register(SignExtend_{32}(op1));$

Description:

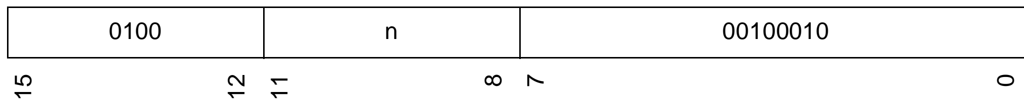
This instruction copies PR to R_n.

Notes:

The PR source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of PR are ignored.

STS.L PR, @-Rn

STS.L PR, @-Rn




```
pr ← SignExtend32(PR');
op1 ← SignExpect32(Rn);
address ← ZeroExtend64(op1 - 4);
WriteMemory32(address, pr);
op1 ← address;
Rn ← Register(SignExtend32(op1));
```

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.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

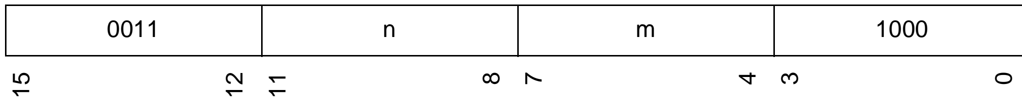
Notes:

The R_n source is required to have a 32-bit sign-extended representation. The PR source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of PR are ignored.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

SUB Rm, Rn

SUB Rm, Rn



op1 ← $SignExpect_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
op2 ← op2 - op1;
R_n ← $Register(SignExtend_{32}(op2))$;

Description:

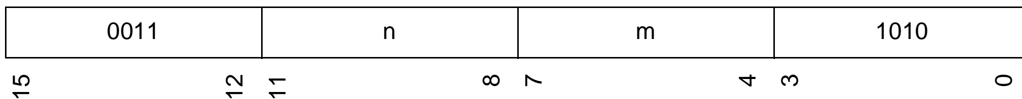
This instruction subtracts R_m from R_n and places the result in R_n.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

SUBC Rm, Rn

SUBC Rm, Rn



$t \leftarrow ZeroExpect_1(T);$
$op1 \leftarrow ZeroExtend_{32}(SignExpect_{32}(R_m));$
$op2 \leftarrow ZeroExtend_{32}(SignExpect_{32}(R_n));$
$op2 \leftarrow (op2 - op1) - t;$
$t \leftarrow op2_{< 32 FOR 1 >};$
$R_n \leftarrow Register(SignExtend_{32}(op2));$
$T \leftarrow Bit(t);$

Description:

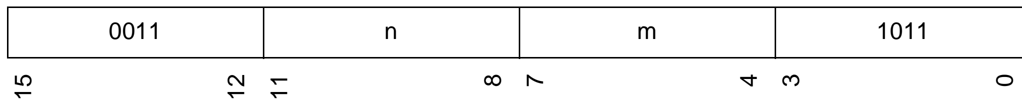
This instruction subtracts R_m and the T-bit from R_n and places the result in R_n. The borrow from the subtraction is placed in the T-bit.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation. The T-bit source is required to have a 0 or 1 value.

SUBV Rm, Rn

SUBV Rm, Rn



op1 ← $SignExpect_{32}(R_m)$;
op2 ← $SignExpect_{32}(R_n)$;
op2 ← op2 - op1;
t ← INT ((op2 < (- 2^{31})) OR (op2 ≥ 2^{31}));
R_n ← Register($SignExtend_{32}$(op2));
T ← Bit(t);

Description:

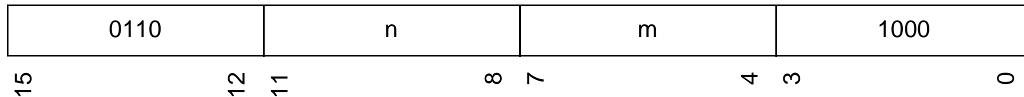
This instruction subtracts R_m from R_n and places the result in R_n. The T-bit is set to 1 if the subtraction result is outside the 32-bit signed range, otherwise the T-bit is set to 0.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

SWAP.B Rm, Rn

SWAP.B Rm, Rn



$op1 \leftarrow \text{ZeroExtend}_{32}(R_m);$
$op2 \leftarrow ((op1_{<\,16\ \text{FOR}\ 16\,>} << 16) \vee (op1_{<\,0\ \text{FOR}\ 8\,>} << 8)) \vee op1_{<\,8\ \text{FOR}\ 8\,>};$
$R_n \leftarrow \text{Register}(\text{SignExtend}_{32}(op2));$

Description:

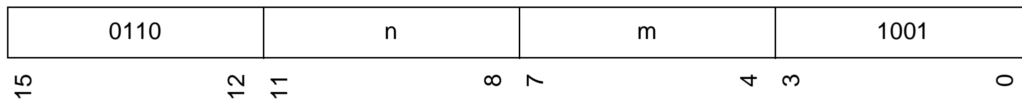
This instruction swaps the values of the lower 2 bytes in R_m and places the result in R_n. Bits [0,7] take the value of bits [8,15]. Bits [8,15] take the value of bits [0,7]. Bits [16,31] are unchanged.

Notes:

The R_m source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

SWAP.W Rm, Rn

SWAP.W Rm, Rn



$op1 \leftarrow ZeroExtend_{32}(R_m);$
$op2 \leftarrow (op1_{<0 \text{ FOR } 16>} << 16) \vee op1_{<16 \text{ FOR } 16>};$
$R_n \leftarrow Register(SignExtend_{32}(op2));$

Description:

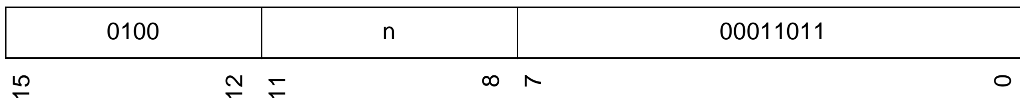
This instruction swaps the values of the 2 words in R_m and places the result in R_n. Bits [0,15] take the value of bits [16,31]. Bits [16,31] take the value of bits [0,15].

Notes:

The R_m source value is not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m are ignored.

TAS.B @Rn

TAS.B @Rn




```
op1 ← SignExpect32(Rn);
address ← ZeroExtend64(op1);
value ← ZeroExtend8(ReadMemory8(address));
t ← INT (value = 0);
value ← value ∨ (1 << 7);
WriteMemory8(address, value);
T ← Bit(t);
```

Description:

This instruction performs a test-and-set operation on the byte data at the effective address specified in R_n. The 8 bits of data at the effective address are read. 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 to the same effective address.

This test-and-set is atomic from the CPU perspective. This instruction cannot be interrupted during its operation. However, atomicity is not provided with respect to accesses from other memory users. It is possible that another memory access from another memory user could occur between the two TAS.B accesses.

There is no special treatment for TAS.B regarding the cache, and it behaves in the same way as a load followed by a store. Depending on the cache behavior, it is possible for the TAS.B accesses to be completed in the cache with no external memory activity.

The SHmedia SWAP.Q instruction (see *Volume 1, Chapter 6: SHmedia memory instructions*) provides an atomic read-modify-write on external memory, and should be used for synchronization with other memory users.

Possible exceptions:

RADDERR, RTLBMISS, READPROT, WRITEPROT

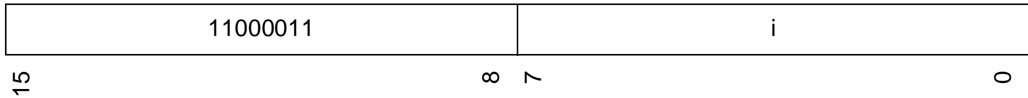
Notes:

The R_n source is required to have a 32-bit sign-extended representation.

The atomicity properties of this instruction are reduced relative to SH-4. The SH-4 TAS.B instruction guarantees atomicity with respect to all memory accesses from all memory users. The SHcompact semantics continue to support the use of TAS.B to synchronize between software threads executing on the same CPU. It cannot be used to synchronize with other memory users or hardware devices.

TRAPA #imm

TRAPA #imm




```
imm ← ZeroExtend8(i);
IF (IsDelaySlot())
    THROW ILLSLOT;
THROW TRAP, 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.

Possible exceptions:

ILLSLOT, TRAP

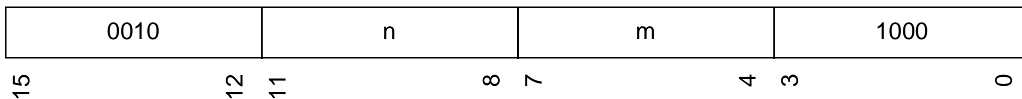
Notes:

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

TST Rm, Rn



$op1 \leftarrow SignExpect_{32}(R_m);$
$op2 \leftarrow SignExpect_{32}(R_n);$
$t \leftarrow INT\ ((op1 \wedge op2) = 0);$
$T \leftarrow Bit(t);$

Description:

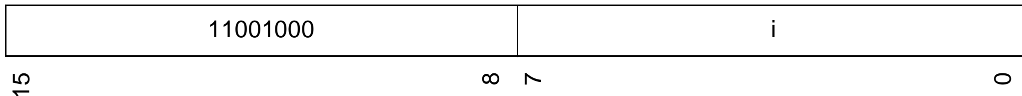
This instruction performs a bitwise AND of R_m with R_n. If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

Notes:

The R_m and R_n sources are required to have a 32-bit sign-extended representation.

TST #imm, R0

TST #imm, R0



$r0 \leftarrow \text{SignExpect}_{32}(R_0);$ $imm \leftarrow \text{ZeroExtend}_8(i);$ $t \leftarrow \text{INT}((r0 \wedge imm) = 0);$ $T \leftarrow \text{Bit}(t);$

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.

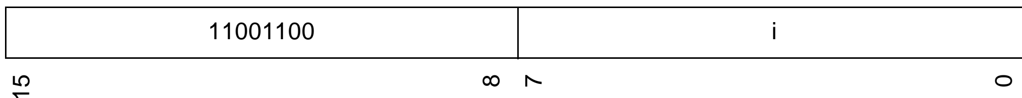
Notes:

The R_0 source is required to have a 32-bit sign-extended representation.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

TST.B #imm, @(R0, GBR)

TST.B #imm, @(R0, GBR)




```
r0 ← SignExpect32(R0);
gbr ← SignExpect32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend64(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
t ← ((value ∧ imm) = 0);
T ← Bit(t);
```

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

Notes:

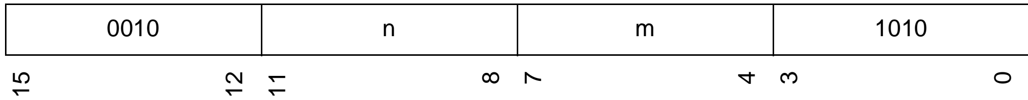
The R_0 and GBR sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

XOR Rm, Rn

XOR Rm, Rn



$op1 \leftarrow ZeroExtend_{64}(R_m);$
$op2 \leftarrow ZeroExtend_{64}(R_n);$
$op2 \leftarrow op2 \oplus op1;$
$R_n \leftarrow Register(op2);$

Description:

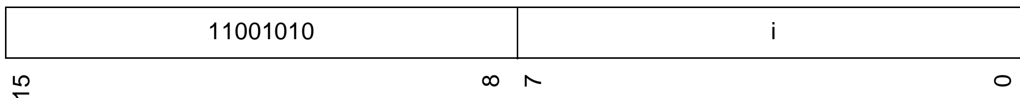
This instruction performs a bitwise XOR of R_m with R_n and places the result in R_n.

Notes:

This instruction performs a 64-bit bitwise XOR. The R_m and R_n sources are not required to have their upper 32 bits as sign-extensions. However, if both source values have a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

XOR #imm, R0

XOR #imm, R0



$r0 \leftarrow \text{ZeroExtend}_{64}(R_0);$
$imm \leftarrow \text{ZeroExtend}_{8}(i);$
$r0 \leftarrow r0 \oplus imm;$
$R_0 \leftarrow \text{Register}(r0);$

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.

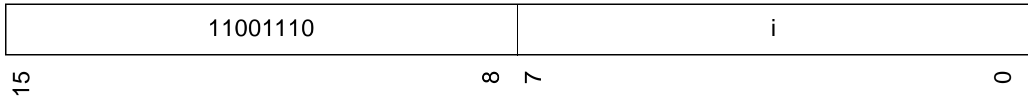
Notes:

This instruction performs a 64-bit bitwise XOR. The R_0 source is not required to have its upper 32 bits as sign-extensions. However, if the R_0 source value has a 32-bit sign-extended representation, then the result will also have a 32-bit sign-extended representation.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

XOR.B #imm, @(R0, GBR)

XOR.B #imm, @(R0, GBR)



r0 ← $\text{SignExpect}_{32}(R_0)$;
gbr ← $\text{SignExpect}_{32}(\text{GBR})$;
imm ← $\text{ZeroExtend}_{8}(i)$;
address ← $\text{ZeroExtend}_{64}(r0 + gbr)$;
value ← $\text{ZeroExtend}_{8}(\text{ReadMemory}_{8}(\text{address}))$;
value ← value ⊕ imm;
$\text{WriteMemory}_{8}(\text{address, value})$;

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.

Possible exceptions:

RADDERR, RTLBMISS, READPROT, WRITEPROT

Notes:

The R_0 and GBR sources are required to have a 32-bit sign-extended representation.

The effective address calculation is performed at 64-bit precision, and can generate an address outside the sign-extended 32-bit address space.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

XTRCT Rm, Rn

XTRCT Rm, Rn

0010	n	m	1101
15 … 12	11 … 8	7 … 4	3 … 0

$op1 \leftarrow ZeroExtend_{32}(R_m);$
$op2 \leftarrow ZeroExtend_{32}(R_n);$
$op2 \leftarrow op2_{<16\ FOR\ 16>} \vee (op1_{<0\ FOR\ 16>} << 16);$
$R_n \leftarrow Register(SignExtend_{32}(op2));$

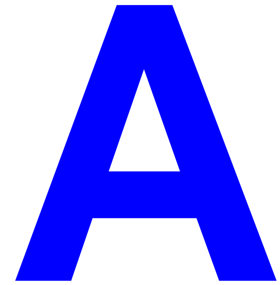
Description:

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.

Notes:

The R_m and R_n source values are not required to have a 32-bit sign-extended representation. The upper 32 bits of R_m and R_n are ignored.

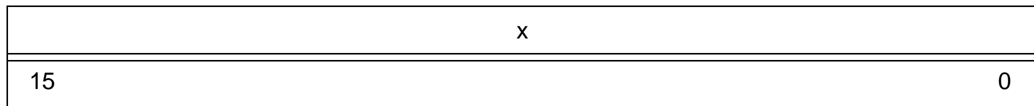
SHcompact instruction encoding



A.1 Formats

SHcompact uses the following instruction formats to encode its 16-bit instructions.

A.2 0 format



Instructions in this format do not have explicit operands. The opcode indirectly refers to a special action to be taken (possibly on an implicit resource).

Format name	Example mnemonic(s)	Operands
0	CLRT NOP RTE DIV0U SLEEP	

Table 14: 0 format summary

A.3 n format

x	n	x
15 12	11 8	7 0

Instructions in this format operate on operands constructed from:

- Direct register
- Indirect register
- Special register

Format name	Example mnemonic(s)	Operands
n	CMP/PZ	Rn
	SHLL	Rn
	STC	GBR, Rn
	STS	MACH, Rn
	JMP	@Rn
	STC.L	GBR, @-Rn
	STS.L	MACH, @-Rn
	BSRF	Rn

Table 15: n format summary

A.4 m format

x	m	x
15 12	11 8	7 0

Instructions in this format operate on operands constructed from:

- Direct register
- Indirect register
- Special register

Format Name	Example Mnemonic(s)	Operands
m	STC	Rm, GBR
	LDS	Rm, MACH
	LDC.L	@Rm+, GBR
	LDC.L	@Rm+, MACH

Table 16: m format summary

A.5 nm format

x	n	m	x
15 12	11 8	7 4	3 0

Instructions in this format operate on operands constructed from:

- Direct register
- Indirect register
- Special register

Format name	Example mnemonic(s)	Operands
nm	ADD	Rm, Rn
	XOR	Rm, Rn
	MOV.B	Rm, @Rn
	MAC.L	@Rm+, @Rn+
	MOV.L	@Rm+, Rn
	MOV.W	Rm, @-Rn
	MOV.W	Rm, @(R0, Rn)
	MOV.L	@(R0, Rm), Rn

Table 17: nm format summary

A.6 md format

x		m		d	
15	8	7	4	3	0

Instructions in this format operate on operands constructed from:

- Direct register
- Indirect register with displacement

Format name	Example mnemonic(s)	Operands
md	MOV.B	@(disp, Rm), R0
	MOV.W	@(disp, Rm), R0

Table 18: md format summary

A.7 nd4 format

x	n	d
15 8	7 4	3 0

Instructions in this format operate on operands constructed from:

- Direct register
- Indirect register with displacement

Format name	Example mnemonic(s)	Operands
nd4	MOV.B	R0, @(disp, Rn)
	MOV.W	R0, @(disp, Rn)

Table 19: nd4 format summary

A.8 nmd format

x	n	m	d
15 12	11 8	7 4	3 0

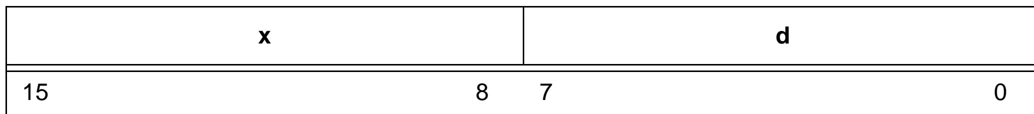
Instructions in this format operate on operands constructed from:

- Direct register
- Indirect register with displacement

Format name	Example mnemonic(s)	Operands
nmd	MOV.L	Rm, @(disp, Rn)
	MOV.L	@(disp, Rm), Rn

Table 20: nmd format summary

A.9 d format



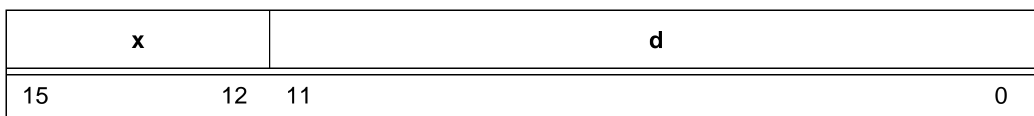
Instructions in this format operate on operands constructed from:

- Direct register
- Indirect Global Base Register (GBR) with displacement
- Program Counter (PC) with displacement

Format Name	Example Mnemonic(s)	Operands
d	MOV.B	R0, @(disp, GBR)
	MOV.L	@(disp, GBR), R0
	MOVA	@(disp, PC), R0
	BT	disp

Table 21: d format summary

A.10 d12 format



Instructions in this format operate on operands constructed from:

- Program Counter (PC) with displacement

Format name	Example mnemonic(s)	Operands
d12	BRA	disp
	BSR	disp

Table 22: d12 format summary

A.11 nd8 format

x		n		d	
15	12	11	8	7	0

Instructions in this format operate on operands constructed from:

- Direct register
- Program Counter (PC) with displacement

Format name	Example mnemonic(s)	Operands
nd8	MOV.W	@(disp, PC), Rn
	MOV.L	@(disp, PC), Rn

Table 23: nd8 format summary

A.12 i format

x		i	
15	8	7	0

Instructions in this format operate on operands constructed from:

- Immediate field
- Direct register
- Indirect Global Base Register (GBR) with index

Format name	Example mnemonic(s)	Operands
i	AND.B	#imm, @(R0, GBR)
	TST	#imm, R0
	CMP/EQ	#imm, R0
	TRAPA	#imm

Table 24: i format summary

A.13 ni format

x		n		i	
15	12	11	8	7	0

Instructions in this format operate on operands constructed from:

- Immediate field
- Direct register

Format Name	Example Mnemonic(s)	Operands
ni	ADD	#imm, Rn
	MOV	#imm, Rn

Table 25: ni format summary

A.14 Opcode assignment

The opcode assignments for each SHcompact instruction are given in *Chapter 2: SHcompact instruction set on page 19*.

A.15 Reserved instructions

Execution of a reserved opcode leads to a reserved instruction exception:

- The SHcompact instruction with encoding 0xFFFD is guaranteed to be reserved on all implementations. Execution of this SHcompact instruction will always result in either a RESINST exception if the instruction is not in a delay slot, or an ILLSLOT exception if the instruction is in a delay slot.
- SHcompact does not implement the privileged instructions of previous SuperH architectures. All privileged-mode instructions of previous SuperH architectures are reserved in the SHcompact architecture, and raise a reserved instruction exception if executed. The non-implemented instructions are listed in *Table 26*.
- Software should not rely on a RESINST exception for the execution of other reserved opcodes. On a future implementation, any of these reserved opcodes can be used to expand the instruction set.

Reserved instruction	Binary encoding (bit 15 to bit 0)	Reserved instruction summary
LDC Rm, DBR	0100mmmm11111010	Copy general-purpose register to debug base register
LDC Rm, Rn_BANK	0100mmmm1nnn1110	Copy general-purpose register to register Rn in back bank: SR.RB = 0 selects BANK1 and SR.RB = 1 selects BANK0, where n is in [0,7]
LDC Rm, SPC	0100mmmm01001110	Copy general-purpose register to saved program counter
LDC Rm, SR	0100mmmm00001110	Copy general-purpose register to status register
LDC Rm, SSR	0100mmmm00111110	Copy general-purpose register to saved status register
LDC Rm, VBR	0100mmmm00101110	Copy general-purpose register to vector base register
LDC.L @Rm+, DBR	0100mmmm11110110	Load debug base register from memory with post-increment
LDC.L @Rm+, Rn_BANK	0100mmmm1nnn0111	Load register Rn in back bank from memory with post-increment: SR.RB = 0 selects BANK1 and SR.RB = 1 selects BANK0, where n is in [0,7]

Table 26: SHcompact reserved instructions

Reserved instruction	Binary encoding (bit 15 to bit 0)	Reserved instruction summary
LDC.L @Rm+, SPC	0100mmmm01000111	Load saved program counter from memory with post-increment
LDC.L @Rm+, SR	0100mmmm00000111	Load status register from memory with post-increment
LDC.L @Rm+, SSR	0100mmmm00110111	Load saved status register from memory with post-increment
LDC.L @Rm+, VBR	0100mmmm00100111	Load vector base register from memory with post-increment
LDTLB	0000000000111000	Load a TLB entry from PTEH and PTEL registers
RTE	0000000000101011	Return from exception
SLEEP	0000000000011011	Place the CPU into power-down mode
STC DBR, Rn	0000nnnn11111010	Copy debug base register to general-purpose register
STC SGR, Rn	0000nnnn00111010	Copy saved general register 15 to general-purpose register
STC Rm_BANK, Rn	0000nnnn1mmm0010	Copy register Rm in back bank to general-purpose register: SR.RB = 0 selects BANK1 and SR.RB = 1 selects BANK0, where m is in [0,7]
STC SPC, Rn	0000nnnn01000010	Copy saved program counter to general-purpose register
STC SR, Rn	0000nnnn00000010	Copy status register to general-purpose register
STC SSR, Rn	0000nnnn00110010	Copy saved status register to general-purpose register
STC VBR, Rn	0000nnnn00100010	Copy vector base register to general-purpose register
STC.L DBR, @-Rn	0100nnnn11110010	Store debug base register to memory with pre-decrement
STC.L SGR, @-Rn	0100nnnn00110010	Store saved general register 15 to memory with pre-decrement

Table 26: SHcompact reserved instructions

Reserved instruction	Binary encoding (bit 15 to bit 0)	Reserved instruction summary
STC.L Rm_BANK, @-Rn	0100nnnn1mmm0011	Store contents of register Rm in back bank to memory with pre-decrement: SR.RB = 0 selects BANK1 and SR.RB = 1 selects BANK0, where m is in [0,7]
STC.L SPC, @-Rn	0100nnnn01000011	Store saved program counter to memory with pre-decrement
STC.L SR, @-Rn	0100nnnn00000011	Store status register to memory with pre-decrement
STC.L SSR, @-Rn	0100nnnn00110011	Store saved status register to memory with pre-decrement
STC.L VBR, @-Rn	0100nnnn00100011	Store vector base register to memory with pre-decrement
(no mnemonic)	1111111111111101	0xFFFD is guaranteed to be a reserved instruction and always generates a reserved instruction exception

Table 26: SHcompact reserved instructions

There is no provision by the architecture for any additional state required by just these unimplemented instructions. This includes DBR, PTEH, PTEL, SGR, SR.RB and the back-bank of 8 32-bit registers.

A.16 Floating-point instructions

The floating-point instruction set consists of:

- All instructions where the highest 4 bits (bit 12 to bit 15) of the instruction encoding have the value 0xF, excluding the reserved instruction which has encoding 0xFFFD.
- The LDS, STS, LDS.L and STS.L instructions that access FPUL and FPSCR

An implementation can choose not to provide floating-point and SR.FD will then always read as 1. If an implementation provides floating-point, software can disable it by setting the SR.FD flag. In both of these cases, execution of an instruction from the floating-point instruction set leads to an FPU disabled exception.

The FPU disabled exception (FPUDIS) takes precedence over a reserved instruction exception (RESINST). Thus, execution of a reserved floating-point instruction, that is not in a delay slot and where the floating-point instruction set is not available, leads to an FPU disabled exception.

Similarly, the delay-slot FPU disabled exception (SLOTFPUDIS) takes precedence over an illegal slot exception (ILLSLOT). Thus, execution of a reserved floating-point instruction, that is in a delay slot and where the floating-point instruction set is not available, leads to a delay-slot FPU disabled exception.

The SHcompact instruction with encoding 0xFFFD is not considered an FPU instruction. Regardless of whether the FPU is enabled or disabled, execution of this instruction will result in a RESINST exception if the instruction is not in a delay slot, or an ILLSLOT exception if the instruction is in a delay slot. This instruction does not cause FPUDIS or SLOTFPUDIS exceptions.

Index

A

ADD 20-21, 68, 264, 268
ADDC 22
ADDV 23
AND 24-26, 268
AND.B .26, 268

B

BACK 9
BANK1 269-271
BF 12, 27, 29
BRA 12, 31, 266
BRAF 12, 32
BREAK 33
BRK 33
BSR 12, 17, 34, 266
BSRF 12, 17, 36, 262
BT 12, 38, 40, 266

C

CMPGT 74

D

DIV0S 54
DIV1 56
DMULS.L 57
DMULU.L 58
DT 59

E

EXTS.B 60
EXTS.W 61
EXTU.B 62
EXTU.W 63

F

FABS 64-65
FADD 66-68
FCNVDS 75, 77
FCNVSD 76-77
FDIV 78-80
FIPR 82-83
FIPR.S 83
FLDI 85-86
FLDS 87
FLOAT 88-90
FMAC 91
FMAC.S 92
FMOV 95-117
FMOV.S 100-103, 112-114

FMUL 118-120
FNEG 121-122
FPU 2, 13, 68, 71, 74, 77, 80, 83, 90, 92,
. . 120, 125, 129, 131, 135, 138,
140, ...272
FPUDIS 13, 64-67, 69-70, 72-73, 75-76,
78-. . . 79, 82, 85-89, 91, 95-119,
121-124, 126-128, 130, 132-134,
136-.137, 140, 148-151, 235-238
FPUL . . 2, 75-76, 87-90, 124, 132, 136-
137, ...150-151, 237-238, 272
FRONT 9
FSCA 124-125, 130
FSQRT 127-129
FSRRA 130-131
FSRRA.S 131
FSTS 132
FSUB 133-135
FTRC 136-138
FTRV 139-140
FTRV.S 140
Function
IsDelaySlot() 13
PackFPSCR .11
SignExpectn(value) 9
UnpackFPSCR 11
ZeroExpectn(value) 9

I

IADDERR .27, 29, 31-32, 34, 36, 38, 40,
143-. 144, 218
ILLSLOT 12, 17, 27-32, 34, 36-41, 143-
145, . . .183, 194, 196, 218, 252
ISA . 13, 15-16, 32, 36-37, 143-145, 218

J

JMP 12, 143, 262
JSR .12, 17, 144

L

LDC 146-147, 263, 269-270
LDC.L 147, 263, 269-270
LDS 17, 148-157, 263, 272
LDS.L 149, 151, 153, 155, 157, 272
LDTLB 270
LTB 270

M

MAC.L 158-159, 264
MAC.W 160-161
MACH 2, 42, 57-58, 152-153, 158, 160,
200-. 202, 239-240, 262-263
MACL . 2, 42, 57-58, 154-155, 158, 160,
200-. 202, 241-242
MOV 12-13, 162-195, 264-268
MOV.B 164-173, 264-266
MOV.L 12, 174-184, 264-267
MOV.W 13, 185-195, 264-265, 267
MOVA 13, 196, 266
MOVCA.L 197
MOVT 199
MUL.L 200
MULS.W 201
MULU.W 202

N

NEG 203
NEGC 204
NOT 206

O

OCBI 207
OCBP 208
OCBWB 209
OR 210-212

OR.B 212

P

PC 12-13, 15-17, 27, 29, 31-32, 34, 36, 38, .40, 143-144, 183, 194, 196, 218, ... 266-267
PR . 1-2, 11, 13, 15-17, 34, 36-37, 144-145, ...156-157, 218, 243-244
PREF 213
PTEH 270
PTEL 270

R

Reggister
  VBR 269-271
Register
  DR 10
  FP 10
  FPSCR . 1-2, 11, 66-68, 71, 74-80, 83, 88- . . 89, 91-92, 118-120, 123, 126-129, . . 131, 133-138, 140, 148-149, 235- 236, 272
  FPSCR.DN . 68, 71, 74, 77, 80, 83, 92, 120, ...129, 131, 135, 138, 140
  FPSCR.FR 2, 123
  FPSCR.RM 66-67, 75-76, 78-79, 88-89, . . . 91, 118-119, 127-128, 133-134, 136- 137
  FPSCR.SZ 2, 126
  FR 1-2, 9, 11, 91, 123
  GBR . . .2, 26, 146-147, 167, 172, 177, 182, . .188, 193, 212, 233-234, 255, 258, ...262-263, 266-268
  R 13-14, 25-26, 46, 99, 103, 108, 111, 114, . . 117, 166-168, 171-173, 176-177, . . 181-182, 187-189, 192-193, 195- . . 197, 211-212, 254-255, 257-258, ... 264-266, 268
  Rm 20, 22-24, 45, 47-50, 53-54, 56-58, . 60-63, 109-117, 146-162, 164-166, 169- . . 171, 173-176, 178-181, 184-187, . 189-192, 195, 200-204, 206, 210, .221, 224, 245-249, 253, 256, 259, ...263-265, 269-270
  SR 1-2, 9, 11, 269-272
  SR.FD 272
  SR.FR .2, 9, 11
  SR.PR 2, 11
  SR.RB 269-271
  SR.SZ 2, 11
  SSR 269-271
  TR 14
  XFi 10
Regsiter
  DBR 269-270
  SGR 270
Ri_BANK 269-271
ROTCL 214
ROTCR 215
ROTL 216
ROTR 217
RTE 261, 270

S

SHAD 221
SHAL 222
SHAR 223
SHLD 224
SHLL 225-228, 262
SHLR 229-232
SLEEP 261, 270
SLOTFPUDIS . .13, 64-67, 69-70, 72-73, 75-. 76, 78-79, 82, 85-89, 91, 95-119, 121-124, 126-128, 130, 132-134, 136-137, 140, 148-151, 235-238
SPC 269-271
STC 233-234, 262-263, 270-271
STC.L 234, 262, 270-271
STS 17, 235-244, 262, 272

STS.L 236, 238, 240, 242, 244, 262, 272
SUB 135, 245
SUBC 246
SUBV 247
SWAP.B 248
SWAP.W 249
SZ 1-2, 11, 126

T

TAS.B 250-251
TRAPA 13, 252, 268
TST 253-255, 268
TST.B 255

XYZ

XMTRX 10, 139
XOR 256-258, 264
XOR.B 258
XTRCT 259