To our customers,

---

**Old Company Name in Catalogs and Other Documents**

On April 1\(^{st}\), 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: [http://www.renesas.com](http://www.renesas.com)

April 1\(^{st}\), 2010
Renesas Electronics Corporation

---

Issued by: Renesas Electronics Corporation ([http://www.renesas.com](http://www.renesas.com))

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.

2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.

3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.

4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.

5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.

6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.

7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depend on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g., artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g., excision, etc.), and any other applications or purposes that pose a direct threat to human life.

8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.

9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.

10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.

11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.

12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.
The revision list can be viewed directly by clicking the title page.

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

SH-2A, SH2A-FPU
Software Manual
Renesas 32-Bit RISC Microcomputer
SuperH™ RISC engine Family
1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corp. product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corp. or a third party.

2. Renesas Technology Corp. assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.

3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corp. without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor for the latest product information before purchasing a product listed herein.

The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corp. assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.

Please also pay attention to information published by Renesas Technology Corp. by various means, including the Renesas Technology Corp. Semiconductor home page (http://www.renesas.com).

4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corp. assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.

5. Renesas Technology Corp. semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.

6. The prior written approval of Renesas Technology Corp. is necessary to reprint or reproduce in whole or in part these materials.

7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.

8. Please contact Renesas Technology Corp. for further details on these materials or the products contained therein.

Keep safety first in your circuit designs!

1. Renesas Technology Corp. puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corp. product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corp. or a third party.

2. Renesas Technology Corp. assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.

3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corp. without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor for the latest product information before purchasing a product listed herein.

The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corp. assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.

Please also pay attention to information published by Renesas Technology Corp. by various means, including the Renesas Technology Corp. Semiconductor home page (http://www.renesas.com).

4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corp. assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.

5. Renesas Technology Corp. semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.

6. The prior written approval of Renesas Technology Corp. is necessary to reprint or reproduce in whole or in part these materials.

7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.

8. Please contact Renesas Technology Corp. for further details on these materials or the products contained therein.
Main Revisions for this Edition

<table>
<thead>
<tr>
<th>Item</th>
<th>Page</th>
<th>Revision (See Manual for Details)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Features</td>
<td>1</td>
<td>Description amended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SH-2A/SH2A-FPU is a 32-bit RISC (reduced instruction set computer) microprocessor that is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upward-compatible with the SH-1, SH-2, and SH-2E at the object code level.</td>
</tr>
<tr>
<td>2.2.2 Control Registers</td>
<td>5</td>
<td>Description amended</td>
</tr>
<tr>
<td>(1) Status Register,</td>
<td></td>
<td>(32-bit, initial value =0000 0000 0000 0000 00X1 00XX 1111 00XX)</td>
</tr>
<tr>
<td>SR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.1 Exception</td>
<td>16</td>
<td>Note amended</td>
</tr>
<tr>
<td>Handling Types and</td>
<td></td>
<td>Notes: 1. Delayed branch instructions: JMP, JSR, BRA, BSR, RTS, RTE, BF/S, BT/S, BSRF, BRAF</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1.2 Exception</td>
<td>18</td>
<td>Description amended</td>
</tr>
<tr>
<td>Handling Operation</td>
<td></td>
<td>⋯ and the vector table address offset of the interrupt exception handling to be executed,⋯</td>
</tr>
<tr>
<td>(2) Address Error,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM Error, Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank Error, Interrupt,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3.1 Address Error</td>
<td>22</td>
<td>Table amended</td>
</tr>
<tr>
<td>Sources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 3.5 Bus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycles and Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Errors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6.3 Interrupt</td>
<td>26</td>
<td>Description amended</td>
</tr>
<tr>
<td>Exception Handling</td>
<td></td>
<td>⋯ and the vector table address offset of the interrupt exception handling to be executed,⋯</td>
</tr>
<tr>
<td>Item</td>
<td>Page</td>
<td>Revision (See Manual for Details)</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>4.3 Instruction Format</td>
<td>45</td>
<td>Table amended</td>
</tr>
<tr>
<td>Table 4.8 Instruction Formats</td>
<td></td>
<td>Instruction Formats</td>
</tr>
<tr>
<td>nid format</td>
<td></td>
<td>Table amended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table amended</td>
</tr>
<tr>
<td>5.1 Instruction Set by Classification</td>
<td>53</td>
<td>Table amended</td>
</tr>
<tr>
<td>Table 5.2 Instruction Code Format</td>
<td></td>
<td>Item Format Explanation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rm: Source register</td>
</tr>
<tr>
<td>5.1.1 Data Transfer Instructions</td>
<td>56</td>
<td>Table amended</td>
</tr>
<tr>
<td>Table 5.3 Data Transfer Instructions</td>
<td></td>
<td>Instruction Code Format</td>
</tr>
<tr>
<td>MOVML.L @R15+,Rn</td>
<td></td>
<td>Instruction Code Format</td>
</tr>
<tr>
<td>MOVMU.L @R15+,Rn</td>
<td></td>
<td>Instruction Code Format</td>
</tr>
<tr>
<td>Note: When Rn = R15, read Rn as PR</td>
<td></td>
<td>Instruction Code Format</td>
</tr>
<tr>
<td>6.2 Format of Instruction Descriptions</td>
<td>76</td>
<td>Description amended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register bank structure definition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(VTO: Interrupt vector table address offset)</td>
</tr>
<tr>
<td>6.3.30 RESBANK RESStore from registerBANK System Control Instruction</td>
<td>145</td>
<td>Note amended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* 19 when a bank overflow has occurred and the register is restored from the stack</td>
</tr>
<tr>
<td>6.4.21 DT Decrement and Test Arithmetic Instruction</td>
<td>196</td>
<td>Program listing amended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DT R#</td>
</tr>
<tr>
<td>6.4.31 MOV MOVe immediate data Data Transfer Instruction</td>
<td>219</td>
<td>Description amended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>... The PC points to the starting address of the fourth byte after this MOV instruction. ... The PC points to the starting address of the fourth byte after this MOV instruction, ...</td>
</tr>
</tbody>
</table>

Rev. 3.00 Jul 08, 2005 page iv of xiv
<table>
<thead>
<tr>
<th>Item</th>
<th>Page</th>
<th>Revision (See Manual for Details)</th>
</tr>
</thead>
</table>
| 6.4.48 RTE  
ReTurn from Exception System Control Instruction | 244 | Description amended |
|  |  | Return from Exception Handling Delayed Branch Instruction |
| 6.4.50 SETT  
SET bit System Control Instruction | 248 | Description amended |
|  | | T Bit Setting |
| 6.4.57 SLEEP  
SLEEP System Control Instruction | 257 | Description amended |
|  | | Transition to Power-Down Mode |
| 6.5.10 FLOAT  
Floating-point convert from integer Floating-Point Instruction | 296 | Description amended |
|  | | ... When FPSCR.enable.I = 1, and FPSCR.PR = 0, an FPU exception trap is generated regardless of whether or not an exception has occurred. |
| 7.1 Overview  
Figure 7.1 Overview of Register Bank Configuration | 325 | Figure amended |
|  | (Before) IVO → (After) VTO | Figure notes amended |
|  | | VTO: Interrupt vector table address offset |
| 7.2.1 Banked Data  
Figure 7.2 Bank Save Operations | 326 | Description amended |
|  | | ... and the interrupt vector table address offsets (VTO) are banked. |
| 7.2.2 Register Banks | 326 | Description amended |
|  | | Register banks are stacked in first in last out (FILO) sequence. |
| 7.2.3 Bank Control Registers  
(2) Bank Number Register (IBNR) (16 bit, Initial value: H’0000) | 327 | Description amended |
|  | | Bits 3 to 0: BN3 to BN0 |
|  | | ... after which the data is retrieved from the register bank. These bits are read-only and cannot be modified. |
| 7.3.1 Save to Bank  
Figure 7.3 Bank Save Timing | 328 | Description amended |
|  | (b) ..., and the interrupt vector table address offset (VTO) are saved to the bank indicated by the BN, bank i. |

---

Figure 7.1 Overview of Register Bank Configuration

Figure 7.2 Bank Save Operations

Figure 7.3 Bank Save Timing

---

Rev. 3.00 Jul 08, 2005 page v of xiv
7.4.2 Register Bank Addressing

Description amended

...and the entry within the bank (R0 to R14, GBR, MACH, MACL, PR, VTO) is specified by address bits 6 to 2 (EN).

Figure 7.4 Register Bank Addressing

Figure amended

(Before) IVO → (After) VTO

8.2 Slots and Pipeline Flow

Figure 8.3 Impossible Pipeline Flow (1)

Figure amended

Instruction 1 IF ID EX MA WB

8.6 Contention Due to FPU

Figure 8.36 Example of Use of Result of Zero-Latency Instruction as Source

Figure amended

(Before) GX → (After) EX

8.9 Pipeline Operations for Each Instruction

Table 8.1 Number of Instruction Stages and Execution States

Table amended

<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>System control</td>
<td>MAC → register</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
<td>STS, MACH,Rn</td>
</tr>
<tr>
<td>System control</td>
<td>register transfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STS, MACL,Rn</td>
</tr>
<tr>
<td>System control</td>
<td>System</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Appendix A SH-2A/SH2A-FPU Parallel Execution

Table amended

<table>
<thead>
<tr>
<th>Classification of First Instruction</th>
<th>Classification of Second Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rev. 3.00 Jul 08, 2005 page vi of xiv
Contents

Section 1  Overview............................................................................................................. 1
  1.1 Features.................................................................................................................... 1

Section 2  Programming Model........................................................................................ 3
  2.1 Data Formats............................................................................................................. 3
  2.2 Register Configuration.............................................................................................. 3
    2.2.1 General Registers................................................................................................. 3
    2.2.2 Control Registers.................................................................................................. 5
    2.2.3 System Registers.................................................................................................. 6
    2.2.4 Floating-Point Registers ...................................................................................... 7
    2.2.5 Floating-Point System Registers.......................................................................... 8
    2.2.6 Register Banks..................................................................................................... 10
    2.2.7 Register Initial Values......................................................................................... 10
  2.3 Data Formats............................................................................................................. 11
    2.3.1 Data Format in Registers..................................................................................... 11
    2.3.2 Data Formats in Memory..................................................................................... 11
    2.3.3 Immediate Data Format ....................................................................................... 12
  2.4 Processing States....................................................................................................... 13

Section 3  Exception Handling ......................................................................................... 15
  3.1 Overview.................................................................................................................... 15
    3.1.1 Exception Handling Types and Priority............................................................... 15
    3.1.2 Exception Handling Operation ............................................................................ 17
    3.1.3 Exception Vector Table ....................................................................................... 18
  3.2 Resets....................................................................................................................... 20
    3.2.1 Types of Reset ..................................................................................................... 20
    3.2.2 Power-On Reset ................................................................................................. 20
    3.2.3 Manual Reset ..................................................................................................... 21
  3.3 Address Errors.......................................................................................................... 22
    3.3.1 Address Error Sources ......................................................................................... 22
    3.3.2 Address Error Exception Handling...................................................................... 23
  3.4 RAM Errors............................................................................................................ 23
    3.4.1 RAM Error Sources ............................................................................................. 23
    3.4.2 RAM Error Exception Handling.......................................................................... 23
  3.5 Register Bank Errors............................................................................................... 24
    3.5.1 Register Bank Error Sources .............................................................................. 24
    3.5.2 Register Bank Error Exception Handling............................................................ 24
  3.6 Interrupts.................................................................................................................. 25
6.3.4 BLD ........ Bit LoaD ........................................ Bit Manipulation Instruction ... 94
6.3.5 BLDNOT ... Bit LoaDNOT ................................ Bit Manipulation Instruction ... 96
6.3.6 BOR ........ Bit OR ........................................ Bit Manipulation Instruction ... 98
6.3.7 BORNOT ... Bit ORNOT .................................. Bit Manipulation Instruction ... 100
6.3.8 BSET ....... Bit SET ....................................... Bit Manipulation Instruction ... 102
6.3.9 BST ........ Bit STore ..................................... Bit Manipulation Instruction ... 104
6.3.10 BXOR ...... Bit exclusive OR .......................... Bit Manipulation Instruction ... 106
6.3.11 CLIPS ...... CLIP as Signed .......................... Arithmetic Instruction .......... 108
6.3.12 CLIPU ...... CLIP as Unsigned ........................ Arithmetic Instruction .......... 111
6.3.13 DIVS ........ DIVide as Signed ........................ Arithmetic Instruction .......... 113
6.3.14 DIVU ....... DIVide as Unsigned ........................ Arithmetic Instruction .......... 114
6.3.15 FMOV ...... Floating-point MOVe .................. Floating-Point Instruction ...... 115
6.3.16 JSR/N ...... Jump to SubRoutine with No delay slot ........................................... Branch Instruction ............... 118
6.3.17 LDBANK ... Load register BANK ................. System Control Instruction ..... 121
6.3.18 LDC ........ Load to Control register ............. System Control Instruction ..... 123
6.3.19 MOV ......... MOVe structure data ................. Data Transfer Instruction ........ 124
6.3.20 MOV ...... MOVe reverse stack ..................... Data Transfer Instruction ........ 127
6.3.21 MOVI20 ... MOVe Immediate 20bits data .... Data Transfer Instruction ........ 130
6.3.22 MOVI20S .. MOVe Immediate 20bits data and 8bits Shift left ................................ Data Transfer Instruction ....... 131
6.3.23 MOVML.L MOVe Multi-register Lower part Data Transfer Instruction .......... 133
6.3.24 MOVMU.L MOVe Multi-register Upper part Data Transfer Instruction .......... 136
6.3.25 MOVRT .... MOVe Reverse Tbit .................... Data Transfer Instruction ........ 139
6.3.26 MOVU ...... MOVe structure data as Unsigned Data Transfer Instruction ........ 140
6.3.27 MULR ...... MULtiply to Register ................ Arithmetic Instruction .......... 142
6.3.28 NOTT ...... NOT Tbit .................................... Data Transfer Instruction .......... 143
6.3.29 PREF ...... PREFetch data to cache ............... Data Transfer Instruction .......... 144
6.3.30 RESBANK REStore from registerBANK ...... System Control Instruction ..... 145
6.3.31 RTS/N ...... ReTurn from Subroutine with No delay slot ..................................... Branch Instruction ............... 147
6.3.32 RTV/N .... ReTurn to Value and from subroutine with No delay slot ................................ Branch Instruction ............... 148
6.3.33 SHAD ...... Shift Arithmetic Dynamically .... Shift Instruction ....................... 150
6.3.34 SHLD ...... Shift Logical Dynamically .......... Shift Instruction ....................... 152
6.3.35 STBANK ... STore register BANK .............. System Control Instruction ..... 154
6.3.36 STC ........ STore Control register .............. System Control Instruction ..... 156
6.4 SH-2E CPU Instructions ................................................................................................... 157
6.4.1 ADD ........ ADD Binary .............................. Arithmetic Instruction .......... 157
6.4.2 ADDC ...... ADD with Carry ........................... Arithmetic Instruction .......... 158
<table>
<thead>
<tr>
<th>Section</th>
<th>Instruction</th>
<th>Description</th>
<th>Instruction Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.3</td>
<td>ADDV</td>
<td>ADD with (V flag) overflow check</td>
<td>Arithmetic Instruction</td>
<td>159</td>
</tr>
<tr>
<td>6.4.4</td>
<td>AND</td>
<td>AND logical</td>
<td>Logical Instruction</td>
<td>161</td>
</tr>
<tr>
<td>6.4.5</td>
<td>BF</td>
<td>Branch if False</td>
<td>Branch Instruction</td>
<td>163</td>
</tr>
<tr>
<td>6.4.6</td>
<td>BF/S</td>
<td>Branch if False with delay Slot</td>
<td>Branch Instruction</td>
<td>165</td>
</tr>
<tr>
<td>6.4.7</td>
<td>BRA</td>
<td>BRAnch</td>
<td>Branch Instruction</td>
<td>167</td>
</tr>
<tr>
<td>6.4.8</td>
<td>BRAF</td>
<td>BRAnch Far</td>
<td>Branch Instruction</td>
<td>169</td>
</tr>
<tr>
<td>6.4.9</td>
<td>BSR</td>
<td>Branch to SubRoutine</td>
<td>Branch Instruction</td>
<td>171</td>
</tr>
<tr>
<td>6.4.10</td>
<td>BSRF</td>
<td>Branch to SubRoutine Far</td>
<td>Branch Instruction</td>
<td>173</td>
</tr>
<tr>
<td>6.4.11</td>
<td>BT</td>
<td>Branch if True</td>
<td>Branch Instruction</td>
<td>175</td>
</tr>
<tr>
<td>6.4.12</td>
<td>BT/S</td>
<td>Branch if True with delay Slot</td>
<td>Branch Instruction</td>
<td>177</td>
</tr>
<tr>
<td>6.4.13</td>
<td>CLRMAC</td>
<td>Clear MAC register</td>
<td>System Control Instruction</td>
<td>179</td>
</tr>
<tr>
<td>6.4.14</td>
<td>CLRT</td>
<td>Clear T bit</td>
<td>System Control Instruction</td>
<td>180</td>
</tr>
<tr>
<td>6.4.15</td>
<td>CMP/cond</td>
<td>Compare conditionally</td>
<td>Arithmetic Instruction</td>
<td>181</td>
</tr>
<tr>
<td>6.4.16</td>
<td>DIVOS</td>
<td>DIVide (step 0) as Signed</td>
<td>Arithmetic Instruction</td>
<td>185</td>
</tr>
<tr>
<td>6.4.17</td>
<td>DIVOU</td>
<td>DIVide (step 0) as Unsigned</td>
<td>Arithmetic Instruction</td>
<td>186</td>
</tr>
<tr>
<td>6.4.18</td>
<td>DIV1</td>
<td>DIVide 1 step</td>
<td>Arithmetic Instruction</td>
<td>187</td>
</tr>
<tr>
<td>6.4.19</td>
<td>DMULS.L</td>
<td>Double-length MULtiply as Signed</td>
<td>Arithmetic Instruction</td>
<td>192</td>
</tr>
<tr>
<td>6.4.20</td>
<td>DMULU.L</td>
<td>Double-length MULtiply as Unsigned</td>
<td>Arithmetic Instruction</td>
<td>194</td>
</tr>
<tr>
<td>6.4.21</td>
<td>DT</td>
<td>Decrement and Test</td>
<td>Arithmetic Instruction</td>
<td>196</td>
</tr>
<tr>
<td>6.4.22</td>
<td>EXTS</td>
<td>EXTend as Signed</td>
<td>Arithmetic Instruction</td>
<td>197</td>
</tr>
<tr>
<td>6.4.23</td>
<td>EXTU</td>
<td>EXTend as Unsigned</td>
<td>Arithmetic Instruction</td>
<td>198</td>
</tr>
<tr>
<td>6.4.24</td>
<td>JMP</td>
<td>Jump</td>
<td>Branch Instruction</td>
<td>199</td>
</tr>
<tr>
<td>6.4.25</td>
<td>JSR</td>
<td>Jump to SubRoutine</td>
<td>Branch Instruction</td>
<td>201</td>
</tr>
<tr>
<td>6.4.26</td>
<td>LDC</td>
<td>Loaad to Control register</td>
<td>System Control Instruction</td>
<td>203</td>
</tr>
<tr>
<td>6.4.27</td>
<td>LDS</td>
<td>Loaad to System register</td>
<td>System Control Instruction</td>
<td>205</td>
</tr>
<tr>
<td>6.4.28</td>
<td>MAC.L</td>
<td>Multiply and ACCumulate Long</td>
<td>Arithmetic Instruction</td>
<td>207</td>
</tr>
<tr>
<td>6.4.29</td>
<td>MAC.W</td>
<td>Multiply and ACCumulate Word</td>
<td>Arithmetic Instruction</td>
<td>211</td>
</tr>
<tr>
<td>6.4.30</td>
<td>MOV</td>
<td>MOVE data</td>
<td>Data Transfer Instruction</td>
<td>214</td>
</tr>
<tr>
<td>6.4.31</td>
<td>MOV</td>
<td>MOVE immediate data</td>
<td>Data Transfer Instruction</td>
<td>219</td>
</tr>
<tr>
<td>6.4.32</td>
<td>MOV</td>
<td>MOVE peripheral Data</td>
<td>Data Transfer Instruction</td>
<td>222</td>
</tr>
<tr>
<td>6.4.33</td>
<td>MOV</td>
<td>MOVE structure data</td>
<td>Data Transfer Instruction</td>
<td>225</td>
</tr>
<tr>
<td>6.4.34</td>
<td>MOVA</td>
<td>MOVE effective Address</td>
<td>Data Transfer Instruction</td>
<td>228</td>
</tr>
<tr>
<td>6.4.35</td>
<td>MOVT</td>
<td>MOVE T bit</td>
<td>Data Transfer Instruction</td>
<td>230</td>
</tr>
<tr>
<td>6.4.36</td>
<td>MUL.L</td>
<td>MULtiply Long</td>
<td>Arithmetic Instruction</td>
<td>231</td>
</tr>
<tr>
<td>6.4.37</td>
<td>MULS.W</td>
<td>MULtiply as Signed Word</td>
<td>Arithmetic Instruction</td>
<td>232</td>
</tr>
<tr>
<td>6.4.38</td>
<td>MULU.W</td>
<td>MULtiply as Unsigned Word</td>
<td>Arithmetic Instruction</td>
<td>233</td>
</tr>
<tr>
<td>6.4.39</td>
<td>NEG</td>
<td>NEGate</td>
<td>Arithmetic Instruction</td>
<td>234</td>
</tr>
<tr>
<td>6.4.40</td>
<td>NEGC</td>
<td>NEGate with Carry</td>
<td>Arithmetic Instruction</td>
<td>235</td>
</tr>
</tbody>
</table>
6.4.41  NOP ........... No OPeration ........................................ System Control Instruction ... 236
6.4.42  NOT ........... NOT-logical complement .................. Logical Instruction ............... 237
6.4.43  OR .......... OR logical ........................................ Logical Instruction .................. 238
6.4.44  ROTCL ...... ROTate with Carry Left ................... Shift Instruction ................... 240
6.4.45  ROTCR ...... ROTate with Carry Right .................. Shift Instruction ................... 241
6.4.46  ROTL ....... ROTate Left .................................. Shift Instruction ................... 242
6.4.47  ROTR ........ ROTate Right ................................ Shift Instruction ................... 243
6.4.48  RTE ........... ReTurn from Exception ................. System Control Instruction ...... 244
6.4.49  RTS ........... ReTurn from Subroutine ................. Branch Instruction ................. 246
6.4.50  SETT ........ SET T bit ...................................... System Control Instruction ...... 248
6.4.51  SHAL ....... SHift Arithmetic Left ..................... Shift Instruction ................... 249
6.4.52  SHAR ........ SHift Arithmetic Right ................... Shift Instruction ................... 250
6.4.53  SHLL ...... SHift Logical Left .......................... Shift Instruction ................... 251
6.4.54  SHLLn ...... n bits SHift Logical Left ................. Shift Instruction ................... 252
6.4.55  SHLR ...... SHift Logical Right ....................... Shift Instruction ................... 253
6.4.56  SHLRn ...... n bits SHift Logical Right ............... Shift Instruction ................... 254
6.4.57  SLEEP .......... SLEEP ........................................ System Control Instruction .... 257
6.4.58  STC ........... STore Control register ................. System Control Instruction ...... 258
6.4.59  STS ........... STore System register .................. System Control Instruction ...... 260
6.4.60  SUB .......... SUBtract binary .......................... Arithmetic Instruction ............. 262
6.4.61  SUBC ......... SUBtract with Carry .................... Arithmetic Instruction ............. 263
6.4.62  SUBV ......... SUBtract with (V flag) underflow check Arithmetic Instruction ............. 263
6.4.63  SWAP ......... SWAP register halves .................... Data Transfer Instruction .......... 266
6.4.64  TAS ........... Test And Set ............................. Logical Instruction ................. 268
6.4.65  TRAPA ...... TRAP Always ................................ System Control Instruction ...... 269
6.4.66  TST ........... TeST logical ................................ Logical Instruction ..................... 271
6.4.67  XOR .......... eXclusive OR logical .................. Logical Instruction ..................... 271
6.4.68  XTRCT ...... eXTRact ..................................... Data Transfer Instruction .......... 275

6.5 Floating-Point Instructions and FPU-Related CPU Instructions................................. 276
6.5.1  FABS ...... Floating-point ABBsolute value ................ Floating-Point Instruction ...... 276
6.5.2  FADD ...... Floating-point ADD ................................ Floating-Point Instruction ...... 277
6.5.3  FCMP ...... Floating-point CoMPare .................... Floating-Point Instruction ...... 280
6.5.4  FCNVDS ... Floating-point CoNVert Double to Single precision Floating-Point Instruction ... 284
6.5.5  FCNVSD ... Floating-point CoNVert Single to Double precision Floating-Point Instruction ..... 287
6.5.6  FDIV ...... Floating-point DIVide ......................... Floating-Point Instruction ........ 289
6.5.7  FLDI0 ...... Floating-point LoaD Immediate 0.0 Floating-Point Instruction ............... 293
6.5.8 FLDI1 ....... Floating-point Load Immediate 1.0
...................................................... Floating-Point Instruction...... 294
6.5.9 FLDS ......... Floating-point Load to System register
...................................................... Floating-Point Instruction...... 295
6.5.10 FLOAT ...... Floating-point convert from integer
...................................................... Floating-Point Instruction...... 296
6.5.11 FMAC ........ Floating-point Multiply and Accumulate
...................................................... Floating-Point Instruction...... 298
6.5.12 FMOV ......... Floating-point MOVE ............... Floating-Point Instruction...... 304
6.5.13 FMUL ........ Floating-point MULtiply ............. Floating-Point Instruction...... 308
6.5.14 FNEG ......... Floating-point NEGate value ....... Floating-Point Instruction...... 310
6.5.15 FSCHG ...... Sz-bit CHanGe ............................ Floating-Point Instruction...... 311
6.5.16 FSQRT ....... Floating-point SQUARE Root ...... Floating-Point Instruction...... 312
6.5.17 FSTS .......... Floating-point STore System register
...................................................... Floating-Point Instruction...... 315
6.5.18 FSUB ......... Floating-point SUBtract .............. Floating-Point Instruction...... 316
6.5.19 FTRC ......... Floating-point TRuncate and Convert to integer
...................................................... Floating-Point Instruction...... 318
6.5.20 LDS ............ Load to FPU System register ...... System Control Instruction...... 321
6.5.21 STS ............ STore from FPU System register  System Control Instruction...... 323

Section 7  Register Banks ................................................................. 325
7.1 Overview ................................................................................. 325
7.2 Register Banks and Bank Control Registers ......................... 326
7.2.1 Banked Data ...................................................................... 326
7.2.2 Register Banks .................................................................. 326
7.2.3 Bank Control Registers ....................................................... 326
7.3 Bank Save and Retrieve Operations ......................................... 328
7.3.1 Save to Bank .................................................................... 328
7.3.2 Retrieve from Bank ............................................................ 329
7.3.3 Save and Retrieve Operations after Saving to All Banks .......... 329
7.4 Register Bank Data Send Instructions ....................................... 330
7.4.1 Description of Instructions .................................................. 330
7.4.2 Register Bank Addressing ................................................... 330
7.5 Register Bank Exceptions ....................................................... 332
7.5.1 Register Bank Error Sources .............................................. 332
7.5.2 Register Bank Error Exception Processing .......................... 332
7.6 SR Register Bank Overflow Bit (BO Bit)................................. 333

Section 8  Pipeline Operation ............................................................ 335
8.1 Basic Pipeline Configuration .................................................... 335
Section 1  Overview

1.1 Features

The SH-2A/SH2A-FPU is a 32-bit RISC (reduced instruction set computer) microprocessor that is upward-compatible with the SH-1, SH-2, and SH-2E at the object code level. The SH2A-FPU has an on-chip floating point unit and the SH-2A does not. The use of 16-bit basic instructions enables code efficiency, performance, and ease of use to be improved.

Features of the SH-2A/SH2A-FPU are summarized in table 1.1.

<table>
<thead>
<tr>
<th>Item</th>
<th>Features</th>
</tr>
</thead>
</table>
| CPU  | • Original Renesas Technology architecture  
      • 32-bit internal data bus  
      • General-register architecture  
      — Sixteen 32-bit general registers  
      — Four 32-bit control registers  
      — Four 32-bit system registers  
      — Register banks for fast interrupt response  
      • RISC-type instruction set (upward-compatible with SH Series)  
      — Instruction length: 16-bit basic instructions for improved efficiency, and 32-bit instructions for improved performance and ease of use  
      — Load-store architecture  
      — Delayed branch instructions  
      — Instruction set based on C language  
      • Superscalar architecture allowing simultaneous execution of two instructions, including FPU  
      • Instruction execution time: Max. 2 instructions/cycle  
      • Address space: 4 Gbytes  
      • On-chip multiplier  
      • Five-stage pipeline  
      • Harvard architecture |
### Floating-Point Unit (FPU)
- On-chip floating-point coprocessor
- Supports single-precision (32 bits) and double-precision (64 bits)
- Supports IEEE754-compliant data types and exceptions
- Two rounding modes: Round to Nearest and Round to Zero
- Handling of denormalized numbers: Truncation to zero
- Floating-point registers
  - Sixteen 32-bit floating-point registers (single-precision x 16 words or double-precision x 8 words)
  - Two 32-bit floating-point system registers
- Supports FMAC (multiply and accumulate) instruction
- Supports FDIV (divide) and FSQRT (square root) instructions
- Supports FLDI0/FLDI1 (load constant 0/1) instructions
- Instruction execution times
  - Latency (FMAC/FADD/FSUB/FMUL): 3 cycles (single-precision), 8 cycles (double-precision)
  - Pitch (FMAC/FADD/FSUB/FMUL): 1 cycle (single-precision), 6 cycles (double-precision)

Note: FMAC is supported for single-precision only.
- Five-stage pipeline
Section 2  Programming Model

2.1  Data Formats

Data formats supported by the SH-2A/SH2A-FPU are shown in figure 2.1.

<table>
<thead>
<tr>
<th>Format</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte (8 bits)</td>
<td>7, 0</td>
</tr>
<tr>
<td>Word (16 bits)</td>
<td>15, 0</td>
</tr>
<tr>
<td>Longword (32 bits)</td>
<td>31, 0</td>
</tr>
<tr>
<td>Single-precision floating-point (32 bits)</td>
<td>31, 30, 22, 0</td>
</tr>
<tr>
<td>Double-precision floating-point (64 bits)</td>
<td>63, 62, 51, 50</td>
</tr>
</tbody>
</table>

Figure 2.1  Data Formats

2.2  Register Configuration

2.2.1  General Registers

Figure 2.2 shows the general registers. There are 16 general registers (Rn) numbered R0 to R15, which are 32 bits in length. General registers are used for data processing and address calculation. R0 is also used as an index register. Several instructions use R0 as a fixed source or destination register. R15 is used as the hardware stack pointer (SP). Saving and recovering the status register (SR) and program counter (PC) in exception processing is accomplished by referencing the stack using R15.
<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0*1</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td></td>
</tr>
<tr>
<td>R15, SP (hardware stack pointer)*2</td>
<td></td>
</tr>
</tbody>
</table>

Notes:  
1. R0 functions as an index register in the indirect indexed register addressing mode and indirect indexed GBR addressing mode. In some instructions, R0 functions as a fixed source register or destination register.  
2. R15 functions as a hardware stack pointer (SP) during exception processing.

Figure 2.2 General Registers
2.2.2 Control Registers

There are four control registers, each 32 bits in length: the status register (SR), global base register (GBR), vector base register (VBR), and jump table base register (TBR).

The status register indicates the processing status of instructions.

The global base register is used as the base address in the GBR indirect addressing mode and to transfer register data from on-chip peripheral modules.

The vector base register is used as the base address for the exception processing vector area, including interrupts.

The table base register is used as the base address for the function table area.

(1) Status Register, SR

(32-bit, initial value = 0000 0000 0000 0000 00X0 00XX 1111 00XX) (X = undefined))

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>BO</td>
<td>CS</td>
<td>M</td>
<td>Q</td>
<td>IMASK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: —: Reserved bits. Always read as 0. The write value should always be 0.

**BO:** Indicates that a register bank has overflowed.

**CS:** Indicates that, in CLIP instruction execution, the value has exceeded the saturation upper-limit value or fallen below the saturation lower-limit value.

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

**IMASK:** Interrupt mask level

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

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

(2) Global Base Register, GBR (32-bit, initial value = undefined)

GBR is referenced as the base address in a GBR-referencing MOV instruction.

(3) Vector Base Register, VBR (32-bit, initial value = H'0000 0000)

VBR is referenced as the branch destination base address in the event of an exception or interrupt.
(4) Jump Table Base Register, TBR (32-bit, initial value = undefined)

TBR is referenced as the start address of a function table located in memory in a JSR/N @@(disp8,TBR) table referencing subroutine call instruction.

2.2.3 System Registers

System registers consist of four 32-bit registers: high and low multiply and accumulate registers (MACH and MACL), the procedure register (PR), and the program counter (PC). The multiply and accumulate registers store the results of multiply and multiply and accumulate operations. The procedure register stores the return address from the subroutine procedure. The program counter indicates the address of the program executing and controls the flow of the processing.

(1) Multiply and Accumulate Register High, MACH (32-bit, initial value = undefined)
Multiply and Accumulate Register Low, MACL (32-bit, initial value = undefined)

MACH/MACL is used as the addition value in a MAC instruction, and to store the operation result of a MAC or MUL instruction.

(2) Procedure Register, PR (32-bit, initial value = undefined)

PR stores the return address of a subroutine call using a BSR, BSRF, or JSR instruction, and is referenced by a subroutine return instruction (RTS).

(3) Program Counter, PC (32-bit, initial value = value of PC in vector table)

The PC indicates the address of the instruction being executed.
2.2.4 Floating-Point Registers

Figure 2.3 shows the floating-point registers. There are sixteen 32-bit floating-point registers, FPR0 to FPR15. These sixteen registers are referenced as FR0 to FR15 and DR0/2/4/6/8/10/12/14. The correspondence between FPRn and the reference name is determined by the PR bit and SZ bit in FPSCR. See figure 2.3.

(1) Floating-Point Registers, FPRn (16 Registers)

FPR0, FPR1, FPR2, FPR3, FPR4, FPR5, FPR6, FPR7,
FPR8, FPR9, FPR10, FPR11, FPR12, FPR13, FPR14, FPR15

(2) Single-Precision Floating-Point Registers, FRi (16 Registers)

FR0 to FR15 are assigned to FPR0 to FPR15.

(3) Double-Precision Floating-Point Registers or Single-Precision Floating-Point Register Pairs, DRi (8 Registers)

A DR register is composed of two FR registers.

DR0 = (FPR0, FPR1), DR2 = (FPR2, FPR3 ),
DR4 = (FPR4, FPR5), DR6 = (FPR6, FPR7),
DR8 = (FPR8, FPR9), DR10 = (FPR10, FPR11),
DR12 = (FPR12, FPR13), DR14 = (FPR14, FPR15)
In case of transfer instruction:
FPSCR.SZ = 0
FPSCR.PR = 0

In case of arithmetic/logical instruction:
FPSCR.SZ = 1
FPSCR.PR = 1

### Figure 2.3   Floating-Point Registers

**Programming Note:**

The values of FPR0 to FPR15 are undefined after a reset.

#### 2.2.5   Floating-Point System Registers

(1) **Floating-Point Communication Register, FPUL** (32-bit, initial value = undefined)

Data transfers between an FPU register and CPU register are performed via FPUL.

(2) **Floating-Point Status/Control Register, FPSCR** (32-bit, initial value = H'0004 0001)
**QIS:** sNaN is treated as qNaN or ±∞. Valid only when the V bit in the enable field of FPSCR is set to 1.

- QIS = 0: Processed as qNaN or ±∞.
- QIS = 1: Exception generated (processed same as sNaN).

**SZ:** Transfer Size Mode

- SZ = 0: The data size of an FMOV instruction is 32 bits.
- SZ = 1: The data size of an FMOV instruction is a 32-bit pair (64 bits).

**PR:** Precision Mode

- PR = 0: Floating-point instructions are executed as single-precision operations.
- PR = 1: Floating-point instructions are executed as double-precision operations (the result of an instruction for which double-precision is not supported is undefined).

**DN:** Denormalization Mode (always 1)

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

**Cause:** FPU exception cause field

**Enable:** FPU exception enable field

**Flag:** FPU exception flag field

<table>
<thead>
<tr>
<th></th>
<th>FPU Error (E)</th>
<th>Invalid Operation (V)</th>
<th>Division by Zero (Z)</th>
<th>Overflow (O)</th>
<th>Underflow (U)</th>
<th>Inexact Exception (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause</td>
<td>Bit 17</td>
<td>Bit 16</td>
<td>Bit 15</td>
<td>Bit 14</td>
<td>Bit 13</td>
<td>Bit 12</td>
</tr>
<tr>
<td>Enable</td>
<td>None</td>
<td>Bit 11</td>
<td>Bit 10</td>
<td>Bit 9</td>
<td>Bit 8</td>
<td>Bit 7</td>
</tr>
<tr>
<td>Flag</td>
<td>None</td>
<td>Bit 6</td>
<td>Bit 5</td>
<td>Bit 4</td>
<td>Bit 3</td>
<td>Bit 2</td>
</tr>
</tbody>
</table>

When an FPU operation instruction is executed, the FPU exception cause field is initially set to 0. When an FPU exception next occurs, the corresponding bit in the FPU exception cause field and FPU exception flag field is set to 1.

The FPU exception flag field retains the status of an exception generated after that field was last cleared.
RM: Rounding Mode

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

Bits 21, 23 to 31: Reserved

Note: The SH-2A does not generate an FPU error.

2.2.6 Register Banks

For the nineteen 32-bit registers comprising general registers R0 to R14, control register GBR, and system registers MACH, MACL, and PR, high-speed register saving and restoration can be carried out using a register bank. Saving to the bank is performed automatically after the CPU accepts an interrupt that uses a register bank. Restoration from the bank is executed by issuing a RESBANK instruction in an interrupt service routine.

For details, refer to section 7, Register Banks.

2.2.7 Register Initial Values

Table 2.1 Initial Values of Registers

<table>
<thead>
<tr>
<th>Classification</th>
<th>Register</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>General registers</td>
<td>R0–R14</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td>R15(SP)</td>
<td>SP value in the program address table</td>
</tr>
<tr>
<td>Control registers</td>
<td>SR</td>
<td>Bits I3–I0 are 1111 (H'F), BO, CS are 0, reserved bits are 0, and other bits are undefined</td>
</tr>
<tr>
<td></td>
<td>GBR, TBR</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td>VBR</td>
<td>H'00000000</td>
</tr>
<tr>
<td>System registers</td>
<td>MACH, MACL, PR</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td>PC</td>
<td>Value of the program counter in the vector address table</td>
</tr>
<tr>
<td>Floating-point registers</td>
<td>FRR0–FRR15</td>
<td>Undefined</td>
</tr>
<tr>
<td>Floating-point system registers</td>
<td>FPUL</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td>FPSCR</td>
<td>H'00040001</td>
</tr>
</tbody>
</table>
2.3 Data Formats

2.3.1 Data Format in Registers

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

2.3.2 Data Formats in Memory

Byte, word, and longword data formats are used. Memory can be accessed in 8-bit bytes, 16-bit words, or 32-bit longwords. A memory operand of fewer than 32 bits is stored in a register in sign-extended or zero-extended form.

A word operand should be accessed starting from a word boundary (2-byte even address: address 2n), and a longword operand from a longword boundary (4-byte even address: address 4n). If this rule is not observed, an address error will occur. A byte operand can be accessed from any address.

Only big-endian byte order can be selected for the data format.

Data formats in memory are shown in figure 2.4.

![Figure 2.4 Data Format in Memory](image-url)
2.3.3 Immediate Data Format

Byte immediate data is located in an instruction code. Immediate data accessed by the MOV, ADD, and CMP/EQ instructions is sign-extended and is handled in registers as longword data. Immediate data accessed by the TST, AND, OR, and XOR instructions is zero-extended and is handled as longword data. Consequently, AND instructions with immediate data always clear the upper 24 bits of the destination register.

20-bit immediate data is stored in the code of a MOVI20 or MOVI20S 32-bit transfer instruction. The MOVI20 instruction stores immediate data in the destination register in sign-extended form. The MOVI20S instruction shifts immediate data by 8 bits in the upper direction, and stores it in the destination register in sign-extended form.

Word or longword immediate data is not located in the instruction code but rather is stored in a memory table. The memory table is accessed by a immediate data transfer instruction (MOV) using the PC relative addressing mode with displacement.

Specific examples are given in 4.1, (10) Immediate Data in section 4, Instruction Features.
2.4 Processing States

The CPU has five processing states: the reset state, exception handling state, bus-released state, program execution state, and power-down state. Figure 2.5 shows the state transitions.

![Diagram of Processing State Transitions]

Figure 2.5 Processing State Transitions
(1) Reset State

In this state, the CPU is reset. There are two kinds of reset, power-on and manual. See the Hardware Manual for details.

(2) Exception Handling State

The exception handling state is a transient state that occurs when the CPU alters the normal programming flow due to a reset, interrupt, or other exception handling source.

In the case of a reset, the CPU fetches the execution start address as the initial value of the program counter (PC) from the exception vector table, and the initial value of the stack pointer (SP), stores these values, branches to the start address, and begins program execution at that address.

In the case of an interrupt, etc., the CPU references the SP and saves the PC and status register (SR) in the stack area. It fetches the start address of the exception service routine from the exception vector table, branches to that address, and begins program execution.

Subsequently, the processing state is the program execution state.

(3) Program Execution State

In the program execution state the CPU executes program instructions in the normal sequence.

(4) Power-Down State

In the power-down state the CPU stops operating to conserve power. Sleep mode or software standby mode is entered by executing a SLEEP instruction. If hardware standby input is received, the CPU enters the hardware standby mode.

(5) Bus-Released State

In the bus-released state, the CPU releases the bus to a device that has requested it.

Note: For information on the processing states, please refer to the hardware manual for the product in question.
Section 3 Exception Handling

3.1 Overview

3.1.1 Exception Handling Types and Priority

As table 3.1 indicates, exception handling may be caused by a reset, address error, RAM error, register bank error, interrupt, or instruction. Exception handling is prioritized as shown in table 3.1. If two or more exceptions occur simultaneously, they are accepted and processed in order of priority.
### Table 3.1  Exception Types and Priority

<table>
<thead>
<tr>
<th>Exception Handling</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td></td>
</tr>
<tr>
<td>Power-on reset</td>
<td>High</td>
</tr>
<tr>
<td>Manual reset</td>
<td></td>
</tr>
<tr>
<td>Address errors</td>
<td></td>
</tr>
<tr>
<td>CPU address error</td>
<td></td>
</tr>
<tr>
<td>DMAC address error</td>
<td></td>
</tr>
<tr>
<td>RAM errors</td>
<td></td>
</tr>
<tr>
<td>RAM error</td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td></td>
</tr>
<tr>
<td>FPU exception</td>
<td></td>
</tr>
<tr>
<td>Integer division exception (division by zero)</td>
<td></td>
</tr>
<tr>
<td>Integer division exception (overflow)</td>
<td></td>
</tr>
<tr>
<td>Register bank errors</td>
<td>Bank underflow</td>
</tr>
<tr>
<td>Bank overflow</td>
<td></td>
</tr>
<tr>
<td>Intermits</td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td></td>
</tr>
<tr>
<td>User break</td>
<td></td>
</tr>
<tr>
<td>H-UDI</td>
<td></td>
</tr>
<tr>
<td>External interrupt (IRQ)</td>
<td></td>
</tr>
<tr>
<td>On-chip peripheral modules</td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td></td>
</tr>
<tr>
<td>Trap instruction (TRAPA instruction)</td>
<td></td>
</tr>
<tr>
<td>General illegal instruction (undefined code)</td>
<td></td>
</tr>
<tr>
<td>Slot illegal instruction (undefined code (FPU instruction or FPU-related CPU instruction in module standby status including FPU or in product with no FPU, or register bank-related instruction<em>2 in product with no register bank) located immediately after delayed branch instruction</em>1, instruction that modifies PC<em>3, 32-bit instruction</em>4, RESBANK instruction, DIVS instruction, or DIVU instruction)</td>
<td>Low</td>
</tr>
</tbody>
</table>

**Notes:**
1. Delayed branch instructions: JMP, JSR, BRA, BSR, RTS, RTE, BF/S, BT/S, BSRF, BRAF
2. Register bank-related instructions: RESBANK, LDBANK, STBANK
3. Instructions that modify PC: JMP, JSR, BRA, BSR, RTS, RTE, BT, BF, TRAPA, BF/S, BT/S, BSRF, BRAF, JSR/N, RTV/N
4. 32-bit instructions: BAND.B, BANDNOT.B, BCLR.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BSET.B, BST.B, BXOR.B, FMov.S @disp12, FMov.D @disp12, MOV.B @disp12, MOV.W @disp12, MOV.L @disp12, MOVI20, MOVI20S, MOVU.B, MOVU.W
### 3.1.2 Exception Handling Operation

Table 3.2 shows the timing of detection and the start of exception handling for each exception source.

<table>
<thead>
<tr>
<th>Exception Handling</th>
<th>Exception Source Detection and Start of Exception Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Started by detection of power-on reset condition</td>
</tr>
<tr>
<td>Manual reset</td>
<td>Started by detection of manual reset condition</td>
</tr>
<tr>
<td>Address error</td>
<td>Detected when instruction is decoded; exception handling is started after completion of currently executing instruction</td>
</tr>
<tr>
<td>RAM error</td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td></td>
</tr>
<tr>
<td>Bank underflow</td>
<td>Started upon attempted execution of RESBANK instruction when save has not been performed to register bank</td>
</tr>
<tr>
<td>Bank overflow</td>
<td>Started when save has already been performed to all register bank areas when acceptance of register overflow exception has been set by interrupt controller, and interrupt that uses register bank is generated and accepted by CPU</td>
</tr>
<tr>
<td>Instruction Trap</td>
<td>Started by execution of TRAPA instruction</td>
</tr>
<tr>
<td>General illegal</td>
<td>Started when undefined code (FPU instruction or FPU-related CPU instruction in module standby status including FPU or in product with no FPU, or register bank-related instruction in product with no register bank) not immediately following delayed branch instruction (delay slot) is decoded</td>
</tr>
<tr>
<td>Slot illegal</td>
<td>Started when undefined code (FPU instruction or FPU-related CPU instruction in module standby status including FPU or in product with no FPU, or register bank-related instruction in product with no register bank) not immediately following delayed branch instruction (delay slot), instruction that modifies PC, 32-bit instruction, RESBANK instruction, DIVS instruction, or DIVU instruction is decoded</td>
</tr>
<tr>
<td>Integer division</td>
<td>Started upon detection of division-by-zero exception or overflow exception caused by dividing negative maximum value (H’80000000) by –1</td>
</tr>
<tr>
<td>Floating-point operation</td>
<td>Started by floating-point operation instruction invalid operation exception (stipulated by IEEE754), or overflow, underflow, or imprecision interrupt. Also started when qNaN or ±∞ is input to a floating-point operation instruction source</td>
</tr>
</tbody>
</table>
When exception handling is initiated, the CPU operates as follows.

(1) **Reset Exception Handling**

The initial values of the program counter (PC) and stack pointer (SP) are fetched from the exception vector table (addresses H'00000000 and H'00000004 in the case of a power-on reset, and addresses H'00000008 and H'0000000C in the case of a manual reset). See section 3.1.3, Exception Vector Table, for details of the exception vector table. Next, the vector base register is cleared to H'00000000, the interrupt mask bits (I3 to I0) in the status register (SR) are set to (H'F) (1111), and the BO and CS bits are initialized to 0. The BN bit in IBNR of INTC is also initialized to 0. In addition, in products with an FPU, FPSCR is initialized to H'00040001. Program execution starts from the PC address fetched from the exception vector table.

(2) **Address Error, RAM Error, Register Bank Error, Interrupt, or Instruction Exception Handling**

SR and PC are saved on the stack indicated by R15. In interrupt exception handling other than NMI and UBC, when register bank use has been set, general registers R0 to R14, control register GBR, system registers MACH, MACL, and PR, and the vector table address offset of the interrupt exception handling to be executed, are saved to the register bank. In the case of exception handling due to an address error, RAM error, register bank error, NMI interrupt or UBC interrupt, saving to a register bank is not performed. Also, when saving is performed to all register banks, automatic saving to the stack is performed instead of register bank saving. In this case, an interrupt controller setting must have been made for register bank overflow exceptions not to be accepted. If a setting has been made for register bank overflow exceptions to be accepted, a register bank overflow exception will be generated. In the case of interrupt exception handling, the interrupt priority level is written to the interrupt mask bits (I3 to I0) in SR. In address error, RAM error, and instruction exception handling, bits I3 to I0 are not affected. Next, the start address is fetched from the exception vector table and program execution is started from that address.

3.1.3 **Exception Vector Table**

Before exception handling is executed, the exception vector table must have been set up in memory. The exception vector table holds the start addresses of the exception service routines (the reset exception handling table holds the initial values of PC and SP).

A different vector number and vector table address offset are assigned to each exception source. The vector table address is calculated from the corresponding vector number and vector table address offset. In exception handling, the start address of the exception service routine is fetched from the exception vector table entry indicated by this vector table address.
The vector numbers and vector table address offsets are shown in table 3.3, and the method of calculating the vector table address in table 3.4.

### Table 3.3 Exception Vector Table

<table>
<thead>
<tr>
<th>Exception Source</th>
<th>Vector Number</th>
<th>Vector Table Address Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-on reset</td>
<td>PC</td>
<td>0 H'00000000 to H'00000003</td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td>1 H'00000004 to H'00000007</td>
</tr>
<tr>
<td>Manual reset</td>
<td>PC</td>
<td>2 H'00000008 to H'0000000B</td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td>3 H'0000000C to H'0000000F</td>
</tr>
<tr>
<td>General illegal instruction</td>
<td>4</td>
<td>H'00000010 to H'00000013</td>
</tr>
<tr>
<td>RAM error</td>
<td>5</td>
<td>H'00000014 to H'00000017</td>
</tr>
<tr>
<td>Slot illegal instruction</td>
<td>6</td>
<td>H'00000018 to H'0000001B</td>
</tr>
<tr>
<td>(Reserved for system)</td>
<td>7</td>
<td>H'0000001C to H'0000001F</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>H'00000020 to H'00000023</td>
</tr>
<tr>
<td>CPU address error</td>
<td>9</td>
<td>H'00000024 to H'00000027</td>
</tr>
<tr>
<td>DMAC address error</td>
<td>10</td>
<td>H'00000028 to H'0000002B</td>
</tr>
<tr>
<td>Interrupt</td>
<td>NMI</td>
<td>11 H'0000002C to H'0000002F</td>
</tr>
<tr>
<td></td>
<td>User break</td>
<td>12 H'00000030 to H'00000033</td>
</tr>
<tr>
<td>FPU exception</td>
<td>13</td>
<td>H'00000034 to H'00000037</td>
</tr>
<tr>
<td>H-UDI</td>
<td>14</td>
<td>H'00000038 to H'0000003B</td>
</tr>
<tr>
<td>Bank overflow</td>
<td>15</td>
<td>H'0000003C to H'0000003F</td>
</tr>
<tr>
<td>Bank underflow</td>
<td>16</td>
<td>H'00000040 to H'00000043</td>
</tr>
<tr>
<td>Integer division exception</td>
<td>17</td>
<td>H'00000044 to H'00000047</td>
</tr>
<tr>
<td>(division by zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer division exception (overflow)</td>
<td>18</td>
<td>H'00000048 to H'0000004B</td>
</tr>
<tr>
<td>(Reserved for system)</td>
<td>19</td>
<td>H'0000004C to H'0000004F</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>H'0000007C to H'0000007F</td>
</tr>
<tr>
<td>Trap instruction (user vector)</td>
<td>32</td>
<td>H'00000080 to H'00000083</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>H'000000FC to H'000000FF</td>
</tr>
<tr>
<td>External interrupt (IRQ), on-chip peripheral module*</td>
<td>64</td>
<td>H'00000100 to H'00000103</td>
</tr>
<tr>
<td></td>
<td>511</td>
<td>H'000007FC to H'000007FF</td>
</tr>
</tbody>
</table>

Note: * For the vector numbers and address offsets of external interrupts and on-chip peripheral module interrupts, see “Internal Module Interrupt Exception Handling Vectors and Priority Order” in the Interrupt Controller section of the hardware manual.
### Table 3.4 Exception Vector Table Address Calculation

<table>
<thead>
<tr>
<th>Exception Source</th>
<th>Vector Table Address Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Vector table address = (vector table address offset)</td>
</tr>
<tr>
<td></td>
<td>= (vector number) × 4</td>
</tr>
<tr>
<td>Address error, RAM error, register bank error,</td>
<td>Vector table address = VBR + (vector table address offset)</td>
</tr>
<tr>
<td>interrupt, instruction</td>
<td>= VBR + (vector number) × 4</td>
</tr>
</tbody>
</table>

**Note:**
- VBR: Vector base register
- Vector table address offset: See table 3.3.
- Vector number: See table 3.3.

### 3.2 Resets

#### 3.2.1 Types of Reset

A reset is the highest-priority exception handling source. There are two types of reset: a power-on reset and a manual reset. The CPU state is initialized by both a power-on reset and a manual reset. The FPU state is initialized by a power-on reset, but not by a manual reset. Refer to the hardware manual of the relevant product for information on the states of on-chip peripheral modules, the PFC, and I/O ports.

#### 3.2.2 Power-On Reset

When a power-on reset condition is detected, the chip enters the power-on reset state. See “Power-On Reset” in the Exception Handling section of the hardware manual for the relevant product for details of power-on reset conditions.

When the power-on reset state is released, power-on reset exception handling is started. CPU operations are as follows.

1. The initial value of the program counter (PC) (i.e. the execution start address) is fetched from the exception vector table.
2. The initial value of the stack pointer (SP) is fetched from the exception vector table.
3. The vector base register (VBR) is cleared to H'00000000, the interrupt mask bits (I3 to I0) in the status register (SR) are set to (H'F) (1111), and the BO and CS bits are initialized to 0. The BN bit in IBNR of INTC is also initialized to 0. In addition, in products with an FPU, FPSCR is initialized to H'00040001.
4. The values fetched from the exception vector table are set in the program counter (PC) and stack pointer (SP), and program execution is started.

Power-on reset processing must always be executed when the system is powered on.

3.2.3 Manual Reset

When a manual reset condition is detected, the chip enters the manual reset state. See “Manual Reset” in the Exception Handling section of the hardware manual for the relevant product for details of manual reset conditions.

When the manual reset state is released, manual reset exception handling is started. CPU operations are as follows.

1. The initial value of the program counter (PC) (i.e. the execution start address) is fetched from the exception vector table.
2. The initial value of the stack pointer (SP) is fetched from the exception vector table.
3. The vector base register (VBR) is cleared to H'00000000, the interrupt mask bits (I3 to I0) in the status register (SR) are set to (H'F) (1111), and the BO and CS bits are initialized to 0. The BN bit in IBNR of INTC is also initialized to 0.
4. The values fetched from the exception vector table are set in the program counter (PC) and stack pointer (SP), and program execution is started.

When a manual reset occurs, the bus cycle is held. If a manual reset occurs while the bus is released or during a DMAC burst transfer, manual reset exception handling is held pending until the CPU acquires the bus. However, if the interval from occurrence of a manual reset until the end of a bus cycle exceeds a given number of cycles, the internal manual reset source is not held pending but is ignored, and manual reset exception handling is not performed. See “Manual Reset” in the Exception Handling section of the hardware manual for the relevant product for details.

A manual reset initializes the CPU and the BN bit in IBNR of the INTC. The FPU and other modules are not initialized.
### 3.3 Address Errors

#### 3.3.1 Address Error Sources

Address errors occur in instruction fetches and data read/write accesses, as shown in table 3.5.

**Table 3.5  Bus Cycles and Address Errors**

<table>
<thead>
<tr>
<th>Bus Cycle Type</th>
<th>Bus Master</th>
<th>Bus Cycle Operation</th>
<th>Address Error Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>CPU</td>
<td>Instruction fetched from even address</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction fetched from odd address</td>
<td>Address error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction fetched from other than on-chip peripheral module space*</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction fetched from on-chip peripheral module space*</td>
<td>Address error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instruction fetched from external memory space in single-chip mode</td>
<td>Address error</td>
</tr>
<tr>
<td>Data read/write</td>
<td>CPU or DMAC</td>
<td>Word data accessed from even address</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Word data accessed from odd address</td>
<td>Address error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Longword data accessed from longword boundary</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Longword data accessed from other than longword boundary</td>
<td>Address error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double longword data accessed from double longword boundary</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double longword data accessed from other than double longword boundary</td>
<td>Address error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Word data or byte data accessed in on-chip peripheral module space*</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Longword data accessed in 16-bit on-chip peripheral module space*</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Longword data accessed in 8-bit on-chip peripheral module space*</td>
<td>No error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>External memory space accessed in single-chip mode</td>
<td>Address error</td>
</tr>
</tbody>
</table>

*Note: * For details of the on-chip peripheral module space, see the Bus State Controller section of the hardware manual for the relevant product.
### 3.3.2 Address Error Exception Handling

When an address error occurs, address error exception handling is started after the end of the bus cycle in which the address error occurred and completion of the currently executing instruction. CPU operations are as follows.

1. The start address of the exception service routine corresponding to the address error is fetched from the exception handling vector table.
2. The status register (SR) is saved on the stack.
3. The program counter (PC) is saved on the stack. The saved PC value is the start address of the instruction following the last instruction executed.
4. Execution jumps to the address fetched from the exception handling vector table and program execution commences. The jump is not a delayed branch.

### 3.4 RAM Errors

#### 3.4.1 RAM Error Sources

A RAM error occurs in the event of a software error in an on-chip RAM read access. For details, see “RAM Errors” in the Exception Handling section of the hardware manual for the relevant product.

#### 3.4.2 RAM Error Exception Handling

When a RAM error occurs, RAM error exception handling is started after the end of the bus cycle in which the error occurred and completion of the currently executing instruction. CPU operations are as follows.

1. The start address of the exception service routine corresponding to the RAM error is fetched from the exception handling vector table.
2. The status register (SR) is saved on the stack.
3. The program counter (PC) is saved on the stack. The saved PC value is the start address of the instruction following the last instruction executed.
4. Execution jumps to the address fetched from the exception handling vector table and program execution commences. The jump is not a delayed branch.
3.5 Register Bank Errors

3.5.1 Register Bank Error Sources

(1) Bank Overflow
   When a save has already been performed to all register bank areas when acceptance of register
   overflow exception has been set by interrupt controller, and an interrupt that uses a register
   bank is generated and is accepted by the CPU

(2) Bank Underflow
   When an attempt is made to execute a RESBANK instruction when a save has not been
   performed to a register bank

3.5.2 Register Bank Error Exception Handling

Register bank error exception handling is started when a register bank error occurs. CPU
operations are as follows.

1. The start address of the exception service routine corresponding to the register bank error is
   fetched from the exception handling vector table.
2. The status register (SR) is saved on the stack.
3. The program counter (PC) is saved on the stack. The saved PC value is the start address of the
   instruction following the last instruction executed, in the case of a bank overflow, or the start
   address of the executed RESBANK instruction, in the case of an underflow. To prevent
   multiple interrupts when a bank overflow occurs, the level of the interrupt that is the source of
   the bank overflow is written to the interrupt mask level bits (I3 to I0) in the status register
   (SR).
4. Execution jumps to the address fetched from the exception handling vector table and program
   execution commences. The jump is not a delayed branch.
3.6 Interrupts

3.6.1 Interrupt Sources

Interrupt exception handling can be initiated by an NMI, a user break, the H-UDI, an external interrupt, or an on-chip peripheral module, as shown in table 3.6.

Table 3.6 Interrupt Sources

<table>
<thead>
<tr>
<th>Type</th>
<th>Request Source</th>
<th>Number of Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI</td>
<td>NMI pin (external input)</td>
<td>1</td>
</tr>
<tr>
<td>User break</td>
<td>User break controller</td>
<td>1</td>
</tr>
<tr>
<td>H-UDI</td>
<td>User debug interface</td>
<td>1</td>
</tr>
<tr>
<td>External interrupt</td>
<td>External interrupt pin, on-chip peripheral module</td>
<td>See Note</td>
</tr>
</tbody>
</table>

Each interrupt source is assigned a different vector number and vector table offset. For details of vector numbers and vector table address offsets, see “Interrupt Exception Vectors and Priority” in the Interrupt Controller section of the hardware manual for the relevant product.

Note: For details and numbers of external interrupts (IRQ) and on-chip peripheral module request sources, see “Interrupt Sources” in the Interrupt Controller section of the hardware manual for the relevant product.

3.6.2 Interrupt Priority

Interrupt sources are assigned priority levels. If a number of interrupts occur simultaneously (multiple interruption), the priority order is determined by the interrupt controller (INTC) and exception handling is initiated accordingly.

Interrupt source priority levels are expressed as values from 0 to 16, with 0 representing the lowest priority level and 16 the highest. The NMI interrupt is the highest-priority interrupt at level 16; it cannot be masked and is always accepted. The user break interrupt and H-UDI are assigned priority level 15. The priority level of IRQ interrupts and on-chip peripheral module interrupts can be set as desired in the interrupt priority level setting registers of the INTC (see table 3.7). Priority levels 0 to 15, but not 16, can be set. For details of the interrupt priority level setting registers, see the Interrupt Controller section of the hardware manual for the relevant product.
### Table 3.7  Interrupt Priority Levels

<table>
<thead>
<tr>
<th>Type</th>
<th>Priority Level</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI</td>
<td>16</td>
<td>Fixed priority level, not maskable</td>
</tr>
<tr>
<td>User break</td>
<td>15</td>
<td>Fixed priority level</td>
</tr>
<tr>
<td>H-UDI</td>
<td>15</td>
<td>Fixed priority level</td>
</tr>
<tr>
<td>External interrupt (IRQ), on-chip module</td>
<td>0 to 15</td>
<td>Can be set in interrupt priority level setting register</td>
</tr>
</tbody>
</table>

#### 3.6.3 Interrupt Exception Handling

When an interrupt occurs, its priority is determined by the interrupt controller (INTC). NMI is always accepted, but other interrupts are only accepted if their priority level is higher than the priority level set in the interrupt mask bits (I3 to I0) in the status register (SR).

When an interrupt is accepted, interrupt exception handling is started. In interrupt exception handling, the CPU saves SR and the program counter (PC) on the stack. In interrupt exception handling other than NMI, UBC, when register bank use has been set, general registers R0 to R14, control register GBR, system registers MACH, MACL, and PR, and the vector table address offset of the interrupt exception handling to be executed, are saved to the register bank. In the case of exception handling due to an address error, RAM error, register bank error, NMI interrupt, UBC interrupt, or instruction, saving to a register bank is not performed. Also, when saving is performed to all register banks, automatic saving to the stack is performed instead of register bank saving. In this case, an interrupt controller setting must have been made for register bank overflow exceptions not to be accepted. If a setting has been made for register bank overflow exceptions to be accepted, a register bank overflow exception will be generated. The interrupt priority level of the accepted interrupt is then written to bits I3 to I0 in SR. In the case of NMI, however, although its priority level is 16, H’F (level 15) is written to bits I3 to I0. Next, the CPU fetches the exception service routine start address from the exception vector table entry corresponding to the accepted interrupt, jumps to that address, and starts executing the exception service routine. For details of interrupt exception handling, see “Operation” in the Interrupt Controller section of the hardware manual for the relevant product.
### 3.7 Instruction Exceptions

#### 3.7.1 Types of Instruction Exception

There are five kinds of instruction that can initiate exception handling: the TRAP instruction, slot illegal instructions, general illegal instructions, integer division instructions, and floating-point operation instructions. These are summarized in table 3.8.

<table>
<thead>
<tr>
<th>Type</th>
<th>Source Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trap instruction</strong></td>
<td>TRAPA</td>
<td>Delayed branch instructions: JMP, JSR, BRA, BSR, RTS, RTE, BF/S, BT/S, BSRF, BRAF</td>
</tr>
<tr>
<td><strong>Slot illegal instruction</strong></td>
<td>Undefined code (FPU instruction or FPU-related CPU instruction in module standby status including FPU or in product with no FPU, or register bank-related instruction in product with no register bank) located immediately after delayed branch instruction (in delay slot), instruction that modifies PC, 32-bit instruction, RESBANK instruction, DIVS instruction, or DIVU instruction</td>
<td>Register bank-related instructions: RESBANK, LDBANK, STBANK</td>
</tr>
<tr>
<td><strong>General illegal instruction</strong></td>
<td>Undefined code (FPU instruction, FPU-related CPU instruction, or register bank-related instruction in module standby status including FPU or in product with no FPU) not in delay slot</td>
<td>Instructions that modify PC: JMP, JSR, BRA, BSR, RTS, RTE, BT, BF, TRAPA, BF/S, BT/S, BSRF, BRAF, JSR/N, RTV/N</td>
</tr>
<tr>
<td><strong>Integer division exception</strong></td>
<td>Division by zero</td>
<td>DIVU, DIVS</td>
</tr>
<tr>
<td><strong>Floating-point operation instruction</strong></td>
<td>Negative maximum value ÷ (-1)</td>
<td>DIVS</td>
</tr>
<tr>
<td></td>
<td>Instruction causing invalid operation defined by IEEE754 standard or division-by-zero exception, instruction causing overflow, underflow, or inexact exception</td>
<td>FADD, FSUB, FMUL, FDIV, FMAC, FCMP/EQ, FCMP/GT, FLOAT, FTRC, FCNVDS, FCNVSD, FSQRT</td>
</tr>
</tbody>
</table>
3.7.2 Trap Instruction

When a TRAPA instruction is executed, trap instruction exception handling is started. The CPU operates as follows.

1. The start address of the exception service routine corresponding to the vector number specified by the TRAPA instruction is fetched from the exception handling vector table.
2. The status register (SR) is saved on the stack.
3. The program counter (PC) is saved on the stack. The saved PC value is the start address of the instruction following the TRAPA instruction.
4. Execution jumps to the address fetched from the exception handling vector table and program execution commences. The jump is not a delayed branch.

3.7.3 Slot Illegal Instructions

An instruction located immediately after a delayed branch instruction is said to be located in the delay slot. If the instruction in the delay slot is undefined code, slot illegal instruction exception handling is started when that undefined code is decoded. Also, if the instruction in the delay slot is one that modifies the program counter (PC), slot illegal instruction exception handling is started when that instruction is decoded. Moreover, in the case of a product that does not have an FPU, or if the FPU is in the module standby state, a floating-point instruction or FPU-related instruction is treated as undefined code, and if located in a delay slot, will cause slot illegal instruction exception handling to be started when decoded. In addition, if the product that does not have a register bank, register bank-related instructions are treated as undefined code. If located in a delay slot, when decoded they will cause slot illegal instruction handling to be started.

Furthermore, if an instruction located in a delay slot is a 32-bit instruction, RESBANK instruction, DIVS instruction, or DIVU instruction, slot illegal instruction exception handling will be started when this instruction is decoded.

CPU operations in slot illegal instruction exception handling are as follows.

1. The start address of the exception service routine is fetched from the exception handling vector table.
2. The status register (SR) is saved on the stack.
3. The program counter (PC) is saved on the stack. The saved PC value is the jump destination address of the delayed branch instruction immediately preceding an undefined code, instruction that overwrites the PC, 32-bit instruction, RESBANK instruction, DIVS instruction, or DIVU instruction.
4. Execution jumps to the address fetched from the exception handling vector table and program execution commences. The jump is not a delayed branch.
3.7.4 General Illegal Instructions

When undefined code located other than immediately after a delayed branch instruction (in a delay slot) is decoded, general illegal instruction exception handling is started. Also, in the case of a product that does not have an FPU, or if the FPU is in the module standby state, a floating-point instruction or FPU-related instruction is treated as undefined code, and if located other than immediately after a delayed branch instruction (in a delay slot), will cause general illegal instruction exception handling to be started when decoded. In addition, if the product that does not have a register bank, register bank-related instructions are treated as undefined code. If not located immediately after a delayed branch instruction (in a delay slot), when decoded they will cause slot illegal instruction handling to be started.

The CPU follows the same procedure as in the case of slot illegal instruction exception handling, except that the PC value saved is the start address of the undefined code.

3.7.5 Integer Division Instructions

An integer division exception is generated if an integer division instruction executes division by zero, or if the result of integer division overflows. Instructions that may cause a division-by-zero exception are DIVU and DIVS. The only instruction that may cause an overflow exception is DIVS, the exception being generated if the negative maximum value is divided by –1. CPU operations in integer division exception handling are as follows.

1. The start address of the exception service routine corresponding to the integer division exception is fetched from the exception handling vector table.
2. The status register (SR) is saved on the stack.
3. The program counter (PC) is saved on the stack. The saved PC value is the start address of the integer division instruction that generated the exception.
4. Execution jumps to the address fetched from the exception handling vector table and program execution commences. The jump is not a delayed branch.

3.7.6 Floating-Point Operation Instructions

An FPU exception is generated when the V, Z, O, U, or I bit in the enable field of the FPSCR register is set. This indicates the occurrence of an invalid operation exception defined by the IEEE754 standard, a division-by-zero exception, overflow (in the case of an instruction for which this is possible), underflow (in the case of an instruction for which this is possible), or an imprecision exception (in the case of an instruction for which this is possible).

Floating-point operation instructions that may cause an exception are as follows.
FADD, FSUB, FMUL, FDIV, FMAC, FCMP/EQ, FCMP/GT, FLOAD, FTRC, FCNVDS, FCNVSD, FSQRT

An FPU exception is generated only when the corresponding enable bit is set. When the FPU detects an exception, FPU operation is halted and exception generation is reported to the CPU. When exception handling is started, CPU operations are as follows.

1. The start address of the exception service routine stored in VBR + H'00000034 is fetched from the exception handling vector table.
2. SR contents are saved on the stack.
3. PC is saved on the stack. The PC value saved is the start address of the instruction following the last instruction executed.
4. Control branches to the address stored in VBR + H'00000034.

The exception flag bits in FPSCR are always updated regardless of whether or not an FPU exception has been accepted, and remain set until explicitly cleared by the user by means of an instruction. The FPSCR source bits change each time an FPU instruction is executed.

When the V bit in the enable field of the FPSCR register is set and the QIS bit in FPSCR is also set, FPU exception handling is started when qNaN or ±∞ is input to a floating-point operation instruction source.

### 3.8 Cases in Which Exceptions Are Not Accepted

There are cases, as shown in table 3.9, in which, if an address error, RAM error, FPU exception, register bank error (overflow), or interrupt occurs immediately after a delayed branch instruction, the exception is not accepted immediately, but is held pending. In such cases, the exception will be accepted when an instruction for which exception acceptance is permitted is decoded.

#### Table 3.9 Exception Source Occurrence Immediately after Delayed Branch Instruction

<table>
<thead>
<tr>
<th>Point of Occurrence</th>
<th>Address Error</th>
<th>RAM Error</th>
<th>FPU Exception</th>
<th>Register Bank Error (Overflow)</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediately after a delayed branch instruction*</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Notes: ×: Not accepted

* Delayed branch instructions: JMP, JSR, BRA, BSR, RTS, RTE, BF/S, BT/S, BSRF, BRAF
### 3.9 Stack Status after Exception Handling

Table 3.10 shows the stack status after completion of exception handling.

#### Table 3.10 Stack Status after Exception Handling

<table>
<thead>
<tr>
<th>Type</th>
<th>Stack Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address error</td>
<td>Address of instruction following executed instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>RAM error</td>
<td>Address of instruction following executed instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>Register bank error</td>
<td>Start address of relevant RESBANK instruction</td>
</tr>
<tr>
<td>(underflow)</td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>Trap instruction</td>
<td>Address of instruction following TRAPA instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>General illegal instruction</td>
<td>Start address of general illegal instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Address of instruction following executed instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>Register bank error</td>
<td>Address of instruction following executed instruction</td>
</tr>
<tr>
<td>(overflow)</td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>Integer division instruction</td>
<td>Start address of relevant integer division instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>Slot illegal instruction</td>
<td>Jump destination address of delayed branch instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
<tr>
<td>FPU exception</td>
<td>Address of instruction following executed instruction</td>
</tr>
<tr>
<td></td>
<td>SR (32 bits)</td>
</tr>
<tr>
<td></td>
<td>SP</td>
</tr>
</tbody>
</table>
3.10 Usage Notes

3.10.1 Stack Pointer (SP) Value

Ensure that the stack pointer (SP) value is a multiple of 4. If it is not, an address error will be caused when the stack is accessed in exception handling.

3.10.2 Vector Base Register (VBR) Value

Ensure that the vector base register (VBR) value is a multiple of 4. If it is not, an address error will be caused when the vector is accessed in exception handling.

3.10.3 Address Errors Occurring in Address Error Exception Handling Stacking

If the stack pointer (SP) value is not a multiple of 4, an address error will occur in exception handling (interrupt, etc.) stacking, and after the exception handling is completed, address error exception handling will be started. An address error will also occur in stacking in the address error exception handling, but this address error will not be accepted in order to prevent endless stacking due to address errors. This enables program control to be switched to the address error exception service routine, and error handling to be carried out.

When an address error occurs in exception handling stacking, the stacking bus cycle (write) is executed. In SR and PC stacking, SP is decremented by 4 in each case, and therefore the SP value is not a multiple of 4 after stacking is completed. Also, the address value output in stacking is the SP value, and the actual address at which the error occurred is output. In this case, the stacked write data is undefined.
Section 4  Instruction Features

4.1  RISC-Type Instruction Set

All instructions are RISC type. Their features are detailed in this section.

(1)  16-Bit Fixed-Length Instructions

Basic instructions have a fixed length of 16 bits, increasing program code efficiency.

(2)  Addition of 32-Bit Fixed-Length Instructions

The SH-2A/SH2A-FPU features the addition of 32-bit fixed-length instructions, improving performance and ease of use.

(3)  One Instruction/Cycle

Basic instructions can be executed in one cycle using the pipeline system.

(4)  Data Length

Longword is the standard data length for all operations. Memory can be accessed in bytes, words, or longwords. Byte or word data accessed from memory is sign-extended and calculated with longword data. Immediate data is sign-extended for arithmetic operations or zero-extended for logic operations. It also is calculated with longword data.

Table 4.1  Sign Extension of Word Data

<table>
<thead>
<tr>
<th>SH-2A/SH2A-FPU CPU</th>
<th>Description</th>
<th>Example for Other CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.W @(disp,PC),R1</td>
<td>Data is sign-extended to 32 bits, and R1 becomes H'00001234. It is next operated upon by an ADD instruction.</td>
<td>ADD.W #H'1234,R0</td>
</tr>
<tr>
<td>ADD     R1,R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..........</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.DATA.W H'1234</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The address of the immediate data is accessed by @(disp, PC).

(5)  Load-Store Architecture

Basic operations are executed between registers. For operations that involve memory access, data is loaded to the registers and executed (load-store architecture). Instructions such as AND that manipulate bits, however, are executed directly in memory.
(6) Delayed Branching

With the exception of some instructions, unconditional branch instructions, etc., are executed as delayed branches. With a delayed branch instruction, the branch is made after execution of the instruction immediately following the delayed branch instruction. This reduces disruption of the pipeline when a branch is made.

In a delayed branch, the actual branch operation occurs after execution of the slot instruction. However, instruction execution for register updating, etc., excluding the branch operation, is performed in delayed branch instruction → delay slot instruction order. For example, even though the contents of the register holding the branch destination address are changed in the delay slot, the branch destination address remains as the register contents prior to the change.

Table 4.2  Delayed Branch Instructions

<table>
<thead>
<tr>
<th>SH-2A/SH2A-FPU CPU</th>
<th>Description</th>
<th>Example of Other CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA TRGET</td>
<td>ADD is executed before branch to TRGET.</td>
<td>ADD.W R1,R0</td>
</tr>
<tr>
<td>ADD R1,R0</td>
<td></td>
<td>BRA TRGET</td>
</tr>
</tbody>
</table>

(7) Addition of Unconditional Branch Instructions with No Delay Slot

The SH-2A/SH2A-FPU features the addition of unconditional branch instructions in which a delay slot instruction is not executed. This makes it possible to cut down on the number of unnecessary NOP instructions, and so reduce the code size.

(8) Multiplication/Accumulation Operation

16bit × 16bit → 32-bit multiplication operations are executed in one to two cycles. 16bit × 16bit + 64bit → 64-bit multiplication/accumulation operations are executed in two to three cycles. 32bit × 32bit → 64-bit multiplication and 32bit × 32bit + 64bit → 64-bit multiplication/accumulation operations are executed in two to four cycles.

(9) T Bit

The T bit in the status register changes according to the result of the comparison, and in turn is the condition (true/false) that determines if the program will branch. The number of instructions after T bit in the status register is kept to a minimum to improve the processing speed.
### Table 4.3 T Bit

<table>
<thead>
<tr>
<th>SH-2A/SH2A-FPU CPU</th>
<th>Description</th>
<th>Example for Other CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP/GE R1,R0</td>
<td>T bit is set when R0 ≥ R1. The program branches to TRGET0 when R0 ≥ R1 and to TRGET1 when R0 &lt; R1.</td>
<td>CMP.W R1,R0</td>
</tr>
<tr>
<td>BT TRGET0</td>
<td></td>
<td>BGE TRGET0</td>
</tr>
<tr>
<td>BF TRGET1</td>
<td></td>
<td>BLT TRGET1</td>
</tr>
<tr>
<td>ADD #–1,R0</td>
<td>T bit is not changed by ADD. T bit is set when R0 = 0. The program branches if R0 = 0.</td>
<td>SUB.W #1,R0</td>
</tr>
<tr>
<td>CMP/EQ #0,R0</td>
<td></td>
<td>BEQ TRGET</td>
</tr>
<tr>
<td>BT TRGET</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(10) Immediate Data

Byte immediate data is located in instruction code. Word or longword immediate data is not input via instruction codes but is stored in a memory table. The memory table is accessed by an immediate data transfer instruction (MOV) using the PC relative addressing mode with displacement.

With the SH-2A/SH2A-FPU, immediate data of 17 to 28 bits can be located in an instruction code. However, for immediate data of 21 to 28 bits, an OR instruction must be executed after a register transfer.

### Table 4.4 Referencing by Means of Immediate Data

<table>
<thead>
<tr>
<th>Type</th>
<th>SH-2A/SH2A-FPU CPU</th>
<th>Example for Other CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit immediate</td>
<td>MOV #H'12,R0</td>
<td>MOV.B #H'12,R0</td>
</tr>
<tr>
<td>16-bit immediate</td>
<td>MOVI20 #H'1234, R0</td>
<td>MOV.W #H'1234,R0</td>
</tr>
<tr>
<td>20-bit immediate</td>
<td>MOVI20 #H'12345, R0</td>
<td>MOV.L #H'12345,R0</td>
</tr>
<tr>
<td>28-bit immediate</td>
<td>MOVI20S #H'12345, R0</td>
<td>MOV.L #H'1234567,R0</td>
</tr>
<tr>
<td>32-bit immediate</td>
<td>MOV.L @(disp,PC),R0</td>
<td>MOV.L #H'12345678,R0</td>
</tr>
</tbody>
</table>

Note: Immediate data is referenced by @(disp,PC).
(11) Absolute Address

When data is accessed by absolute address, the value already in the absolute address is placed in
the memory table. Loading the immediate data when the instruction is executed transfers that
value to the register and the data is accessed in the indirect register addressing mode.

With the SH-2A/SH2A-FPU, when data is referenced using an absolute address not exceeding 28
bits, it is also possible to transfer immediate data located in the instruction code to a register, and
reference the data using register indirect addressing mode. However, when referencing data using
an absolute address of 21 to 28 bits, an OR instruction must be used after the register transfer.

Table 4.5 Referencing by Means of Absolute Address

<table>
<thead>
<tr>
<th>Type</th>
<th>SH-2A/SH2A-FPU CPU</th>
<th>Example for Other CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 20 bits</td>
<td>MOVI20 #H'12345, R1</td>
<td>MOV.B @H'12345,R0</td>
</tr>
<tr>
<td></td>
<td>MOV.B @R1, R0</td>
<td></td>
</tr>
<tr>
<td>21 to 28 bits</td>
<td>MOVI20S #H'12345, R1</td>
<td>MOV.B @H'1234567,R0</td>
</tr>
<tr>
<td></td>
<td>OR #H'67, R1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV.B @R1, R0</td>
<td></td>
</tr>
<tr>
<td>29 bits or more</td>
<td>MOV.L @(disp,PC),R1</td>
<td>MOV.B @H'12345678,R0</td>
</tr>
<tr>
<td></td>
<td>MOV.B @R1,R0</td>
<td></td>
</tr>
<tr>
<td>. . . . . . .</td>
<td>.DATA.L H'12345678</td>
<td></td>
</tr>
</tbody>
</table>

(12) 16-Bit/32-Bit Displacement

When data is accessed by 16-bit or 32-bit displacement, the pre-existing displacement value is
placed in the memory table. Loading the immediate data when the instruction is executed transfers
that value to the register and the data is accessed in the indirect indexed register addressing mode.

Table 4.6 Displacement Accessing

<table>
<thead>
<tr>
<th>Type</th>
<th>SH-2A/SH2A-FPU CPU</th>
<th>Example for Other CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit displacement</td>
<td>MOV.W @(disp,PC),R0</td>
<td>MOV.W @(H'1234,R1),R2</td>
</tr>
<tr>
<td></td>
<td>MOV.W @(R0,R1),R2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>. . . . . . .</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.DATA.W H'1234</td>
<td></td>
</tr>
</tbody>
</table>
### 4.2 Addressing Modes

Addressing modes effective address calculation by the CPU core are described below.

**Table 4.7 Addressing Modes and Effective Addresses**

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Instruction Format</th>
<th>Effective Addresses Calculation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct register addressing</td>
<td>Rn</td>
<td>The effective address is register Rn. (The operand is the contents of register Rn.)</td>
<td>—</td>
</tr>
<tr>
<td>Indirect register addressing</td>
<td>@Rn</td>
<td>The effective address is the content of register Rn.</td>
<td>Rn</td>
</tr>
<tr>
<td>Post-increment indirect register addressing</td>
<td>@Rn +</td>
<td>The effective address is the content of register Rn. A constant is added to the content of Rn after the instruction is executed. 1 is added for a byte operation, 2 for a word operation, or 4 for a longword operation.</td>
<td>Rn (After the instruction is executed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte: Rn + 1 → Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Word: Rn + 2 → Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Longword: Rn + 4 → Rn</td>
</tr>
<tr>
<td>Pre-decrement indirect register addressing</td>
<td>@–Rn</td>
<td>The effective address is the value obtained by subtracting a constant from Rn. 1 is subtracted for a byte operation, 2 for a word operation, or 4 for a longword operation.</td>
<td>Rn (Instruction executed with Rn after calculation)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte: Rn – 1 → Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Word: Rn – 2 → Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Longword: Rn – 4 → Rn</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>Instruction Format</td>
<td>Effective Addresses Calculation</td>
<td>Formula</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------</td>
<td>--------------------------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| Indirect register addressing with displacement | @(disp:4, Rn) | The effective address is Rn plus a 4-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, or is quadrupled for a longword operation. | Byte: Rn + disp  
Word: Rn + disp × 2  
Longword: Rn + disp × 4 |
| Indirect indexed register addressing | @(R0, Rn) | The effective address is the Rn value plus R0. | Rn + R0 |
| Indirect GBR addressing with displacement | @(disp:8, GBR) | The effective address is the GBR value plus an 8-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, or is quadrupled for a longword operation. | Byte: GBR + disp  
Word: GBR + disp × 2  
Longword: GBR + disp × 4 |
<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Instruction Format</th>
<th>Effective Addresses Calculation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect indexed GBR addressing</td>
<td>@((R0, GBR)</td>
<td>The effective address is the GBR value plus R0.</td>
<td>GBR + R0</td>
</tr>
<tr>
<td>TBR duplicate indirect with displacement</td>
<td>@@(disp:8, TBR)</td>
<td>Effective address is register TBR contents with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 4.</td>
<td>(TBR + disp x 4) address contents</td>
</tr>
<tr>
<td>PC relative addressing with displacement</td>
<td>@(disp:8, PC)</td>
<td>The effective address is the PC value plus an 8-bit displacement (disp). The value of disp is zero-extended, and disp is doubled for a word operation, or is quadrupled for a longword operation. For a longword operation, the lowest two bits of the PC are masked.</td>
<td>Word: PC + disp x 2 Longword: PC &amp; H'FFFFFFFC + disp x 4</td>
</tr>
</tbody>
</table>

\[
\text{GBR} + \text{R0}
\]

\[
\text{TBR} + \text{disp} \times 4
\]

\[
\text{PC} + \text{disp} \times 2 \text{ or } \text{PC} \& \text{H'FFFFFFFC} + \text{disp} \times 4
\]
### Addressing Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Instruction Format</th>
<th>Effective Addresses Calculation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC relative addressing</td>
<td>disp:8</td>
<td>The effective address is the PC value sign-extended with an 8-bit displacement (disp), doubled, and added to the PC.</td>
<td>PC + disp × 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image1.png" alt="Diagram" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td>disp:12</td>
<td>The effective address is the PC value sign-extended with a 12-bit displacement (disp), doubled, and added to the PC.</td>
<td>PC + disp × 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td>The effective address is the register PC plus Rn.</td>
<td>PC + Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td></td>
</tr>
<tr>
<td>Immediate addressing</td>
<td>#imm:20</td>
<td>20-bit immediate data imm of MOVI20 instruction is sign-extended.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image4.png" alt="Diagram" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>20-bit immediate data imm of MOVI20S instruction is left-shifted 8 bits, upper part is sign-extended, and lower part is zero-padded.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image5.png" alt="Diagram" /></td>
<td></td>
</tr>
</tbody>
</table>
### Addressing Mode

<table>
<thead>
<tr>
<th>Instruction Format</th>
<th>Effective Addresses Calculation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate addressing</td>
<td>#imm:8 The 8-bit immediate data (imm) for the TST, AND, OR, and XOR instructions are zero-extended.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>#imm:8 The 8-bit immediate data (imm) for the MOV, ADD, and CMP/EQ instructions are sign-extended.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>#imm:8 Immediate data (imm) for the TRAPA instruction is zero-extended and is quadrupled.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>#imm:3 3-bit immediate data imm of BAND, BOR, BXOR, BST, BLD, BSET, or BCLR instruction indicates bit position.</td>
<td>—</td>
</tr>
</tbody>
</table>

### 4.3 Instruction Format

The instruction format table, table 5.8, refers to the source operand and the destination operand. The meaning of the operand depends on the instruction code. The symbols are used as follows:

- xxxx: Instruction code
- mmmm: Source register
- nnnn: Destination register
- iiii: Immediate data
- dddd: Displacement
### Table 4.8 Instruction Formats

<table>
<thead>
<tr>
<th>Instruction Formats</th>
<th>Source Operand</th>
<th>Destination Operand</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>xxxx xxxx xxxx xxxx 0</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>n format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>xxxx nnnn xxxx xxxx 0</td>
<td>0</td>
<td>MOV T Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nnnn: Register direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control register or system register</td>
<td>STS MACH,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nnnn: Register direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R0 (register direct)</td>
<td>DIVU R0, Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nnnn: Register direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control register or system register</td>
<td>STC.L SR,@-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nnnn: Register indirect with pre-decrement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmmmm: Register direct</td>
<td>MOVU.L Rm,@-R15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R15 (register indirect with pre-decrement)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R15 (register indirect with post-increment)</td>
<td>MOVU.L @R15+,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nnnn: Register direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R0 (register direct)</td>
<td>MOV.L R0,@Rn+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nnnn: Register indirect with post-increment</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>xxxx mmmmm xxxx xxxx 0</td>
<td>0</td>
<td>LDC Rm,SR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmmmm: Control register or system register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmmmm: Register indirect with post-increment</td>
<td>LDC.L @Rm+,SR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control register or system register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmmmm: Register indirect</td>
<td>JMP @Rm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmmmm: R0 (register direct)</td>
<td>MOV.L @-Rm, R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register indirect with pre-decrement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mmmmm: PC-relative using Rm</td>
<td>BRAF Rm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Formats</td>
<td>Source Operand</td>
<td>Destination Operand</td>
<td>Example</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>nm format</td>
<td>mmm: Direct register</td>
<td>nnn: Direct register</td>
<td>ADD Rm,Rn</td>
</tr>
<tr>
<td></td>
<td>mmm: Direct register</td>
<td>nnn: Indirect register</td>
<td>MOV.L Rm,@Rn</td>
</tr>
<tr>
<td></td>
<td>mmm: Indirect post-increment register (multiply/accumulate)</td>
<td>MACH, MACL</td>
<td>MAC.W @Rm+,@Rn+</td>
</tr>
<tr>
<td></td>
<td>mmm: Indirect post-increment register</td>
<td>nnn: Direct register</td>
<td>MOV.L @Rm+,Rn</td>
</tr>
<tr>
<td></td>
<td>mmm: Direct register</td>
<td>nnn: Indirect pre-decrement register</td>
<td>MOV.L Rm,@-Rn</td>
</tr>
<tr>
<td></td>
<td>mmm: Direct register</td>
<td>nnn: Indirect indexed register</td>
<td>MOV.L Rm,@(R0,Rn)</td>
</tr>
<tr>
<td>md format</td>
<td>mmmdddd: Indirect register with displacement</td>
<td>R0 (Direct register)</td>
<td>MOV.B @disp,Rm,R0</td>
</tr>
<tr>
<td></td>
<td>R0 (Direct register)</td>
<td>nnndddd: Indirect register with displacement</td>
<td>MOV.B R0,@disp,Rn</td>
</tr>
<tr>
<td>nd4 format</td>
<td>mmm: Direct register</td>
<td>nnndddd: Indirect register with displacement</td>
<td>MOV.L Rm,@disp,Rn</td>
</tr>
<tr>
<td></td>
<td>mmmdddd: Indirect register with displacement</td>
<td>nnn: Direct register</td>
<td>MOV.L @disp,Rm,Rn</td>
</tr>
</tbody>
</table>
## Instruction Formats

<table>
<thead>
<tr>
<th>Source Operand</th>
<th>Destination Operand</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmm: Register direct</td>
<td>nnnn: Register</td>
<td>MOV.L Rm,@(disp12, Rn)</td>
</tr>
<tr>
<td>mmmmm: Register indirect with displacement</td>
<td>nnn: Register direct</td>
<td>MOV.L @(disp12,Rm), Rn</td>
</tr>
<tr>
<td>dddd: GBR indirect with displacement</td>
<td>R0 (register direct)</td>
<td>MOV.L @(disp,GBR),R0</td>
</tr>
<tr>
<td>ddddddddd: GBR indirect with displacement</td>
<td>R0 (register direct)</td>
<td>MOV.L @(disp,GBR)</td>
</tr>
<tr>
<td>ddddddddd: PC-relative with displacement</td>
<td>R0 (register direct)</td>
<td>MOVA @(disp,PC),R0</td>
</tr>
<tr>
<td>ddddddddd: TBR duplicate indirect with displacement</td>
<td>—</td>
<td>JSR/N @@(disp8,TBR)</td>
</tr>
<tr>
<td>ddddddddd: PC-relative</td>
<td>—</td>
<td>BF label</td>
</tr>
<tr>
<td>ddddddddddd: PC relative</td>
<td>—</td>
<td>BRA label (label = disp + PC)</td>
</tr>
<tr>
<td>ddddddddd: PC relative</td>
<td>nnnn: Direct register</td>
<td>MOV.L @(disp,PC),Rn</td>
</tr>
<tr>
<td>iiiiiii: Immediate</td>
<td>Indirect indexed GBR</td>
<td>AND.B #imm,@(R0,GBR)</td>
</tr>
<tr>
<td>iiii: Immediate</td>
<td>R0 (Direct register)</td>
<td>AND #imm,R0</td>
</tr>
<tr>
<td>iiii: Immediate</td>
<td>—</td>
<td>TRAPA #imm</td>
</tr>
<tr>
<td>iiiiiii: Immediate</td>
<td>nnnn: Direct register</td>
<td>ADD #imm,Rn</td>
</tr>
</tbody>
</table>

---

Rev. 3.00  Jul 08, 2005  page 44 of 484
REJ09B0051-0300

RENESAS
### Instruction Formats

<table>
<thead>
<tr>
<th>ni3 format</th>
<th>Source Operand</th>
<th>Destination Operand</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnnn: Register direct</td>
<td>—</td>
<td>nnnn: Register direct</td>
<td>BLD #imm3,Rn</td>
</tr>
<tr>
<td>iii: Immediate</td>
<td>—</td>
<td>iii: Immediate</td>
<td>BST #imm3,Rn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ni20 format</th>
<th>Source Operand</th>
<th>Destination Operand</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnnn: Register direct</td>
<td>nnnn: Register direct</td>
<td>MOVI20 #imm20,Rn</td>
<td></td>
</tr>
<tr>
<td>iii: Immediate</td>
<td>—</td>
<td>iii: Immediate</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nid format</th>
<th>Source Operand</th>
<th>Destination Operand</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnnn: Register indirect with displacement</td>
<td>nnnn: Register indirect with displacement</td>
<td>BLD.B #imm3,@(disp12,Rn)</td>
<td></td>
</tr>
<tr>
<td>iii: Immediate</td>
<td>—</td>
<td>iii: Immediate</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** * In multiply/accumulate instructions, nnnn is the source register.
## 5.1 Instruction Set by Classification

Table 5.1 shows instruction by classification.

**Table 5.1 Classification of Instruction**

<table>
<thead>
<tr>
<th>Classification</th>
<th>Instruction Type</th>
<th>Op Code</th>
<th>Function</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transfer instructions</td>
<td>13</td>
<td>MOV</td>
<td>Data transfer</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Immediate data transfer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peripheral module data transfer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Structure data transfer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reverse stack transfer</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td>Execution address transfer</td>
<td></td>
</tr>
<tr>
<td>MOV120</td>
<td></td>
<td>MOVI20</td>
<td>20-bit immediate data transfer</td>
<td></td>
</tr>
<tr>
<td>MOV120S</td>
<td></td>
<td>MOVI20S</td>
<td>20-bit immediate data transfer, 8-bit left-shift</td>
<td></td>
</tr>
<tr>
<td>MOVML</td>
<td></td>
<td></td>
<td>R0-Rn register save/restore</td>
<td></td>
</tr>
<tr>
<td>MOVMU</td>
<td></td>
<td></td>
<td>Rn-R14, PR register save/restore</td>
<td></td>
</tr>
<tr>
<td>MOVRT</td>
<td></td>
<td></td>
<td>T bit inversion and transfer to Rn</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td>T bit transfer</td>
<td></td>
</tr>
<tr>
<td>MOVU</td>
<td></td>
<td></td>
<td>Unsigned data transfer</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td></td>
<td></td>
<td>T bit inversion</td>
<td></td>
</tr>
<tr>
<td>PREF</td>
<td></td>
<td></td>
<td>Prefetch to operand cache</td>
<td></td>
</tr>
<tr>
<td>SWAP</td>
<td></td>
<td></td>
<td>Upper/lower swap</td>
<td></td>
</tr>
<tr>
<td>XTRCT</td>
<td></td>
<td></td>
<td>Extraction of middle of linked registers</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Set

<table>
<thead>
<tr>
<th>Classification</th>
<th>Instruction Type</th>
<th>Op Code</th>
<th>Function</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operation instructions</td>
<td>26</td>
<td>ADD</td>
<td>Binary addition</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDC</td>
<td>Binary addition with carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDV</td>
<td>Binary addition with overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CMP/cond</td>
<td>Comparison</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLIPS</td>
<td>Signed saturation value comparison</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLIPU</td>
<td>Unsigned saturation value comparison</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIVS</td>
<td>Signed division (32 ÷ 32)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIVU</td>
<td>Unsigned division (32 ÷ 32)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIV1</td>
<td>1-step division</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIV0S</td>
<td>Signed 1-step division initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIV0U</td>
<td>Unsigned 1-step division initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMULS</td>
<td>Signed double-precision multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMULU</td>
<td>Unsigned double-precision multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DT</td>
<td>Decrement and test</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTS</td>
<td>Sign extension</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTU</td>
<td>Zero extension</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC</td>
<td>Multiply and accumulate, double-precision multiply and accumulate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUL</td>
<td>Double-precision multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MULR</td>
<td>Rn result storage signed multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MULS</td>
<td>Signed multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MULU</td>
<td>Unsigned multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEG</td>
<td>Sign inversion</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEGC</td>
<td>Sign inversion with borrow</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUB</td>
<td>Binary subtraction</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUBC</td>
<td>Binary subtraction with borrow</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUBV</td>
<td>Binary subtraction with underflow</td>
<td></td>
</tr>
<tr>
<td>Classification</td>
<td>Instruction Type</td>
<td>Op Code</td>
<td>Function</td>
<td>Number of Instructions</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------------------</td>
<td>---------</td>
<td>---------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Logic operation instructions</td>
<td>6</td>
<td>AND</td>
<td>Logical AND</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOT</td>
<td>Bit inversion</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>Logical OR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAS</td>
<td>Memory test and bit setting</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TST</td>
<td>Logical AND T bit setting</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>XOR</td>
<td>Exclusive logical OR</td>
<td></td>
</tr>
<tr>
<td>Shift instructions</td>
<td>12</td>
<td>ROTL</td>
<td>1-bit left rotation</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROTR</td>
<td>1-bit right rotation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROTCL</td>
<td>1-bit left rotation with T bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROTCR</td>
<td>1-bit right rotation with T bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHAD</td>
<td>Dynamic arithmetic shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHAL</td>
<td>Arithmetic 1-bit left shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHAR</td>
<td>Arithmetic 1-bit right shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHLD</td>
<td>Dynamic logical shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHLL</td>
<td>Logical 1-bit left shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHLLn</td>
<td>Logical n-bit left shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHLR</td>
<td>Logical 1-bit right shift</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHLRn</td>
<td>Logical n-bit right shift</td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Set

<table>
<thead>
<tr>
<th>Classification</th>
<th>Instruction Type</th>
<th>Op Code</th>
<th>Function</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch instructions</td>
<td>10</td>
<td>BF</td>
<td>Conditional branch, delayed conditional branch (branches if ( T = 0 ))</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BT</td>
<td>Conditional branch, delayed conditional branch (branches if ( T = 1 ))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRA</td>
<td>Unconditional delayed branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRAF</td>
<td>Unconditional delayed branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSR</td>
<td>Delayed branch to subroutine procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSRF</td>
<td>Delayed branch to subroutine procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>JMP</td>
<td>Unconditional delayed branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>JSR</td>
<td>Branch to subroutine procedure, delayed branch to subroutine procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTS</td>
<td>Return from subroutine procedure, delayed return from subroutine procedure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTV/N</td>
<td>Return from subroutine procedure with ( Rm \rightarrow R0 ) transfer</td>
<td></td>
</tr>
<tr>
<td>System control instructions</td>
<td>14</td>
<td>CLRT</td>
<td>T bit clear</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLRMAC</td>
<td>MAC register clear</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDBANK</td>
<td>Register restoration from specified register bank entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDC</td>
<td>Load into control register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDS</td>
<td>Load into system register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESBANK</td>
<td>Register restoration from register bank</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTE</td>
<td>Return from exception handling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SETT</td>
<td>T bit setting</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SLEEP</td>
<td>Transition to power-down state</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STBANK</td>
<td>Register save to specified register bank entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STC</td>
<td>Store from control register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STS</td>
<td>Store from system register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TRAPA</td>
<td>Trap exception handling</td>
<td></td>
</tr>
<tr>
<td>Classification</td>
<td>Instruction Type</td>
<td>Op Code</td>
<td>Function</td>
<td>Number of Instructions</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------</td>
<td>---------</td>
<td>----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Floating-point instructions</td>
<td>19</td>
<td>FABS</td>
<td>Floating-point absolute value</td>
<td>48</td>
</tr>
<tr>
<td>Float</td>
<td>Floating-point addition</td>
<td>FADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point comparison</td>
<td>FCMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conversion from double-precision to single-precision</td>
<td>FCNVDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conversion from single-precision to double-precision</td>
<td>FCNVSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point division</td>
<td>FDIV</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point load immediate 0</td>
<td>FLDI0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point load immediate 1</td>
<td>FLDI1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point load into system register FPUL</td>
<td>FLDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conversion from integer to floating-point</td>
<td>FLOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point multiply and accumulate operation</td>
<td>FMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point data transfer</td>
<td>FMOV</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point multiplication</td>
<td>FMUL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point sign inversion</td>
<td>FNEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SZ bit inversion</td>
<td>FSCHG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point square root</td>
<td>FSQRT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point store from system register FPUL</td>
<td>FSTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating-point conversion with rounding to integer</td>
<td>FTRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load into floating-point system register</td>
<td>LDS</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store from floating-point system register</td>
<td>STS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Bit manipulation instructions

<table>
<thead>
<tr>
<th>Classification</th>
<th>Instruction Type</th>
<th>Op Code</th>
<th>Function</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit manipulation</td>
<td>10</td>
<td>BAND</td>
<td>Bit AND</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCLR</td>
<td>Bit clear</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLD</td>
<td>Bit load</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BOR</td>
<td>Bit OR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSET</td>
<td>Bit setting</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BST</td>
<td>Bit store</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BXOR</td>
<td>Bit exclusive OR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BANDNOT</td>
<td>Bit NOT AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BORNOT</td>
<td>Bit NOT OR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLDNOT</td>
<td>Bit NOT load</td>
<td></td>
</tr>
</tbody>
</table>

Total 112 253
Table 5.2 shows the format used in tables 5.3 to 5.8, which list instruction codes, operation, and execution states in order by classification.

**Table 5.2 Instruction Code Format**

<table>
<thead>
<tr>
<th>Item</th>
<th>Format</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Rm:</td>
<td>Source register</td>
</tr>
<tr>
<td></td>
<td>Rn:</td>
<td>Destination register</td>
</tr>
<tr>
<td></td>
<td>imm:</td>
<td>Immediate data</td>
</tr>
<tr>
<td></td>
<td>disp:</td>
<td>Displacement*</td>
</tr>
<tr>
<td>Instruction code</td>
<td>mmmm:</td>
<td>Source register</td>
</tr>
<tr>
<td></td>
<td>nnnn:</td>
<td>Destination register</td>
</tr>
<tr>
<td></td>
<td>0000:</td>
<td>R0</td>
</tr>
<tr>
<td></td>
<td>0001:</td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>1111:</td>
<td>R15</td>
</tr>
<tr>
<td></td>
<td>iii:</td>
<td>Immediate data</td>
</tr>
<tr>
<td></td>
<td>dddd:</td>
<td>Displacement</td>
</tr>
</tbody>
</table>

- **Operation**
  - **→**, **←**: Direction of transfer
  - **(xx)**: Memory operand
  - **M/Q/T**: Flag bits in the SR
  - **&**: Logical AND of each bit
  - **||**: Logical OR of each bit
  - **^**: Exclusive OR of each bit
  - **~**: Logical NOT of each bit
  - **<<n**: n-bit left shift
  - **>>n**: n-bit right shift

- **Execution cycles**
  - **—**: Value when no wait states are inserted*2

- **T bit**
  - **—**: Value of T bit after instruction is executed.
  - An em-dash (—) in the column means no change.

**Notes:**
1. Depending on the operand size, displacement is scaled ×1, ×2, or ×4. For details, see section 5, Instruction Descriptions.
2. Instruction execution cycles: The execution cycles shown in the table are minimums. The actual number of cycles may be increased when (1) contention occurs between instruction fetches and data access, or (2) when the destination register of the load instruction (memory → register) and the register used by the next instruction are the same.
### 5.1.1 Data Transfer Instructions

#### Table 5.3 Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #imm, Rn</td>
<td>1110nnnni1111i</td>
<td>imm → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W @(disp, PC), Rn</td>
<td>1001nnnn11ddddd</td>
<td>(disp×8+PC) → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L @(disp, PC), Rn</td>
<td>1101nnnn11ddddd</td>
<td>(disp×4+PC) → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV Rm, Rn</td>
<td>0110nnnnmmmm0011</td>
<td>Rm → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.B Rm, @Rn</td>
<td>0010nnnnmmmm0000</td>
<td>Rm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W Rm, @Rn</td>
<td>0010nnnnmmmm0001</td>
<td>Rm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L Rm, @Rn</td>
<td>0010nnnnmmmm0010</td>
<td>Rm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.B @Rm, Rn</td>
<td>0110nnnnmmmm0000</td>
<td>(Rm) → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W @Rm, Rn</td>
<td>0110nnnnmmmm0001</td>
<td>(Rm) → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L @Rm, Rn</td>
<td>0110nnnnmmmm0010</td>
<td>(Rm) → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.B Rm, @-Rn</td>
<td>0010nnnnmmmm0100</td>
<td>Rn - 1 → Rn, Rm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W Rm, @-Rn</td>
<td>0010nnnnmmmm0101</td>
<td>Rn - 2 → Rn, Rm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L Rm, @-Rn</td>
<td>0010nnnnmmmm1110</td>
<td>Rn - 4 → Rn, Rm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.B @Rm+, Rn</td>
<td>0110nnnnmmmm0100</td>
<td>(Rm) → sign extension → Rn, Rm + 1 → Rm</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W @Rm+, Rn</td>
<td>0110nnnnmmmm0101</td>
<td>(Rm) → sign extension → Rn, Rm + 2 → Rm</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.B @Rm+, Rn</td>
<td>0110nnnnmmmm0110</td>
<td>(Rm) → Rn, Rm + 4 → Rm</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W @Rm+, Rn</td>
<td>0110nnnnmmmm0101</td>
<td>(Rm) → (Rm)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L R0, @(disp, Rn)</td>
<td>10000000nnnn1ddd</td>
<td>R0 → (disp+Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W R0, @(disp, Rn)</td>
<td>10000001nnnn1ddd</td>
<td>R0 → (disp+2+Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L Rm, @(disp, Rn)</td>
<td>0001nnnnnnmmddd</td>
<td>Rm → (disp+1+Rn)</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W @(R0, Rm), R0</td>
<td>10000101nnnn1ddd</td>
<td>(disp+Rm) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.L @(disp, Rm), R0</td>
<td>0101nnnnnnnn1ddd</td>
<td>(disp+2+Rm) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.B @R0, Rm</td>
<td>0000nnnnnmm1100</td>
<td>(R0+Rm) → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>MOV.W @R0, Rm</td>
<td>0000nnnnnmm1101</td>
<td>(R0+Rm) → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>SH2E, SHH, FPU</td>
</tr>
<tr>
<td>Instruction</td>
<td>Code</td>
<td>Operation</td>
<td>Cycles</td>
<td>T Bit</td>
<td>Compatibility</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-----------------------------------------------</td>
<td>--------</td>
<td>-------</td>
<td>---------------------</td>
</tr>
<tr>
<td>MOV.L @Rn, Rn</td>
<td>0000nnnnnnnnnn</td>
<td>(R0+Rm) → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.B R0, @Rn</td>
<td>0100nnnnnnnnn</td>
<td>R0 → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W R0, @Rn+</td>
<td>0100nnnnnnnnn</td>
<td>R0 → (Rn), Rn + 2 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.L @Rn, Rn</td>
<td>0100nnnnnnnnn</td>
<td>R0 → (Rn), Rn + 4 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W @Rn, R0</td>
<td>0100nnnnnnnnn</td>
<td>Rm → (Rm), *(Rm) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.L @Rn, R0</td>
<td>0100nnnnnnnnn</td>
<td>Rm → (Rm), *(Rm) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W @Rn, R0</td>
<td>0100nnnnnnnnn</td>
<td>Rm → (Rm), *(Rm) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.L @Rn, R0</td>
<td>0100nnnnnnnnn</td>
<td>Rm → (Rm), *(Rm) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.B Rm, @Rn</td>
<td>0110nnnnnnnnn</td>
<td>(R0+Rm) → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.L Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.B Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>(R0+Rm) → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.L Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.B Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>(R0+Rm) → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.W Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV.L Rm, @Rn+</td>
<td>0110nnnnnnnnn</td>
<td>Rm → (Rn), *(Rn) → sign extension → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVA @Rn, R0</td>
<td>1100nnnnnnnnn</td>
<td>disp * PC → R0</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVI20 #imm20, Rn</td>
<td>0000nnnnnnnnn</td>
<td>imm → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVI20S #imm20, Rn</td>
<td>0000nnnnnnnnn</td>
<td>imm&lt;&lt;8 → sign extension → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVML.L Rm, @-R15</td>
<td>0100nnnn11110001</td>
<td>R15 - 4 → R15, Rm → (R15) R15 - 4 → R15, Rm - 1 → (R15) R15 - 4 → R15, R0 → (R15) Note: When Rm = R15, read Rm as PR</td>
<td>1 to 16</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVML.L @R15+, Rn</td>
<td>0100nnnn11110101</td>
<td>(R15) → R0, R15 + 4 → R15 (R15) → R1, R15 + 4 → R15 (R15) → Rn Note: When Rn = R15, read Rn as PR</td>
<td>1 to 16</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVUL.L Rm, @-R15</td>
<td>0100nnnn11110000</td>
<td>R15 - 4 → R15, PR → (R15) R15 - 4 → R15, R14 → (R15) R15 - 4 → R15, Rm → (R15) Note: When Rm = R15, read Rm as PR</td>
<td>1 to 16</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVUL.L @R15+, Rn</td>
<td>0100nnnn11110100</td>
<td>(R15) → Rn, R15 + 4 → R15 (R15) → Rn + 1, R15 + 4 → R15 (R15) → R14, R15 + 4 → R15 (R15) → PR Note: When Rn = R15, read Rn as PR</td>
<td>1 to 16</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVRT Rn</td>
<td>0000nnnn00111001</td>
<td>~ T → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOV T Rn</td>
<td>0000nnnn00101001</td>
<td>T → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>MOVU.B @(disp12,Rm), Rn</td>
<td>0011nnnnnn0001 1000dddddddddddd</td>
<td>(disp+Rm) → zero extension → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MOVU.W @(disp12,Rm), Rn</td>
<td>0011nnnnnn0001 1001dddddddddddd</td>
<td>(disp×2+Rm) → zero extension → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>NOTT</td>
<td>0000000000110100</td>
<td>~ T → T</td>
<td>1</td>
<td>Operation result</td>
<td>Yes</td>
</tr>
<tr>
<td>PREF @Rn</td>
<td>0000nnnn10000011</td>
<td>(Rn) → operand cache</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Compatibility

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>SH2E</th>
<th>SH4</th>
<th>New SH-ZA/SH2A-FPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAP.B</td>
<td>0110nnnnnnnn1000</td>
<td>Rm → swap lower 2 bytes → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>SWAP.W</td>
<td>0110nnnnnnnn1001</td>
<td>Rm → swap upper/lower words → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>XTRCT</td>
<td>0010nnnnnnnn1101</td>
<td>Rm:Rn middle 32 bits → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### 5.1.2 Arithmetic Operation Instructions

#### Table 5.4 Arithmetic Operation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rm, Rn</td>
<td>0011nnnnmm1100</td>
<td>Rn + Rm → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>ADD #imm, Rn</td>
<td>0111nnniiiiiiii</td>
<td>Rn + imm → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>ADDC Rm, Rn</td>
<td>0011nnnnmm1100</td>
<td>Rn + Rm + T → Rn, carry → T</td>
<td>1</td>
<td>Carry</td>
<td>Yes</td>
</tr>
<tr>
<td>ADDV Rm, Rn</td>
<td>0011nnnnmm1111</td>
<td>Rn + Rm → Rn, overflow → T</td>
<td>1</td>
<td>Overflow</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| CMP/EQ #imm, R0 | 10001000iiiiiiii | When R0 = imm, 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/EQ Rm, Rn | 0011nnnnmm0000 | When Rn = Rm, 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/HS Rm, Rn | 0011nnnnmm0010 | When Rn ≥ Rm (unsigned), 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/GE Rm, Rn | 0011nnnnmm0111 | When Rn ≥ Rm (signed), 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/HI Rm, Rn | 0011nnnnmm0110 | When Rn > Rm (unsigned), 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/GT Rm, Rn | 0011nnnnmm0111 | When Rn > Rm (signed), 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/PL Rn | 0100nnnn00010101 | When Rn > 0, 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/PZ Rn | 0100nnnn00010001 | When Rn ≥ 0, 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CMP/STR Rm, Rn | 0010nnnnmm1100 | When any bytes are equal, 1 → T  
Otherwise, 0 → T | 1 | Comparison result | Yes | Yes |
| CLIPS.B Rn | 0100nnnn10010001 | When Rn > (H’00000007F),  
(H’00000007F) → Rn, 1 → CS  
When Rn < (H’FFFFFF80),  
(H’FFFFFF80) → Rn, 1 → CS | 1 | — | Yes |
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIPS.W Rn</td>
<td>0100nnnn10010101</td>
<td>When Rn &gt; (H'000007FF), (H'000000FF) → Rn, 1 → CS</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When Rn &lt; (H'FFFF8000), (H'FFFF8000) → Rn, 1 → CS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLIPU.B Rn</td>
<td>0100nnnn10000001</td>
<td>When Rn &gt; (H'000000FF), (H'000000FF) → Rn, 1 → CS</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>CLIPU.W Rn</td>
<td>0100nnnn10000101</td>
<td>When Rn &gt; (H'000000FF), (H'000000FF) → Rn, 1 → CS</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>DIV1 Rn, Rn</td>
<td>0011nnnnmmmm0100</td>
<td>1-step division (Rn + Rm)</td>
<td>1</td>
<td></td>
<td>Calculati-on result</td>
</tr>
<tr>
<td>DIV0S Rn, Rn</td>
<td>0010nnnnmmmm0111</td>
<td>MSB of Rn → Q, MSB of Rm → M, M ^ Q → T</td>
<td>1</td>
<td></td>
<td>Calculati-on result</td>
</tr>
<tr>
<td>DIV0U</td>
<td>0000000000011001</td>
<td>0 → M/Q/T</td>
<td>1</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>DIVS R0, Rn</td>
<td>0100nnnn10010100</td>
<td>Signed, Rn + R0 → Rn</td>
<td>36</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>DIVU R0, Rn</td>
<td>0100nnnn10000100</td>
<td>Unsigned, Rn + R0 → Rn</td>
<td>34</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>DMULS.L Rm, Rn</td>
<td>0011nnnnmmmm1101</td>
<td>Signed, Rn × Rm → MACH, MACL</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>DMULU.L Rm, Rn</td>
<td>0011nnnnmmmm1101</td>
<td>Unsigned, Rn × Rm → MACH, MACL</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>DT Rn</td>
<td>0100nnnn00010000</td>
<td>Rn - 1 → Rn; when Rn = 0, 1 → T</td>
<td>1</td>
<td></td>
<td>Comparison result</td>
</tr>
<tr>
<td>EXTS.B Rn, Rn</td>
<td>0110nnnnmmmm1110</td>
<td>Rm sign-extended from byte → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>EXTS.W Rm, Rn</td>
<td>0110nnnnmmmm1111</td>
<td>Rm sign-extended from word → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>EXTU.B Rm, Rn</td>
<td>0110nnnnmmmm1100</td>
<td>Rm zero-extended from byte → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>EXTU.W Rm, Rn</td>
<td>0110nnnnmmmm1101</td>
<td>Rm zero-extended from word → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MAC.L @Rm+, @Rn+</td>
<td>0000nnnnmmmm1111</td>
<td>Signed, (Rn) × (Rm) + MAC → MAC</td>
<td>4</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MAC.W @Rm+, @Rn+</td>
<td>0100nnnnmmmm1111</td>
<td>Signed, (Rn) × (Rm) + MAC → MAC</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>MUL.L Rm, Rn</td>
<td>0000nnnnmmmm1111</td>
<td>Rn × Rm → MACL</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Section 5 Instruction Set**
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>SH2E</th>
<th>SH4</th>
<th>New SH-2A/SH2A-FPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULR R0, Rn</td>
<td>0100nnnn10000000</td>
<td>R0 \times Rn \rightarrow Rn 32 \times 32 \rightarrow 32 bits</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>MULS.W Rm, Rn</td>
<td>0010nnnnmmm1111</td>
<td>Signed, Rn \times Rm \rightarrow MACL 16 \times 16 \rightarrow 32 bits</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>MULU.W Rm, Rn</td>
<td>0010nnnnmmm1110</td>
<td>Unsigned, Rn \times Rm \rightarrow MACL 16 \times 16 \rightarrow 32 bits</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>NEG Rm, Rn</td>
<td>0110nnnnmmm1011</td>
<td>0 - Rm \rightarrow Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>NEGC Rm, Rn</td>
<td>0110nnnnmmm1010</td>
<td>0 - Rm - T \rightarrow Rn, borrow \rightarrow T 1 Borrow</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>SUB Rm, Rn</td>
<td>0011nnnnmmm1000</td>
<td>Rn - Rm \rightarrow Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>SUBC Rm, Rn</td>
<td>0011nnnnmmm1010</td>
<td>Rn - Rm - T \rightarrow Rn, borrow \rightarrow T 1 Borrow</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>SUBV Rm, Rn</td>
<td>0011nnnnmmm1011</td>
<td>Rn - Rm \rightarrow Rn, underflow \rightarrow T 1 Overflow</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### 5.1.3 Logic Operation Instructions

#### Table 5.5 Logic Operation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND Rm, Rn</td>
<td>0010nnnnnnmm1001</td>
<td>Rn &amp; Rm → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>AND #imm, R0</td>
<td>1100100111111111</td>
<td>R0 &amp; imm → R0</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>AND.B #imm, @(R0, GBR)</td>
<td>1100110111111111</td>
<td>(R0+GBR) &amp; imm → (R0+GBR)</td>
<td>3</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>NOT Rm, Rn</td>
<td>0110nnnnnnnn0111</td>
<td>~ Rm → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>OR Rm, Rn</td>
<td>0010nnnnnnnn1011</td>
<td>Rn</td>
<td>Rm → Rn</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>OR #imm, R0</td>
<td>1100101111111111</td>
<td>R0</td>
<td>imm → R0</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>OR.B #imm, @(R0, GBR)</td>
<td>1100111111111111</td>
<td>(R0+GBR)</td>
<td>imm → (R0+GBR)</td>
<td>3</td>
<td>—</td>
</tr>
<tr>
<td>TAS.B @Rn</td>
<td>0100nnnn00011011</td>
<td>When (Rn) = 0, 1 → T, otherwise 0 → T, 1 → MSB of (Rn)</td>
<td>3</td>
<td>Test result</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>TST Rm, Rn</td>
<td>0010nnnnnnnn1000</td>
<td>Rn &amp; Rm; when result = 0, 1 → T, otherwise 0 → T</td>
<td>1</td>
<td>Test result</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>TST #imm, R0</td>
<td>1100100001111111</td>
<td>R0 &amp; imm; when result = 0, 1 → T, otherwise 0 → T</td>
<td>1</td>
<td>Test result</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>TST.B #imm, @(R0, GBR)</td>
<td>1100110001111111</td>
<td>(R0 + GBR) &amp; imm; when result = 0, 1 → T, otherwise 0 → T</td>
<td>3</td>
<td>Test result</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>XOR Rm, Rn</td>
<td>0010nnnnnnnn1010</td>
<td>Rn ^ Rm → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>XOR #imm, R0</td>
<td>1100101011111111</td>
<td>R0 ^ imm → R0</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>XOR.B #imm, @(R0, GBR)</td>
<td>1100111011111111</td>
<td>(R0+GBR) ^ imm → (R0+GBR)</td>
<td>3</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
</tbody>
</table>
### 5.1.4 Shift Instructions

#### Table 5.6 Shift Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTL Rn</td>
<td>0100nnnn0000100</td>
<td>T ← Rn ← MSB</td>
<td>1</td>
<td>MSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>ROTR Rn</td>
<td>0100nnnn00000101</td>
<td>LSB → Rn → T</td>
<td>1</td>
<td>LSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>ROTCL Rn</td>
<td>0100nnnn00100100</td>
<td>T ← Rn ← T</td>
<td>1</td>
<td>MSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>ROTCR Rn</td>
<td>0100nnnn00100101</td>
<td>T → Rn → T</td>
<td>1</td>
<td>LSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHAD Rm, Rn</td>
<td>0100nnnnmmmm1100</td>
<td>When Rm ≥ 0, Rn&lt;&lt;Rm → Rn When Rm &lt; 0, Rn&gt;&gt;[Rm] → [MSB → Rn]</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>SHAL Rn</td>
<td>0100nnnn00100000</td>
<td>T ← Rn ← 0</td>
<td>1</td>
<td>MSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHAR Rn</td>
<td>0100nnnn00100001</td>
<td>MSB → Rn → T</td>
<td>1</td>
<td>LSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLD Rm, Rn</td>
<td>0100nnnnmmmm1101</td>
<td>When Rm ≥ 0, Rn&lt;&lt;Rm → Rn When Rm &lt; 0, Rn&gt;&gt;[Rm] → [0 → Rn]</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>SHLL Rn</td>
<td>0100nnnn00000000</td>
<td>T ← Rn ← 0</td>
<td>1</td>
<td>MSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLR Rn</td>
<td>0100nnnn00000001</td>
<td>0 → Rn → T</td>
<td>1</td>
<td>LSB</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLL2 Rn</td>
<td>0100nnnn00001000</td>
<td>Rn&lt;&lt;2 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLR2 Rn</td>
<td>0100nnnn00001001</td>
<td>Rn&gt;&gt;2 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLL8 Rn</td>
<td>0100nnnn00011000</td>
<td>Rn&lt;&lt;8 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLR8 Rn</td>
<td>0100nnnn00011001</td>
<td>Rn&gt;&gt;8 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLL16 Rn</td>
<td>0100nnnn00101000</td>
<td>Rn&lt;&lt;16 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>SHLR16 Rn</td>
<td>0100nnnn00101001</td>
<td>Rn&gt;&gt;16 → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
</tbody>
</table>
### 5.1.5 Branch Instructions

#### Table 5.7 Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF label</td>
<td>10001011ddddddd</td>
<td>When ( T = 0 ), ( \text{disp} \times 2 + \text{PC} \rightarrow \text{PC} ), when ( T = 1 ), nop</td>
<td>3/1*</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>BF/S label</td>
<td>10001111ddddddd</td>
<td>Delayed branch, when ( T = 0 ), ( \text{disp} \times 2 + \text{PC} \rightarrow \text{PC} ), when ( T = 1 ), nop</td>
<td>2/1*</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>BT label</td>
<td>10001101ddddddd</td>
<td>When ( T = 1 ), ( \text{disp} \times 2 + \text{PC} \rightarrow \text{PC} ), when ( T = 0 ), nop</td>
<td>3/1*</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>BT/S label</td>
<td>10001101ddddddd</td>
<td>Delayed branch, when ( T = 1 ), ( \text{disp} \times 2 + \text{PC} \rightarrow \text{PC} ), when ( T = 0 ), nop</td>
<td>2/1*</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>BRAF Rm</td>
<td>000000mm00100011</td>
<td>Delayed branch, ( \text{Rm} + \text{PC} \rightarrow \text{PC} )</td>
<td>2</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>BSR label</td>
<td>1011ddddddddddd</td>
<td>Delayed branch, ( \text{PC} \rightarrow \text{PR} ), ( \text{disp} \times 2 + \text{PC} \rightarrow \text{PC} )</td>
<td>2</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>BSRF Rm</td>
<td>000000mm00000011</td>
<td>Delayed branch, ( \text{PC} \rightarrow \text{PR} ), ( \text{Rm} + \text{PC} \rightarrow \text{PC} )</td>
<td>2</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>JMP @Rm</td>
<td>010000mm00101011</td>
<td>Delayed branch, ( \text{Rm} \rightarrow \text{PC} )</td>
<td>2</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>JSR @Rm</td>
<td>010000mm00001011</td>
<td>Delayed branch, ( \text{PC} \rightarrow \text{PR} ), ( \text{Rm} \rightarrow \text{PC} )</td>
<td>2</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>JSR/N @Rm</td>
<td>011000mm01001011</td>
<td>( \text{PC} - 2 \rightarrow \text{PR} ), ( \text{Rm} \rightarrow \text{PC} )</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>JSR/N @@(disp8, TBR)</td>
<td>10000001dddddddd</td>
<td>( \text{PC} - 2 \rightarrow \text{PR} ), ( \text{disp} \times 4 + \text{TBR} \rightarrow \text{PC} )</td>
<td>5</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>RTS</td>
<td>0000000000001011</td>
<td>Delayed branch, ( \text{PR} \rightarrow \text{PC} )</td>
<td>2</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>RTS/N</td>
<td>0000000001101011</td>
<td>( \text{PR} \rightarrow \text{PC} )</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>RTV/N Rm</td>
<td>000000mm01110111</td>
<td>( \text{Rm} \rightarrow \text{R0} ), ( \text{PR} \rightarrow \text{PC} )</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note: * One state when the program does not branch.
### Table 5.8  System Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SH2E</td>
</tr>
<tr>
<td>CLRT</td>
<td>00000000000101000</td>
<td>0 → T</td>
<td>1</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>CLRMAC</td>
<td>0000000000101000</td>
<td>0 → MACH, MACL</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDBANK</td>
<td>@Rm, R0</td>
<td>(Specified register bank entry) → R0</td>
<td>6</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC</td>
<td>Rm, SR</td>
<td>Rm → SR</td>
<td>3</td>
<td>LSB</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC</td>
<td>Rm, TBR</td>
<td>Rm → TBR</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC</td>
<td>Rm, GBR</td>
<td>Rm → GBR</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC</td>
<td>Rm, VBR</td>
<td>Rm → VBR</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC.L @Rm+, SR</td>
<td>0100000000000111</td>
<td>(Rm) → SR, Rm + 4 → Rm</td>
<td>5</td>
<td>LSB</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC.L @Rm+, GBR</td>
<td>0100000000101110</td>
<td>(Rm) → GBR, Rm + 4 → Rm</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDC.L @Rm+, VBR</td>
<td>0100000001001110</td>
<td>(Rm) → VBR, Rm + 4 → Rm</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDS</td>
<td>Rm, MACH</td>
<td>Rm → MACH</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDS</td>
<td>Rm, MACL</td>
<td>Rm → MACL</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDS</td>
<td>Rm, PR</td>
<td>Rm → PR</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDS.L @Rm+, MACH</td>
<td>0100000000010100</td>
<td>(Rm) → MACH, Rm + 4 → Rm</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDS.L @Rm+, MACL</td>
<td>0100000000101000</td>
<td>(Rm) → MACL, Rm + 4 → Rm</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>LDS.L @Rm+, PR</td>
<td>0100000001001000</td>
<td>(Rm) → PR, Rm + 4 → Rm</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>NOP</td>
<td>0000000000001001</td>
<td>No operation</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>RESBANK</td>
<td>0000000000110111</td>
<td>Bank → R0 to R14, GBR, MACH, MACL, PR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTE</td>
<td>0000000001010111</td>
<td>Delayed branch, stack area → PC/SR</td>
<td>6</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>SETT</td>
<td>0000000000111001</td>
<td>1 → T</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>SLEEP</td>
<td>0000000000111001</td>
<td>Sleep</td>
<td>5</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STBANK</td>
<td>R0, @Rn</td>
<td>R0 (specified register bank entry)</td>
<td>7</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC</td>
<td>SR, Rn</td>
<td>SR → Rn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC</td>
<td>TBR, Rn</td>
<td>TBR → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC</td>
<td>GBR, Rn</td>
<td>GBR → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC</td>
<td>VBR, Rn</td>
<td>VBR → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC.L</td>
<td>SR, @-Rn</td>
<td>Rn - 4 → Rn, SR → (Rn)</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC.L</td>
<td>GBR, @-Rn</td>
<td>Rn - 4 → Rn, GBR → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STC.L</td>
<td>VBR, @-Rn</td>
<td>Rn - 4 → Rn, VBR → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Instruction Set Compatibility

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS MACH, Rn</td>
<td>0000nnnn00001010</td>
<td>MACH → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STS MACL, Rn</td>
<td>0000nnnn00011010</td>
<td>MACL → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STS PR, Rn</td>
<td>0000nnnn00101010</td>
<td>PR → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STS.L MACH, @-Rn</td>
<td>0100nnnn00000010</td>
<td>Rn - 4 → Rn, MACH → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>STS.L MACL, @-Rn</td>
<td>0100nnnn00010010</td>
<td>Rn - 4 → Rn, MACL → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>TRAPA #imm</td>
<td>110000111111111</td>
<td>PC/SR → stack area, (imm × 4 + VBR) → PC</td>
<td>5</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:** The execution cycles shown in the table are minimums. The actual number of cycles may be increased when (1) contention occurs between instruction fetches and data access, or (2) when the destination register of the load instruction (memory → register) and the register used by the next instruction are the same.

* In the event of bank overflow, the number of states is 19.
## 5.1.7 Floating-Point Instructions

Table 5.9 Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SH2E</td>
</tr>
<tr>
<td>FABS FRn</td>
<td>1111nnn010111101</td>
<td>[FRn] → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FABS DRn</td>
<td>1111nnn010111101</td>
<td>[DRn] → DRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FADD FRm, FRn</td>
<td>1111nnnn000000000</td>
<td>FRm + FRn → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FADD DRm, DRn</td>
<td>1111nnn00mm0000000</td>
<td>DRm + DRn → DRn</td>
<td>6</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FCMP/EQ.FRm, FRn</td>
<td>1111nnnn000000000</td>
<td>(FRm=FRn)? 1:0 → T</td>
<td>1</td>
<td>Comparison result</td>
<td>Yes</td>
</tr>
<tr>
<td>FCMP/EQ.DRm, DRn</td>
<td>1111nnn0mm01000</td>
<td>(DRm=DRm)? 1:0 → T</td>
<td>2</td>
<td>Comparison result</td>
<td>Yes</td>
</tr>
<tr>
<td>FCMP/GT.FRm, FRn</td>
<td>1111nnnn000001000</td>
<td>(FRm&gt;FRm)? 1:0 → T</td>
<td>1</td>
<td>Comparison result</td>
<td>Yes</td>
</tr>
<tr>
<td>FCMP/GT.DRm, DRn</td>
<td>1111nnn0mm01000</td>
<td>(DRm&gt;DRm)? 1:0 → T</td>
<td>2</td>
<td>Comparison result</td>
<td>Yes</td>
</tr>
<tr>
<td>FCNVDS DRm, FPUL</td>
<td>1111nnn010111101</td>
<td>(float) DRm → FPUL</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FCNVSD FPUL, DRn</td>
<td>1111nnn010111101</td>
<td>(double) FPUL → DRn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FDIV FRm, FRn</td>
<td>1111nnnn0000000000</td>
<td>FRn/FRm → FRn</td>
<td>10</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FDIV DRm, DRn</td>
<td>1111nnn00mm01100</td>
<td>DRm/DRm → DRn</td>
<td>23</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FLD0 FRn</td>
<td>1111nnn010111101</td>
<td>0 × 00000000 → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FLD1 FRn</td>
<td>1111nnn100011101</td>
<td>0 × 3F000000 → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FLDS FRm, FPUL</td>
<td>1111nnn000001101</td>
<td>(Rm) → FPUL</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOAT FPUL,FRn</td>
<td>1111nnn000101101</td>
<td>(float) FPUL → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOAT FPUL,DRn</td>
<td>1111nnn00001011101</td>
<td>(double) FPUL → DRn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMAC FR0,FRm,FRn</td>
<td>1111nnnn000011110</td>
<td>FR0 × FRm + FRn → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV FRm, FRn</td>
<td>1111nnn00mm0101000</td>
<td>FRm → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV DRm, DRn</td>
<td>1111nnn00mm0110000</td>
<td>DRm → DRn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.S @(R0, Rm), FRn</td>
<td>1111nnn00mm01110</td>
<td>(R0+Rm) → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.D @(R0, Rm), DRn</td>
<td>1111nnn00mm01110</td>
<td>(R0+Rm) → DRn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.S @Rm+, FRn</td>
<td>1111nnn00mm1000</td>
<td>(Rm) → FRn, Rm+ = 4</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.D @Rm+, DRn</td>
<td>1111nnn00mm1000</td>
<td>(Rm) → DRn, Rm+ = 8</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.S @Rm, FRn</td>
<td>1111nnn00mm1000</td>
<td>(Rm) → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.D @Rm, DRn</td>
<td>1111nnn00mm1000</td>
<td>(Rm) → DRn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Rev. 3.00 Jul 08, 2005 page 66 of 484
REJ09B0051-0300
## Section 5  Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV.S  @(disp12,Rm),FRn</td>
<td>0011nnnnnmmm0001</td>
<td>(disp×4+Rm) → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>0111ddddddddddddd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMOV.D  @(disp12,Rm),DRn</td>
<td>0011nnnnnmmm0001</td>
<td>(disp×8+Rm) → DRn</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>0111ddddddddddddd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMOV.S  FRm, @(R0,Rn)</td>
<td>1111nnnnnmmm0111</td>
<td>FRm → (R0+Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FMOV.D  DRm, @(R0,Rn)</td>
<td>1111nnnnnmmm0111</td>
<td>DRm → (R0+Rn)</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.S  FRm, @-Rn</td>
<td>1111nnnnnmmm1011</td>
<td>FRm → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FMOV.D  DRm, @-Rn</td>
<td>1111nnnnnmmm1011</td>
<td>DRm → (Rn)</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FMOV.S  FRm, @(disp12,Rn)</td>
<td>0011nnnnnmmm00010</td>
<td>FRm → (disp×4+Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>0111ddddddddddddd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMOV.D  DRm, @(disp12,Rn)</td>
<td>0011nnnnnmmm00010</td>
<td>DRm → (disp×8+Rn)</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>0111ddddddddddddd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL   FRn, FRn</td>
<td>1111nnnnnmmm0010</td>
<td>FRn × FRn → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FMUL   DRn, DRn</td>
<td>1111nnnnnmmm00010</td>
<td>DRn × DRn → DRn</td>
<td>6</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FNEG   FRn</td>
<td>1111nnnnnmmm1011</td>
<td>-FRn → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FNEG   DRn</td>
<td>1111nnnnnmmm1011</td>
<td>-DRn → DRn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FSCHG</td>
<td>1111001111111101</td>
<td>FPSCR.SZ = ~ FPSCR.SZ</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FSQRT  FRn</td>
<td>1111nnnnnmmm0010</td>
<td>√FRn → FRn</td>
<td>9</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FSQRT  DRn</td>
<td>1111nnnnnmmm0010</td>
<td>√DRn → DRn</td>
<td>22</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FSTS   FPUL,FRn</td>
<td>1111nnnnnmmm00010</td>
<td>FPUL → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FSUB   FRn, FRn</td>
<td>1111nnnnnmmm00010</td>
<td>FRn - FRn → FRn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FSUB   DRn, DRn</td>
<td>1111nnnnnmmm00010</td>
<td>DRn - DRn → DRn</td>
<td>6</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>FTRC   FRn, FPUL</td>
<td>1111nnnnnmmm0011101</td>
<td>(long) FRn → FPUL</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>FTRC   DRn, FPUL</td>
<td>1111nnnnnmmm0011101</td>
<td>(long) DRn → FPUL</td>
<td>2</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### 5.1.8 FPU-Related CPU Instructions

#### Table 5.10 FPU-Related CPU Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS Rm,FPSCR</td>
<td>0100nnnn0111010</td>
<td>Rm → FPSCR</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>LDS Rm,FPUL</td>
<td>0100nnnn0111110</td>
<td>Rm → FPUL</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>LDS.L @Rm+, FPSCR</td>
<td>0100nnnn01100110</td>
<td>(Rm) → FPSCR, Rm = 4</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>LDS.L @Rm+, FPUL</td>
<td>0100nnnn01101110</td>
<td>(Rm) → FPUL, Rm = 4</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>STS FPSCR, Rn</td>
<td>0000nnnn01101010</td>
<td>FPSCR → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>STS FPUL,Rn</td>
<td>0000nnnn01101100</td>
<td>FPUL → Rn</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>STS.L FPSCR,@-Rn</td>
<td>0100nnnn01100010</td>
<td>Rn = 4, FPSCR → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
<tr>
<td>STS.L FPUL,@-Rn</td>
<td>0100nnnn01110010</td>
<td>Rn = 4, FPUL → (Rn)</td>
<td>1</td>
<td>—</td>
<td>Yes Yes</td>
</tr>
</tbody>
</table>
5.1.9 Bit Manipulation Instructions

Table 5.11 Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Operation</th>
<th>Cycles</th>
<th>T Bit</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SH2E</td>
</tr>
<tr>
<td>BAND.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn011100101000ddddd</td>
<td>(imm of (disp+Rn)) &amp; T &amp; T</td>
<td>3</td>
<td>Operation result</td>
<td>Yes</td>
</tr>
<tr>
<td>BANDNOT.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn011101001100ddddd</td>
<td>~ (imm of (disp+Rn)) &amp; T → T</td>
<td>3</td>
<td>Operation result</td>
<td>Yes</td>
</tr>
<tr>
<td>BCLR.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01110000001100ddddd</td>
<td>0 → (imm of (disp+Rn))</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>BCLR #imm3, Rn</td>
<td>1000011000nnn01111000ddddd</td>
<td>0 → imm of Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>BLD.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>(imm of (disp+Rn)) → T</td>
<td>3</td>
<td>Operation result</td>
<td>Yes</td>
</tr>
<tr>
<td>BLD #imm3, Rn</td>
<td>100001111nnnn1111111111111</td>
<td>imm of Rn → T</td>
<td>1</td>
<td>Operation result</td>
<td>Yes</td>
</tr>
<tr>
<td>BLDNOT.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>~ (imm of (disp+Rn)) → T</td>
<td>3</td>
<td>Operation result</td>
<td>Yes</td>
</tr>
<tr>
<td>BOR.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>(imm of (disp+Rn))</td>
<td>T → T</td>
<td>3</td>
<td>Operation result</td>
</tr>
<tr>
<td>BORNOT.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>~ (imm of (disp+Rn))</td>
<td>T → T</td>
<td>3</td>
<td>Operation result</td>
</tr>
<tr>
<td>BSET.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>1 → (imm of (disp+Rn))</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>BSET #imm3, Rn</td>
<td>1000011000nnn11111000ddddd</td>
<td>1 → imm of Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>BST.B #imm3,@(disp12,Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>(imm of (disp+Rn)) → T</td>
<td>3</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>BST #imm3, Rn</td>
<td>100001111nnnn1111111111111</td>
<td>T → imm of Rn</td>
<td>1</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>BXOR.B #imm3, @(disp12, Rn)</td>
<td>0011nnnn01111000001100ddddd</td>
<td>(imm of (disp+ Rn))</td>
<td>T → T</td>
<td>3</td>
<td>Operation result</td>
</tr>
</tbody>
</table>
Section 6  Instruction Descriptions

6.1 Overview of New Instructions

In the SH-2A/SH2A-FPU, new instructions have been added in vacant locations other than instruction codes assigned to SH-2E CPU instructions (instruction codes with upper 4 bits of 0000 to 1110) and SH4 FPU instructions (instruction codes with upper 4 bits of 1111). However, the SH-2A does not support the following SH4 FPU instructions: (a) FMOV instructions specifying XDm/XDn, (b) the FRCHG instruction, and (c) FIPR, and FTRV instructions.

This section gives detailed descriptions of the new instructions.

The new instructions are those described in (1) to (14) below. (1) to (3) are 32-bit fixed-length instructions, and (4) to (14) are 16-bit fixed-length instructions.

(1) Immediate Transfer Instructions

MOVI20, MOVI20S

These instructions transfer 20-bit immediate data in the instruction code to a register. Combination with one of these instructions simplifies generation of a 28-bit address, making it possible to specify on-chip memory addresses for a maximum of 256 MB.
(2) **Structure Access Instructions**

MOV.B/W/L Rm, @(disp12, Rn), MOV.B/W/L @(disp12, Rm), Rn  
MOVU.B/W @(disp12, Rm), Rn  
FMOV.S FRm, @(disp12, Rn), FMOV.S @(disp12, Rm), FRn  
FMOV.D DRm, @(disp12, Rn), FMOV.D @(disp12, Rm), DRn

These instructions reference memory by specifying a 12-bit displacement located in the instruction code. An MOVU unsigned load instruction that automatically performs execution of zero extension has also been added.

(3) **Bit Manipulation Instructions (Operating on Memory)**

BAND.B #imm3, @(disp12, Rn), BOR.B #imm3, @(disp12, Rn)  
BCLR.B #imm3, @(disp12, Rn), BSET.B #imm3, @(disp12, Rn)  
BST.B #imm3, @(disp12, Rn), BLD.B #imm3, @(disp12, Rn)  
BXOR.B #imm3, @(disp12, Rn)  
BANDNOT.B #imm3, @(disp12, Rn), BORNOT.B #imm3, @(disp12, Rn)  
BLDNOT.B #imm3, @(disp12, Rn)

The BAND.B, BOR.B, and BXOR.B instructions perform logical operations between a bit in memory and the T bit, and store the result in the T bit. The BCLR.B and BSET.B instructions manipulate a bit in memory. The BST.B and BLD.B instructions execute a transfer between a bit in memory and the T bit. The BANDNOT.B and BORNOT.B instructions perform logical operations between the value resulting from inverting a bit in memory and the T bit, and store the result in the T bit. The BLDNOT.B instruction inverts a bit in memory and stores the result in the T bit. Bits other than the specified bit are not affected.

(4) **Bit Manipulation Instructions (Operating on a General Register)**

BCLR #imm3, Rn, BSET #imm3, Rn  
BST #imm3, Rn, BLD #imm3, Rn

The BCLR and BSET instructions manipulate one of the LSB 8 bits of a general register Rn. The BST and BLD instructions execute a transfer between one of the LSB 8 bits of a general register Rn and the T bit. Bits other than the specified bit are not affected.
(5) Multiplication Result Rn Storage Instruction

MULR

MULR performs a 32-bit x 32-bit multiplication, and stores the lower 32 bits of the result in a general register Rn.

(6) Batch Division Instructions

DIVS, DIVU

These instructions perform batch 32-bit ÷ 32-bit division. The DIVU instruction performs division of unsigned data, and the DIVS instruction performs division of signed data.

(7) Saturation Value Comparison Instructions

CLIPS, CLIPU

These instructions perform a comparison with a saturation value, and store the saturation upper-limit value in a general register Rn if the general register Rn contents exceed the saturation upper-limit value, or store the saturation lower-limit value in general register Rn if the general register Rn contents are less than the saturation upper-limit value. Only byte and word saturation values are supported.

(8) Barrel Shift Instructions

SHAD, SHLD

These instructions shift arbitrary bits. Two kinds of instructions are provided, for an arithmetic shift and a logical shift.

(9) Multiple Register Save/Restore Instructions

MOVML, MOVMU

These instructions save a number of consecutive registers to memory, or restore a number of consecutive registers from memory. It is possible to specify a general register Rn, and to save or restore consecutive general registers higher than or lower than the specified Rn.
(10) T Bit Inversion and Transfer Instructions

    MOVRT, NOTT

These instructions invert the T bit and transfer the resulting value to a general register Rn or the T bit.

(11) Register Bank Related Instructions

    RESBANK, STBANK, LDBANK

These are register bank related instructions that are provided in order to speed up interrupt handling.

(12) Reverse Stack Transfer Instructions

    MOV.B/W/L

These are transfer instructions in which the stack expansion direction is reversed.

(13) Unconditional Branch Instructions with No Delay Slot

    JSR/N, RTS/N

Instructions that do not have a delay slot are provided in order to reduce the code size by cutting down on the number of unnecessary NOP instructions.

(14) Cache-Related Instruction

    PREF

An SH3-DSP cache-related instruction is provided.
6.2 Format of Instruction Descriptions

**Format of this Section:** The format used for describing instructions is as shown below.

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Instruction Function (Explanation of Instruction Name)</th>
<th>Instruction Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>Abstract</td>
<td>Compatibility</td>
</tr>
<tr>
<td>Shown in assembler input format. imm and disp are numeric values, expressions, or symbols.</td>
<td>Summarizes the operation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shown in MSB ↔ LSB order.</td>
<td>Value in case of no-wait operation.</td>
</tr>
<tr>
<td></td>
<td>Shows the value of the T bit after execution of the instruction.</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Describes the operation of the instruction.

**Notes**

Mentions points requiring particular attention when using the instruction.

**Operation**

Shows the operation of the instruction in C. Provided as a reference to explain the operation of the instruction. The use of the following resources is assumed here.

```c
unsigned char Read_Blk (unsigned long Addr);
unsigned short Read_Blk (unsigned long Addr);
unsigned int Read_Blk (unsigned long Addr);
unsigned long Read_Blk (unsigned long Addr);
unsigned double Read_Blk (unsigned long Addr);
```

The size of address Addr is returned. A word read from other than a 2n address or a longword read from other than a 4n address will be detected as an address error.

```c
unsigned long Read_Bank_Long (unsigned long Addr);
```

The contents of the register bank entry indicated by the contents of address Addr are returned.
unsigned char Write_Byte (unsigned long Addr, unsigned long Data);
unsigned short Write_Word (unsigned long Addr, unsigned long Data);
unsigned int Write_Int (unsigned long Addr, unsigned long Data);
unsigned long Write_Long (unsigned long Addr, unsigned long Data);
unsigned double Write_Quad (unsigned long Addr, unsigned long Data);

Data Data is written to address Addr using the respective size. A word write to other than a 2n address or a longword write to other than a 4n address will be detected as an address error.

unsigned long Write_Bank_Long (unsigned long Addr, unsigned long Data);

Data Data is written to the register bank entry indicated by the contents of address Addr.

unsigned long R[16];
unsigned long SR, GBR, VBR, TBR;
unsigned long MACH, MACL, PR;
unsigned long PC;

Respective registers

struct BANK {
    unsigned long Rn_BANK[15];
    unsigned long GBR_BANK;
    unsigned long MACH_BANK;
    unsigned long MACL_BANK;
    unsigned long PR_BANK;
    unsigned long IVN;
};

BANK Register_Bank[512];

Register bank structure definition
(VTO: Interrupt vector table address offset)

struct SR0 {
    unsigned long dummy0:17;
    unsigned long BO0:1
    unsigned long CS0:1;
    unsigned long dummy1:3;
    unsigned long M0:1;
    unsigned long Q0:1;
    unsigned long I0:4;

unsigned long dummy2:2;
unsigned long S0:1;
unsigned long T0:1;
}

SR structure definition

#define BO ((* (struct SR0 *) (&SR)).BO0)
#define CS ((* (struct SR0 *) (&SR)).CS0)
#define M ((* (struct SR0 *) (&SR)).M0)
#define Q ((* (struct SR0 *) (&SR)).Q0)
#define I ((* (struct SR0 *) (&SR)).I0)
#define S ((* (struct SR0 *) (&SR)).S0)
#define T ((* (struct SR0 *) (&SR)).T0)

Definition of bits in SR

Error (char *er);

Error indication function

These are floating-point number definition statements.
#define PZERO 0
#define NZERO 1
#define DENORM 2
#define NORM 3
#define PINF 4
#define NINF 5
#define qNaN 6
#define sNaN 7
#define EQ 0
#define GT 1
#define LT 2
#define UO 3
#define INVALID 4
#define FADD 0
#define FSUB 1

#define CAUSE 0x0003f000 /* FPSCR(bit17-12) */
#define SET_E       0x00020000  /* FPSCR(bit17) */
#define SET_V       0x00010040  /* FPSCR(bit16,6) */
#define SET_Z       0x00008020  /* FPSCR(bit15,5) */
#define SET_O       0x00004010  /* FPSCR(bit14,4) */
#define SET_U       0x00002008  /* FPSCR(bit13,3) */
#define SET_I       0x00001004  /* FPSCR(bit12,2) */
#define ENABLE_VOUI 0x00000b80  /* FPSCR(bit11,9-7) */
#define ENABLE_V    0x00000800  /* FPSCR(bit11) */
#define ENABLE_Z    0x00000400  /* FPSCR(bit10) */
#define ENABLE_OUI  0x00000380  /* FPSCR(bit9-7) */
#define ENABLE_I    0x00000080  /* FPSCR(bit7) */
#define FLAG        0x0000007C  /* FPSCR(bit6-2) */

#define FPSCR_FR    FPSCR>>21&1
#define FPSCR_PR    FPSCR>>19&1
#define FPSCR_DN    FPSCR>>18&1
#define FPSCR_I     FPSCR>>12&1
#define FPSCR_RM    FPSCR&1
#define FR_HEX      frf.l[ FPSCR_FR]
#define FR          frf.f[ FPSCR_FR]
#define DR_HEX      frf.f[ FPSCR_FR]
#define DR       frf.d[ FPSCR_FR]

union {
    int  l[2][16];
    float f[2][16];
    double d[2][8];
} frf;
int FPSCR;

int sign_of(int n)
{
    return(FR_HEX[n]>>31);
}

int data_type_of(int n) {

int abs;
abs = FR_HEX[n] & 0x7fffffff;
if(FPSCR_PR == 0) { /* Single-precision */
if(abs < 0x00800000){
   if((FPSCR_DN == 1) || (abs == 0x00000000)){
      if(sign_of(n) == 0) {zero(n, 0); return(PZERO);} else zero(n, 1); return(NZERO);} 
} else return(DENORM);
} else if(abs < 0x7f800000) return(NORM);
else if(abs == 0x7f800000) {
   if(sign_of(n) == 0) return(PINF); else return(NINF);
}
else if(abs < 0x7fc00000) return(qNaN);
else return(sNaN);
}
else { /* Double-precision */
if(abs < 0x00100000){
   if((FPSCR_DN == 1) || ((abs == 0x00000000) && (FR_HEX[n+1] == 0x00000000))){
      if(sign_of(n) == 0) {zero(n, 0); return(PZERO);} else {zero(n, 1); return(NZERO);} 
} else return(DENORM);
} else if(abs < 0x7ff00000) return(NORM);
else if((abs == 0x7ff00000) && (FR_HEX[n+1] == 0x00000000)) {
   if(sign_of(n) == 0) return(PINF); else return(NINF);
}
else if(abs < 0x7ff80000) return(qNaN);
else return(sNaN);
void register_copy(int m, n)
{
    FR[n] = FR[m];
    if(FPSCR_PR == 1) FR[n+1] = FR[m+1];
}

void normal_faddsub(int m, n, type)
{
    union {
        float f;
        int l;
    } dstf, srcf;
    union {
        long d;
        int l[2];
    } dstd, srcd;
    union {
        long double x;
        int l[4];
    } dstx,
    if(FPSCR_PR == 0) {
        if(type == FADD) srcf.f = FR[m];
        else srcf.f = -FR[m];
        dstd.d = FR[n]; /* Conversion from single-precision to double-precision */
        dstd.d += srcf.f;
        if(((dstd.d == FR[n]) && (srcf.f != 0.0)) ||
            ((dstd.d == srcf.f) && (FR[n] != 0.0))) {
            set_I();
            if(sign_of(m)^ sign_of(n)) {
                dstd.l[1] = 1;
                if(dstd.l[1] == 0xffffffff) dstd.l[0] = 1;
            }
        }
        if(dstd.l[1] & 0xffffffff) set_I();
    }
dstf.f = srcf.f; /* Round to nearest */
if(FPSCR_RM == 1) {
    
    dstd.l[1] &= 0xe0000000; /* Round to zero */
    dstf.f = dstd.d;
}
check_single_exception(&FR[n],dstf.f);
} else {
    if(type == FADD) srcd.d = DR[m>>1];
    else srcd.d = -DR[m>>1];
dstx.x = DR[n>>1];
    /* Conversion from double-precision to extended double-precision */
dstx.x += srcd.d;
if(((dstx.x == DR[n>>1]) && (srcd.d != 0.0)) ||
    ((dstx.x == srcd.d) && (DR[n>>1] != 0.0))) {
    set_I();
    if(sign_of(m) ^ sign_of(n)) {
        
        dstl.l[3] -= 1;
        if(dstl.l[2] == 0xffffffff) {dstl.l[1] -= 1;
        if(dstl.l[1] == 0xffffffff) {dstl.l[0] -= 1;}}}
    }
}
if((dstl.l[2] & 0xffffffff) || dstl.l[3]) set_I();
dst.d += srcd.d; /* Round to nearest */
if(FPSCR_RM == 1) {
    dstl.l[2] &= 0xf0000000; /* Round to zero */
    dstl.l[3] = 0x00000000;
    dst.d = dstl.x;
}
check_double_exception(&DR[n>>1],dst.d);
}
void normal_fmul(int m,n)
{
    union {

float f;
int l;
} tmpf;
union {
    double d;
    int l[2];
} tmpd;
union {
    long double x;
    int l[4];
} tmpx;

if(FPSCR_PR == 0) {
    tmpd.d = FR[n]; /* Single-precision to double-precision */
    tmpd.d *= FR[m]; /* Precise creation */
    tmpf.f *= FR[m]; /* Round to nearest */
    if(tmpf.f != tmpd.d) set_I();
    if((tmpf.f > tmpd.d) && (FPSCR_RM == 1)) {
        tmpf.l -= 1; /* Round to zero */
    }
    check_single_exception(&FR[n],tmpf.f);
} else {
    tmpx.x = DR[n>>1]; /* Single-precision to double-precision */
    tmpx.x *= DR[m>>1]; /* Precise creation */
    tmpd.d *= DR[m>>1]; /* Round to nearest */
    if(tmpd.d != tmpx.x) set_I();
    if(tmpd.d > tmpx.x) && (FPSCR_RM == 1)) {
        tmpd.l[1] -= 1; /* Round to zero */
        if(tmpd.l[1] == 0xffffffff) tmpd.l[0] -= 1;
    }
    check_double_exception(&DR[n>>1], tmpd.d);
}

void check_single_exception(float *dst,result)
{
union {

Rev. 3.00 Jul 08, 2005 page 82 of 484
REJ09B0051-0300

REJ09B0051-0300
float f;
int l;
}
tmp;
float abs;
if(result < 0.0) tmp.l = 0xff800000; /* –infinity */
else tmp.l = 0x7f800000; /* +infinity */
if(result == tmp.f) {
    set_O(); set_I();
    if(FPSCR_RM == 1) {
        tmp.l -= 1; /* Maximum value of normalized number */
        result = tmp.f;
    }
}
if(result < 0.0) abs = -result;
else abs = result;
tmp.l = 0x00800000; /* Minimum value of normalized number */
if(abs < tmp.f) {
    if((FPSCR_DN == 1) && (abs != 0.0)) {
        set_I();
        if(result < 0.0) result = -0.0; /* Zeroize denormalized number */
        else result = 0.0;
    }
    if(FPSCR_I == 1) set_U();
}
if(FPSCR & ENABLE_OUI) fpu_exception_trap();
else *dst = result;
}

void check_double_exception(double *dst, result)
{
union {
    double d;
    int l[2];
} tmp;
double abs;
if(result < 0.0) tmp.l[0] = 0xfff00000; /* –infinity */
else              tmp.l[0] = 0x7ff00000; /* + infinity */
                    tmp.l[1] = 0x00000000;

        if(result == tmp.d)
            set_O(); set_I();
            if(FPSCR_RM == 1) {
                tmp.l[0] -= 1;
                tmp.l[1] = 0xffffffff;
                result = tmp.d; /* Maximum value of normalized number */
            }
        }

        if(result < 0.0)  abs = -result;
        else              abs =  result;
        tmp.l[0] = 0x00100000; /* Minimum value of normalized number */
                    tmp.l[1] = 0x00000000;
        if(abs < tmp.d) {
            if((FPSCR_DN == 1) && (abs != 0.0)) {
                set_I();
                if(result < 0.0) result = -0.0;
                /* Zeroize denormalized number */
                else             result =  0.0;
            }
            if(FPSCR_I == 1) set_U();
        }

        if(FPSCR & ENABLE_OUI) fpu_exception_trap();
        else                    *dst = result;
}

int check_product_invalidate(int m,n)
{
    return(check_product_infinity(m,n)  &&
            ((data_type_of(m) == PZERO) || (data_type_of(n) == PZERO) ||
            (data_type_of(m) == NZERO) || (data_type_of(n) == NZERO)));
}

int check_product_infinity(int m,n)
{
    return((data_type_of(m) == PINF) || (data_type_of(n) == PINF) ||
(data_type_of(m) == NINF) || (data_type_of(n) == NINF));

int check_positive_infinity(int m, n)
{
    return(((check_product_infinity(m, n) && (~sign_of(m) ^ sign_of(n)))
    ||
    ((check_product_infinity(m+1, n+1) && (~sign_of(m+1) ^
      sign_of(n+1)))) ||
    ((check_product_infinity(m+2, n+2) && (~sign_of(m+2) ^
      sign_of(n+2)))) ||
    ((check_product_infinity(m+3, n+3) && (~sign_of(m+3) ^
      sign_of(n+3)))));
}

int check_negative_infinity(int m, n)
{
    return((((check_product_infinity(m, n) && (sign_of(m) ^ sign_of(n)))
    ||
    ((check_product_infinity(m+1, n+1) && (sign_of(m+1) ^ sign_of(n+1)))
    ||
    ((check_product_infinity(m+2, n+2) && (sign_of(m+2) ^ sign_of(n+2)))
    ||
    ((check_product_infinity(m+3, n+3) && (sign_of(m+3) ^
      sign_of(n+3)))));
}

void clear_cause () {FPSCR &= ~CAUSE;}

void set_E() {FPSCR |= SET_E; fpu_exception_trap();}
void set_V() {FPSCR |= SET_V;}
void set_Z() {FPSCR |= SET_Z;}
void set_O() {FPSCR |= SET_O;}
void set_U() {FPSCR |= SET_U;}
void set_I() {FPSCR |= SET_I;}

void invalid(int n)
{
    set_V();
    if((FPSCR & ENABLE_V) == 0 qnan(n);
    else fpu_exception_trap();
}

void dz(int n, sign)
Section 6  Instruction Descriptions

{ 
    set_Z();
    if((FPSCR & ENABLE_Z) == 0 inf(n,sign);
    else    fpu_exception_trap();
}

void zero(int n,sign) 
{
    if(sign == 0)    FR_HEX [n]   = 0x00000000;
    else             FR_HEX [n]   = 0x80000000;
    if (FPSCR_PR==1) FR_HEX [n+1] = 0x00000000;
}

void inf(int n,sign) {
    if (FPSCR_PR==0) {
        if(sign == 0)  FR_HEX [n]   = 0x7f800000;
        else           FR_HEX [n]   = 0xff800000;
    } else {
        if(sign == 0)  FR_HEX [n]   = 0x7ff00000;
        else           FR_HEX [n]   = 0xfff00000;
        FR_HEX [n+1] = 0x00000000;
    }
}

void qnan(int n) {
    if (FPSCR_PR==0)  FR[n]   = 0x7fbfffff;
    else {            FR[n]   = 0x7ff7ffff;
    FR[n+1] = 0xffffffff;
    }
}

Example

An example is shown using assembler mnemonics, indicating the states before and after execution of the instruction.

Italics (e.g., .align) indicate an assembler control instruction. The meaning of the assembler control instructions is given below. For details, refer to the Cross-Assembler User’s Manual.

.org            Location counter setting
.data.w         Word integer data allocation
.data.l         Longword integer data allocation
.sdata          String data allocation
.align 2         2-byte boundary alignment
.align 4         4-byte boundary alignment
.align 32        32-byte boundary alignment
.arepeat 16      16-times repeat expansion
.arepeat 32      32-times repeat expansion
.aendr           Count-specification repeat expansion end

Note: SH Series cross-assembler version 1.0 does not support conditional assembler functions.
### 6.3 New Instructions

**6.3.1 BAND**

<table>
<thead>
<tr>
<th>Bit Logical AND</th>
<th>Bit Manipulation Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH-2A/SH2A-FPU (New)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAND.B #imm3, @(disp12, Rn)</td>
<td>(&lt;imm&gt; of (disp+Rn)) &amp; T → T</td>
<td>0011nnnn01ii10010100dddddddddd</td>
<td>3</td>
<td>Operation result</td>
</tr>
</tbody>
</table>

**Description**

ANDs a specified bit in memory at the address indicated by (disp + Rn) with the T bit, and stores the result in the T bit. The bit number is specified by 3-bit immediate data. With this instruction, data is read from memory as a byte unit.

\[
\text{BAND.B #imm3, @(disp12, Rn)}
\]
Operation

BANDM (long d, long i, long n) /*BAND.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp, assignbit;

    disp = (0x00000FFF & (long)d);
    imm = (0x00000007&(long)i);
    temp = (long) Read_Byte (R[n]+disp);
    assignbit = (0x00000001<<imm)&temp;
    if((T==0)||(assignbit==0)) T=0;
    else T=1;
    PC+=4;
}

Examples:

BAND.B #H'5,@(2,R0) ; Before execution: @(R0 + 2) = H'DF, T=1
; After execution: @(R0 + 2) = H'DF, T=0
### 6.3.2 BANDNOT Bit ANDNOT Bit Manipulation Instruction

**Bit NOT Logical AND**

**SH-2A/SH2A-FPU (New)**

#### Format

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANDNOT.B #imm3, @(disp12,Rn)</td>
<td>~(&lt;imm&gt; of (disp+Rn)) &amp; T</td>
<td>0011nnnn0i111001111000dddddddddd</td>
<td>3</td>
<td>Operation result</td>
</tr>
</tbody>
</table>

#### Description

ANDs the value obtained by inverting a specified bit of memory at the address indicated by (disp + Rn) with the T bit, and stores the result in the T bit. The bit number is specified by 3-bit immediate data. With this instruction, data is read from memory as a byte unit.

**BANDNOT.B #imm3, @(disp12, Rn)**

#### Operation

```
BANDNOTM (long d, long i, long n) /*BANDNOT.B #imm3, @(disp12, Rn) */
{
   long disp, imm, temp, assignbit;

   disp = (0x00000FFF & (long)d);
   imm = (0x00000007&(long)i);
   temp = (long) Read_Byte (R[n]+disp);
   assignbit = (0x00000001<<imm)&temp;
   if((T==1)&&(assignbit==0)) T=1;
   else T=0;
   PC+=4;
}
```
Examples:

BANDNOT.B #H'5, @(2, R0) ; Before execution: @(R0 + 2) = H'20, T = 1
; After execution: @(R0 + 2) = H'20, T = 0
6.3.3 BCLR Bit CLeaR Bit Manipulation Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCLR.B #imm3, @(disp12,Rn)</td>
<td>0 → (&lt;imm&gt; of (disp+Rn))</td>
<td>0011nnnn0i10010000ddddddd</td>
<td>3</td>
<td>—</td>
</tr>
<tr>
<td>BCLR #imm3, Rn</td>
<td>0 → &lt;imm&gt; of Rn</td>
<td>10000100nnnn0i1i</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Clears a specified bit of memory at the address indicated by (disp + Rn), or of the LSB 8 bits of a general register Rn. The bit number is specified by 3-bit immediate data. With the BCLR.B instruction, after data is read from memory as a byte unit, clearing of the specified bit is executed, and the resulting data is then written to memory as a byte unit.

BCLR.B #imm3, @(disp12, Rn)
Operation

BCLRM (long d, long i, long n) /*BCLR.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp;

disp = (0x00000FFF & (long)d);
imm= (0x00000007&(long)i);
temp= (long) Read_Byte (R[n]+disp);
temp&=(~(0x00000001<<imm));
Write_Byte (R[n]+disp, temp);
PC+=4;
}

BCLR (long i, long n) /*BCLR #imm3, Rn */
{
    long imm, temp;

imm= (0x00000007 & (long)i);
R[n]&=(~(0x00000001<<imm));
PC+=2;
}

Examples:

BCLR.B #H'5,@(2,R0) ; Before execution: @(R0 + 2) = H'FF
; After execution: @(R0 + 2) = H'DF

BCLR #H'4, R0 ; Before execution: @R0 = H'FFFFFFFF
; After execution: @R0 = H'FFFFFFFF
6.3.4 BLD Bit Load

**Bit Load SH-2A/SH2A-FPU (New)**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLD.B #imm3, @(disp12,Rn)</td>
<td>(&lt;imm&gt; of (disp+Rn)) → T</td>
<td>0011nnnn011110010011ddddddd3ddddd</td>
<td>3</td>
<td>Operation result</td>
<td></td>
</tr>
<tr>
<td>BLD #imm3, Rn</td>
<td>&lt;imm&gt; of Rn → T</td>
<td>1000011nnnnliii</td>
<td>1</td>
<td>Operation result</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Stores a specified bit of memory at the address indicated by (disp + Rn), or of the LSB 8 bits of a general register Rn, in the T bit. The bit number is specified by 3-bit immediate data. With the BLD.B instruction, data is read from memory as a byte unit.

BLD.B #imm3, @(disp12, Rn)

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specified by #imm3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(disp+Rn)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T
```

BLD #imm3, Rn

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower 8 bits specified by #imm3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Rev. 3.00 Jul 08, 2005 page 94 of 484
REJ09B0051-0300
Operation

BLDM (long d, long i, long n) /*BLD.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp, assignbit;

disp = (0x00000FFF & (long)d);
imm = (0x00000007 & (long)i);
temp = (long) Read_Byte (R[n]+disp);
assignbit=(0x00000001<<imm) & temp;
if(assignbit==0) T=0;
else T=1;
PC+=4;
}

BLD (long i, long n) /*BLD #imm3, Rn */
{
    long imm, assignbit;

    imm = (0x00000007 & (long)i);
assignbit=(0x00000001<<imm) & R[n];
if(assignbit ==0) T=0;
else T=1;
PC+=2;
}

Examples:

BLD.B #H'5,(2,R0) ; Before execution: @(R0 + 2) = H'20, T = 0
; After execution: @(R0 + 2) = H'20, T = 1

BLD #H'4,R0     ; Before execution: R0 = H'000000EF, T = 1
; After execution: R0 = H'000000EF, T = 0
6.3.5 BLDNOT Bit Load NOT Bit Manipulation Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLDNOT.B #imm3, @(disp12,Rn)</td>
<td>~ (&lt;imm&gt; of (disp+Rn)) → T</td>
</tr>
<tr>
<td></td>
<td>0011nnnn01iiii0011011ddddd0ddddd0</td>
</tr>
</tbody>
</table>

**Description**

Inverts a specified bit of memory at the address indicated by (disp + Rn), and stores the resulting value in the T bit. The bit number is specified by 3-bit immediate data. With the BLDNOT.B instruction, data is read from memory as a byte unit.

BLDNOT.B #imm3, @(disp12, Rn)

**Operation**

```c
BLDNOTM (long d, long i, long n) /*BLDNOT.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp,assignbit;

disp = (0x00000FFF & (long)d);
imm= (0x00000007&(long)i);
temp = (long) Read_Byte (R[n]+disp);
assignbit=(0x00000001<<imm)&temp;
if(assignbit==0) T=1;
else T=0;
PC+=4;
}
```
Examples:

BLDNOT.B #H'5,@(2,R0)  ; Before execution: @(R0 + 2) = H'20, T = 1
                           ; After execution: @(R0 + 2) = H'20, T = 0
### 6.3.6 BOR Bit OR Bit Manipulation Instruction

**SH-2A/SH2A-FPU (New)**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th>Operation result</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOR.B #imm3, @(disp12,Rn)</td>
<td><code>&lt;imm&gt;</code> of <code>(disp+Rn)) -&gt; T</code></td>
<td>0011nnnn01110010101ddddddd</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

ORs a specified bit in memory at the address indicated by `(disp + Rn)` with the T bit, and stores the result in the T bit. The bit number is specified by 3-bit immediate data. With this instruction, data is read from memory as a byte unit.

BOR.B #imm3, @(disp12, Rn)
Operation

BORM (long d, long i, long n) /*BOR.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp, assignbit;

    disp = (0x000000FF & (long)d);
    imm = (0x00000007&(long)i);
    temp = (long) Read_Byte (R[n]+disp);
    assignbit = (0x00000001<<imm)&temp;
    if((T==0)&(assignbit==0)) T=0;
    else T=1;

    PC+=4;
}

Examples:

BOR.B #H'50',(2,R0) ; Before execution: @R0,2 = H'20, T = 0
; After execution: @R0,2 = H'20, T = 1
6.3.7 BORNOT Bit ORNOT Bit Manipulation Instruction

Bit NOT Logical OR

SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th>Operation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>BORNOT.B #imm3, @(disp12,Rn)</td>
<td>~&lt;(imm) of (disp+Rn))</td>
<td>0011nnnn011101101dddddaddddd</td>
<td>3</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

ORs the value obtained by inverting a specified bit of memory at the address indicated by (disp + Rn) with the T bit, and stores the result in the T bit. The bit number is specified by 3-bit immediate data. With this instruction, data is read from memory as a byte unit.

BORNOT.B #imm3, @(disp12, Rn)
Operation

BORNOTM (long d, long i, long n) /*BORNOT.B #imm3, @(disp12, Rn) */
{
  long disp, imm, temp, assignbit;

  disp = (0x00000FFF & (long)d);
  imm= (0x00000007&(long)i);
  temp= (long) Read_Byte (R[n]+disp);
  assignbit =(0x00000001<<imm)&temp;
  if((T==1)||(assignbit==0)) T=1;
  else T=0;

  PC+=4;
}

Examples:

BORNOT.B #H'5,@(2,R0) ; Before execution: @(R0 + 2) = H'DF, T = 0
; After execution: @(R0 + 2) = H'DF, T = 1
### Description

Sets to 1 a specified bit of memory at the address indicated by (disp + Rn), or of the LSB 8 bits of a general register Rn. The bit number is specified by 3-bit immediate data. With the BSET.B instruction, after data is read from memory as a byte unit, the specified bit is set to 1, and the resulting data is then written to memory as a byte unit.

\[
\text{BSET.B } \#\text{imm3}, @(\text{disp12}, \text{Rn})
\]

\[
\text{Specified by } \#\text{imm3}
\]

\[
\text{(disp+Rn)}
\]

\[
\begin{array}{c}
\text{1} \\
\end{array}
\]

\[
\text{BSET } \#\text{imm3}, \text{Rn}
\]

\[
\begin{array}{c}
\text{31} \\
\text{7} \\
\text{0}
\end{array}
\]

\[
\text{Lower 8 bits specified by } \#\text{imm3}
\]

\[
\begin{array}{c}
\text{1}
\end{array}
\]
Operation

BSETM (long d, long i, long n) /*BSET.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp;

    disp = (0x00000FFF & (long)d);
    imm= (0x00000007&(long)i);
    temp= (long) Read_Byte (R[n]+disp);
    temp|=(0x00000001<<imm);
    Write_Byte (R[n]+disp, temp);
    PC+=4;
}

BSET (long i, long n) /*BSET #imm3, Rn */
{
    long imm, temp;

    imm= (0x00000007 &(long)i);
    R[n]|=(0x00000001<<imm);
    PC+=2;
}

Examples:

BSET.B #H'5,@(2,R0)  ; Before execution: @(R0 + 2) = H'00
                       ; After execution: @(R0 + 2) = H'20

BSET #H'4,R0          ; Before execution: R0 = H'00000000
                       ; After execution: R0 = H'00000010
### Description

Transfers the contents of the T bit to a specified 1-bit location of memory at the address indicated by (disp + Rn), or of the LSB 8 bits of a general register Rn. The bit number is specified by 3-bit immediate data. With the BST.B instruction, after data is read from memory as a byte unit, transfer from the T bit to the specified bit is executed, and the resulting data is then written to memory as a byte unit.

\[
\text{BST.B} \ #\text{imm3}, \ @(\text{disp12}, \ Rn)
\]

\[
\text{BST} \ #\text{imm3}, \ Rn
\]
Operation

BSTM (long d, long i, long n) /*BST.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp;

disp = (0x00000FFF & (long)d);
imm = (0x00000007&(long)i);
temp = (long) Read_Byte (R[n]+disp);
if (T==0) temp&=~(0x00000001<<imm));
else temp|=0x00000001<<imm);;
Write_Byte (R[n]+disp, temp);

PC+=4;
}

BST (long i, long n) /*BST #imm3, Rn */
{
    long disp, imm;

disp = (0x00000FFF & (long)d);
imm = (0x00000007&(long)i);
if (T==0) R[n]&=~(0x00000001<<imm));
else R[n]|=(0x00000001<<imm);;

PC+=2;
}

Examples:

BST.B #H'4,@(2,R0) ; Before execution: @(R0 + 2) = H'FF, T = 0
; After execution: @(R0 + 2) = H'EF, T = 0

BST #H'4,R0 ; Before execution: R0 = H'00000000, T = 1
; After execution: R0 = H'00000010, T = 1
6.3.10 BXOR Bit exclusive OR Bit Manipulation Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BXOR.B #imm3, @(disp12, Rn)</td>
<td>(&lt;imm&gt;) of (disp+Rn) (\wedge) T</td>
<td>0111nnnniii100101010dddddddddd</td>
<td>3</td>
<td>Operation result</td>
<td></td>
</tr>
</tbody>
</table>

Description

Exclusive-ORs a specified bit in memory at the address indicated by (disp + Rn) with the T bit, and stores the result in the T bit. The bit number is specified by 3-bit immediate data. With this instruction, data is read from memory as a byte unit.

BXOR.B #imm3, @(disp12, Rn)
Operation

BXORM (long d, long i, long n) /*BXOR.B #imm3, @(disp12, Rn) */
{
    long disp, imm, temp, assignbit;

    disp = (0x00000FFF & (long)d);
    imm = (0x00000007&(long)i);
    temp = (long) Read_Byte (R[n]+disp);
    assignbit = (0x00000001<<imm)&temp;
    if (assignbit==0)
    {
        if (T==0) T=0;
        else  T=1;
    }
    else
    {
        if (T==0) T=1;
        else  T=0;
    }
    PC+=4;
}

Examples:

BXOR.B #H'5,@(2,R0)  ; Before execution: @(R0 + 2) = H'FF, T = 1
; After execution: @(R0 + 2) = H'FF, T = 0
### Description

Determines saturation. Signed data is used with this instruction. The saturation upper-limit value is stored in general register Rn if the contents of Rn exceed the saturation upper-limit value, or the saturation lower-limit value is stored in Rn if the contents of Rn are less than the saturation lower-limit value, and the CS bit is set to 1. The saturation upper-limit value and lower-limit value for each instruction are shown in the table below.

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Saturation Lower-Limit Value</th>
<th>Saturation Upper-Limit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLIPS.B Rn</td>
<td>H'FFFFFF80</td>
<td>H'0000007F</td>
</tr>
<tr>
<td>2</td>
<td>CLIPS.W Rn</td>
<td>H'FFFF8000</td>
<td>H'000007FF</td>
</tr>
</tbody>
</table>

### Notes

The CS bit value does not change if the contents of general register Rn do not exceed the saturation upper-limit value or are not less than the saturation lower-limit value.
Operation

CLIPSB(long n) /* CLIPS.B Rn*/
{
    if ( R[n] > 0x0000007F)
    {
        R[n]=0x0000007F;
        CS=1;
    }
    else if (R[n] < 0xFFFFFFFF)
    {
        R[n]=0xFFFFFFFF;
        CS=1;
    }
    PC+2;
}

CLIPSW(long n) /* CLIPS.W Rn*/
{
    if ( R[n] > 0x00007FFF)
    {
        R[n]=0x00007FFF;
        CS=1;
    }
    else if (R[n] < 0xFFFF8000)
    {
        R[n]=0xFFFF8000;
        PC+2;
    }
}
Examples:

CLIPS.B R0 ; Before execution: R0 = H'0000000F, CS = 0
; After execution: R0 = H'0000000F, CS = 0

CLIPS.B R1 ; Before execution: R1 = H'00000080, CS = 0
; After execution: R1 = H'0000007E, CS = 1

CLIPS.W R0 ; Before execution: R0 = H'FFFFFFFF, CS = 0
; After execution: R0 = H'FFFFFFFF, CS = 0

CLIPS.W R1 ; Before execution: R1 = H'FFFFFF8000, CS = 0
; After execution: R1 = H'FFFFFF8000, CS = 1
### 6.3.12 CLIPU

**CLIP as Unsigned Arithmetic Instruction**

Unsigned Saturation Value Compare Instruction

<table>
<thead>
<tr>
<th>No.</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLIPU.B Rn</td>
<td>If Rn &gt; (saturation value), (saturation value) → Rn, 1 → CS</td>
<td>0100nnnn10000001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>CLIPU.W Rn</td>
<td>(saturation value)</td>
<td>0100nnnn10000101</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Determines saturation. Unsigned data is used with this instruction. If the contents of general register Rn exceed the saturation value, the saturation value is stored in Rn and the CS bit is set to 1. The saturation value for each instruction is shown in the table below.

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Saturation Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLIPU.B Rn</td>
<td>H'000000FF</td>
</tr>
<tr>
<td>2</td>
<td>CLIPU.W Rn</td>
<td>H'0000FFFF</td>
</tr>
</tbody>
</table>

**Notes**

The CS bit value does not change if the contents of general register Rn do not exceed the saturation upper-limit value.
Operation

CLIPUB(long n) /* CLIPU.B Rn*/
{
    if ( R[n] > 0x000000FF)
    {
        R[n]=0x000000FF;
        CS=1;
    }
    PC+2;
}

CLIPUW(long n) /* CLIPU.W Rn*/
{
    if ( R[n] > 0x0000FFFF)
    {
        R[n]=0x0000FFFF;
        CS=1;
    }
    PC+2;
}

Examples:

CLIPU.B R0 ; Before execution: R0 = H'0000000F, CS = 0
             ; After execution: R0 = H'0000000F, CS = 0

CLIPU.B R1 ; Before execution: R1 = H'00000100, CS = 0
             ; After execution: R1 = H'000000FF, CS = 1

CLIPU.W R0 ; Before execution: R0 = H'00000FFF, CS = 0
             ; After execution: R0 = H'000000FF, CS = 0

CLIPU.W R1 ; Before execution: R1 = H'00010000, CS = 0
             ; After execution: R1 = H'0000FFFF, CS = 1
**6.3.13 DIVS DIVide as Signed Instruction**

**Description**

Executes division of the 32-bit contents of a general register Rn (dividend) by the contents of R0 (divisor). This instruction executes signed division and finds the quotient only. A remainder operation is not provided. To obtain the remainder, find the product of the divisor and the obtained quotient, and subtract this value from the dividend. The sign of the remainder will be the same as that of the dividend.

**Notes**

An overflow exception will occur if the negative maximum value (H'00000000) is divided by –1. If division by zero is performed a division by zero exception will occur.

If an interrupt is generated while this instruction is being executed, execution will be halted. The return address will be the start address of this instruction, and this instruction will be re-executed.

**Operation**

```
DIVS (long n) /* DIVS R0, Rn */
{
   R[n]=R[n] / R[0];
   PC+=2;
}
```

**Examples:**

```
DIVS R0, R1 ; R1(32bits) / R0 (32bits) = R1(32bits); signed
```
### 6.3.14 DIVU  DIVide as Unsigned  Arithmetic Instruction

Unsigned Division

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVU R0, Rn</td>
<td>Unsigned, Rn ÷ R0 → Rn</td>
<td>0100nnnn10000100</td>
<td>34</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Executes division of the 32-bit contents of a general register Rn (dividend) by the contents of R0 (divisor). This instruction executes unsigned division and finds the quotient only. A remainder operation is not provided. To obtain the remainder, find the product of the divisor and the obtained quotient, and subtract this value from the dividend.

**Notes**

A division by zero exception will occur if division by zero is performed.

If an interrupt is generated while this instruction is being executed, execution will be halted. The return address will be the start address of this instruction, and this instruction will be re-executed.

**Operation**

```c
DIVU (long n) /* DIVU R0, Rn */
{
  (unsigned long) R[n] = (unsigned long)R[n] / (unsigned long)R[0];
  PC+=2;
}
```

**Examples:**

```c
DIVU R0, R1 ; R1(32bits) / R0(32bits) = R1(32bits); unsigned
```
## 6.3.15 FMOV Floating-point MOVe

<table>
<thead>
<tr>
<th>No.</th>
<th>SZ</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>FMOV.S FRm, @(disp12,Rn)</td>
<td>FRm → (disp×4+Rn)</td>
<td>0011nnnnnnnn00010011dddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>FMOV.D DRm, @(disp12,Rn)</td>
<td>DRm → (disp×8+Rn)</td>
<td>0011nnnnnnnn00010011dddddddddddd</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>FMOV.S @(disp12,Rm), FRn</td>
<td>(disp×4+Rm) → FRn</td>
<td>0011nnnnnnnn00010111dddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>FMOV.D @(disp12,Rm), DRn</td>
<td>(disp×8+Rm) → DRn</td>
<td>0011nnnnnnnn00010111dddddddddddd</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

### Description

1. Transfers FRm contents to memory at the address indicated by (disp + Rn).
2. Transfers DRm contents to memory at the address indicated by (disp + Rn).
3. Transfers memory contents at the address indicated by (disp + Rn) to FRn.
4. Transfers memory contents at the address indicated by (disp + Rn) to DRn.

### Note

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×4, ×8) as displacement values.

### Operation

```c
void FMOV_INDEX_DISP12_STORE(int m,n) /* FMOV.S FRm, @(disp12,Rn) */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    Write_Int ( R[n]+(disp<<2), FR[m]);
    PC +=4;
}

void FMOV_INDEX_DISP12_STORE_DR(int m,n) /* FMOV.D DRm, @(disp12,Rn) */
{
    long disp;

    disp = (0x00000FFF & (long)d);
```

---

Rev. 3.00 Jul 08, 2005 page 115 of 484

REJ09B0051-0300
Write_Quad (R[n]+(disp<<3), DR[m>>1]);
PC +=4;
}

void FMOV_INDEX_DISP12_LOAD(int m,n) /*FMOV.S @(disp12,Rm), FRn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    FR[n] = Read_Int (R[m]+(disp<<2));
    PC +=4;
}

void FMOV_INDEX_DISP12_LOAD_DR(int m,n) /*FMOV.D @(disp12,Rm), DRn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    DR[n>>1] = Read_Quad (R[m]+(disp<<3));
    PC +=4;
}
Examples:

FMOV.S FR0, @(2, R2) ; Before execution: FR0 = H'12345670
; After execution: @(R2 + 8) = H'12345670

FMOV.D DR0, @(2, R2) ; Before execution: FR0 = H'01234567
               FR1 = H'89ABCDEF
; After execution: @(R2 + 16) = H'01234567
               @(R2 + 20) = H'89ABCDEF

FMOV.S @(2, R2), FR0 ; Before execution: @(R2 + 8) = H'12345670
; After execution: FR0 = H'12345670

FMOV.D @(2, R2), DR0 ; Before execution: @(R2 + 16) = H'01234567
               @(R2 + 20) = H'89ABCDEF
; After execution: FR0 = H'01234567
               FR1 = H'89ABCDEF
6.3.16 JSR/N Jump to SubRoutine with No delay slot Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR/N @Rm</td>
<td>PC - 2 → PR, Rm → PC</td>
<td>0100mmmm01001011</td>
<td>3</td>
<td>—</td>
</tr>
<tr>
<td>JSR/N @@(disp8, TBR)</td>
<td>PC - 2 → PR, (disp×4+TBR) → PC</td>
<td>10000011dddddddd</td>
<td>5</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Branches to a subroutine procedure at the designated address. The contents of PC are stored in PR and execution branches to the address indicated by the contents of general register Rm as 32-bit data or to the address read from memory address (disp × 4 + TBR). The stored contents of PC indicate the starting address of the second instruction after the present instruction. This instruction is used with RTS as a subroutine procedure call.

Notes

This is not a delayed branch instruction.

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×4) as displacement values.
Operation

JSRN (long m) /* JSR/N @Rm, */
{
    unsigned long temp;

    temp=PC;
    PR=PC-2;
    PC=R[m]+4;
}

JSRNM (long d) /* JSR/N @@(disp8, TBR) */
{
    unsigned long temp;
    long disp;

    temp=PC;
    PR=PC-2;
    disp=(0x000000FF & d);
    PC=Read_Long(TBR+(disp<<2))+4;
}
Examples:

```
MOV.L  JSRN_TABLE,R0       ; R0 = TRGET address
JSR/N  @R0                  ; Branch to TRGET.
ADD    R0,R1                ; ← Procedure return destination
                        (PR contents)
                        . . . . . . .
.align 4
JSRN_TABLE: .data.1    TRGET ; Jump table
TRGET:      NOP            ; ← Entry to procedure
            MOV    R2,R3
            RTS/N        ; Return to above ADD instruction.
                        . . . . . . .
TBR+H‘08   .data.1    FFFF7F80 ;
                        . . . . . . .
JSR/N    @@(2,TBR)        ; Branch to address stored in address TBR + H’08
ADD    R0,R1                ; ← Procedure return destination
                        (PR contents)
                        . . . . . . .
            FFFF7F80    NOP            ; ← Entry to procedure
            FFFF7F82    MOV    R2,R3
            FFFF7F84    RTS/N        ; Return to above ADD instruction.
```
6.3.17 LDBANK Load register BANK System Control Instruction
Transfer to Specified Register Bank Entry
SH-2A/SH2A-FPU (New)

Format | Abstract | Code | Cycle | T Bit
--- | --- | --- | --- | ---
LDBANK @Rm, R0 (Specified register bank entry) → R0 | 0100mmmm11100101 | 6 | —

Description
The register bank entry indicated by the contents of general register Rm is transferred to general register R0. The register bank number and register stored in the bank are specified by general register Rm.

![Diagram of register bank and entry fields]
Note
The architecture supports a maximum of 512 banks. However, the number of banks differs depending on the product.

Operation

LDBANK (long m) /*LDBANK @Rm, R0 */
{
    R[0]=Read_Bank_Long(R[m]);
    PC+=2;
}

Examples:

LDBANK @R1,R0 ; Before execution: R1 = H'00000108
                   ; After execution: R0 = Contents of R2 stored in R0 = bank 2
6.3.18 LDC Load to Control Register

**System Control Instruction**

SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC Rm, TBR</td>
<td>Rm → TBR</td>
<td>0100mmmm01001010</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Stores a source operand in control register TBR.

**Operation**

```c
LDCTBR (long m) /* LDC Rm, TBR*/
{
    TBR=R[m];
    PC+=2;
}
```

**Examples:**

```assembly
LDC R0, TBR
; Before execution: R0 = H'12345678, TBR = H'00000000
; After execution: TBR = H'12345678
```
6.3.19 MOV MOVe structure data Data Transfer Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.B Rm, @(disp12,Rn)</td>
<td>Rm → (disp+Rn)</td>
<td>0011nnnnnnnnnn00010000dddddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W Rm, @(disp12,Rn)</td>
<td>Rm → (disp×2+Rn)</td>
<td>0011nnnnnnnnnn00010001dddddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L Rm, @(disp12,Rn)</td>
<td>Rm → (disp×4+Rn)</td>
<td>0011nnnnnnnnnn00010010dddddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B @(disp12,Rm), Rn</td>
<td>(disp+Rm) → sign extension → Rn</td>
<td>0011nnnnnnnnnn00010100dddddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @(disp12,Rm), Rn</td>
<td>(disp×2+Rm) → sign extension → Rn</td>
<td>0011nnnnnnnnnn00010101dddddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @(disp12,Rm), Rn</td>
<td>(disp×4+Rm) → Rn</td>
<td>0011nnnnnnnnnn00010110dddddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Transfers a source operand to a destination. This instruction is ideal for data access in a structure or the stack.

**Note**

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×1, ×2, ×4) as displacement values.

**Operation**

```c
MOVBS12 (long d, long m, long n) /* MOV.B Rm, @(disp12,Rn) */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    Write_Byte(R[n]+disp,R[m]);
    PC+=4;
}

MOVWS12 (long d, long m, long n) /* MOV.W Rm, @(disp12,Rn) */
{
    long disp;

    disp = (0x00000FFF & (long)d);
```

Rev. 3.00  Jul 08, 2005  page 124 of 484
REJ09B0051-0300
Write_Word(R[n]+(disp<<1),R[m]);
PC+=4;
}

MOVLS12 (long d, long m, long n) /* MOV.L Rm, @(disp12,Rn) */
{
    long disp;
    disp = (0x00000FFF & (long)d);
    Write_Long(R[n]+(disp<<2), R[m]);
    PC+=4;
}

MOVBL12 (long d, long m, long n) /* MOV.B @(disp12,Rm), Rn */
{
    long disp;
    disp = (0x00000FFF & (long)d);
    R[n]=Read_Byte(R[m]+disp);
    if (( R[n] & 0x80 ) ==0) R[n] &=0x000000FF;
    else R[n] |=0xFFFFFF00;
    PC+=4;
}

MOVWL12 (long d, long m, long n) /* MOV.W @(disp12,Rm), Rn */
{
    long disp;
    disp = (0x00000FFF & (long)d);
    R[n]=Read_Word(R[m]+(disp<<1));
    if ((R[n] & 0x8000) ==0) R[n] &=0x0000FFFF;
    else R[n] |=0xFFFF0000;
    PC+=4;
}
MOVLL12 (long d, long m, long n) /* MOV.L @(disp12,Rm), Rn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    R[n]=Read_Long(R[m]+(disp<<2));
    PC+=4;
}

Examples:

MOV.B R0,@(1,R1)  ; Before execution: R0 = H'FFFF7F80
                  ; After execution: @(R1 + 1) = H'80

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

Data Transfer Instruction

SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.B</td>
<td>R0, @Rn+</td>
<td>R0 → (Rn), Rn + 1 → Rn</td>
<td>0100nnnn10001011</td>
<td>1</td>
</tr>
<tr>
<td>MOV.W</td>
<td>R0, @Rn+</td>
<td>R0 → (Rn), Rn + 2 → Rn</td>
<td>0100nnnn10011011</td>
<td>1</td>
</tr>
<tr>
<td>MOV.L</td>
<td>R0, @Rn+</td>
<td>R0 → (Rn), Rn + 4 → Rn</td>
<td>0100nnnn10101011</td>
<td>1</td>
</tr>
<tr>
<td>MOV.B</td>
<td>@-Rm, R0</td>
<td>Rm - 1 → Rm (Rm) → sign extension → R0</td>
<td>0100mmmm11001011</td>
<td>1</td>
</tr>
<tr>
<td>MOV.W</td>
<td>@-Rm, R0</td>
<td>Rm - 2 → Rm (Rm) → sign extension → R0</td>
<td>0100mmmm11011011</td>
<td>1</td>
</tr>
<tr>
<td>MOV.L</td>
<td>@-Rm, R0</td>
<td>Rm - 4 → Rm (Rm) → R0</td>
<td>0100mmmm11101011</td>
<td>1</td>
</tr>
</tbody>
</table>

Description

Transfers a source operand to a destination.

Operation

MOVRSBP (long n) /* MOV.B R0, @Rn+/ 

{ 
    Write_B (n, R[0]);
    R[n]+=1;
    PC+=2;
}

MOVRSWP (long n) /* MOV.W R0, @Rn+/ 

{ 
    Write_W (n, R[0]);
    R[n]+=2;
    PC+=2;
}
MOVRLP (long n)  /* MOV.L R0, @Rn*/
{
    Write_Long(R[n], R[0]);
    R[n]+=4;
    PC+=2;
}

MOVRSLM (long m)  /* MOV.L @-Rm, R0*/
{
    R[m]-=4;
    R[0]=Read_Long (R[m]);
    PC+=2;
}

MOVRSBM (long m)  /* MOV.B @-Rm, R0*/
{
    R[m]-=1;
    R[0]=(long) Read_Word (R[m]);
    if ((R[0]&0x80)==0)  R[0] &= 0x000000FF;
    else  R[0] |= 0xFFFF0000;
    PC+=2;
}

MOVRSWM (long m)  /* MOV.W @-Rm, R0*/
{
    R[m]-=2;
    R[0]=(long) Read_Word (R[m]);
    if ((R[0]&0x8000)==0)  R[0] &= 0x0000FFFF;
    else  R[0] |= 0xFFFF0000;
    PC+=2;
}

MOVRSLM (long m)  /* MOV.L @-Rm, R0*/
{
    R[m]-=4;
    R[0]=Read_Long (R[m]);
    PC+=2;
}
Examples:

```
MOV.B R0, @R1+ ; Before execution: R0 = H'AAAAAAAA, R1 = FFFF7F80
    ; After execution: R1 = H'FFFF7F81, @(H'FFFF7F80) = H'AA

MOV.L @-R1, R0 ; Before execution: R1 = H'12345678
    ; After execution: R1 = H'12345674, R0 = @(H'12345674)
```
6.3.21 MOVI20 MOVe Immediate 20bits data Data Transfer Instruction

20-Bit Immediate Data Transfer

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVI20  #imm20, Rn</td>
<td>imm → sign extension → Rn</td>
<td>0000nnnniii0000iiiiiiiiiiiiiiii</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Stores immediate data that has been sign-extended to longword in general register Rn.

**Operation**

MOVI20 (long i, long n) /* MOVI20 #imm, Rn */
{
    if (i&0x00080000) ==0) R[n]= (0x000FFFFF & (long) i);
    else R[n]=(0xFFF00000 | (long) i);

    PC+=4;
}

**Examples:**

MOVI20 H’7FFFFF,R0 ; Before execution: R0 = H’00000000
                  ; After execution: R0 = H’00007FFFF
### 6.3.22 MOVI20S

**MOVe Immediate 20bits data and 8bits Shift left**

- **Data Transfer Instruction**
- **20-Bit Immediate Data Transfer and 8-Bit Left-Shift**
- **SH-2A/SH2A-FPU (New)**

#### Format
<table>
<thead>
<tr>
<th>MOVI20S</th>
<th>#imm20, Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm&lt;&lt;8 → sign extension → Rn</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>0000nnniiii0001iiiiiiiiiiiiiiii</td>
</tr>
<tr>
<td>Cycle</td>
<td>1</td>
</tr>
<tr>
<td>T Bit</td>
<td>—</td>
</tr>
</tbody>
</table>

#### Description
Shifts immediate data 8 bits to the left and performs sign extension to longword, then stores the resulting data in general register Rn. Using an OR or ADD instruction as the next instruction enables a 28-bit absolute address to be generated. See section Appendix B, Programming Guidelines, for details.

![Diagram of MOVI20S instruction](image)

#### Note
For the Renesas Technology Super H RISC engine assembler, declarations should use immediate data that has been shifted 8 bits to the left.
Operation

MOVI20S (long i, long n)  /* MOVI20S #imm, Rn */
{
  if (i&0x00080000) ==0) R[n]= (0x000FFFFF & (long) i);
  else R[n]=(0xFFF00000 | (long) i);
  R[n]<<=8;
  PC+=4;
}

Examples:

MOVI20S H'7FFF,R0  ; Before execution: R0 = H'00000000
                     ; After execution:  R0 = H'07FFFF00
### 6.3.23 MOVML.L

**MOVe Multi-register Lower part**

R0-Rn Register Save/Restore Instruction

**Data Transfer Instruction**

SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVML.L Rm, @-R15</td>
<td>R15 - 4 → R15, Rm → (R15)</td>
<td>0100mmmm11110001</td>
<td>1 to 16</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>R15 - 4 → R15, Rm → 1 → (R15)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R15 - 4 → R15, R0 → (R15)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td>When Rm = R15, read Rm as PR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVML.L @R15+, Rn</td>
<td>(R15) → R0, R15 + 4 → R15</td>
<td>0100nnnn11110101</td>
<td>1 to 16</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>(R15) → R1, R15 + 4 → R15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(R15) → Rn, R15 + 4 → R15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td>When Rn = R15, read Rn as PR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Transfers a source operand to a destination. This instruction performs transfer between a number of general registers (R0 to Rn/Rm) not exceeding the specified register number and memory with the contents of R15 as its address.

If R15 is specified, PR is transferred instead of R15. That is, when nnnn(mmmm) = 1111 is specified, R0 to R14 and PR are the general registers subject to transfer.

**Operation**

```c
MOVMLML (long m) /*MOVML.L Rm, @-R15*/
{
    long i;

    for (i=m; i>=0; i--)
    {
        if (i==15)
        {
            Write_Long (R[15]-4, PR);
            R[15]-=4;
        }
        else
```
{  
    Write_Long (R[15]-4, R[i]);  
    R[15]-=4;  
    }  
  
PC+=2;  
}  

MOVLPML (long n) /*MOVML.L @R15+, Rn */ 
{  
    int i;  
    
for (i=0; i≤n; i++)  
{    
    if (i==15)  
    {  
        PR=Read_Long (R[15]);  
    }  
    else  
    {  
        R[i] = Read_Long (R[15]);  
    }  
    R[15]+=4;  
    }  
PC+=2;  
}
Examples:

```
MOVML. L R7, @-R15 ; Before execution: R15 = H'FFFF7F80
                    R0 = H'00000000, R1 = H'11111111
                    R2 = H'22222222, R3 = H'33333333
                    R4 = H'44444444, R5 = H'55555555
                    R6 = H'66666666, R7 = H'77777777

                    ; After execution: R15 = H'FFFF7F60
                    @(H'FFFF7F60) = H'77777777
                    @(H'FFFF7F64) = H'66666666
                    @(H'FFFF7F68) = H'55555555
                    @(H'FFFF7F6C) = H'44444444
                    @(H'FFFF7F70) = H'33333333
                    @(H'FFFF7F74) = H'22222222
                    @(H'FFFF7F78) = H'11111111
                    @(H'FFFF7F7C) = H'00000000

MOVML. L @R15+, R7 ; Before execution: R15 = H'FFFF7F60
                    @(H'FFFF7F60) = H'00000000
                    @(H'FFFF7F64) = H'11111111
                    @(H'FFFF7F68) = H'22222222
                    @(H'FFFF7F6C) = H'33333333
                    @(H'FFFF7F70) = H'44444444
                    @(H'FFFF7F74) = H'55555555
                    @(H'FFFF7F78) = H'66666666
                    @(H'FFFF7F7C) = H'77777777

                    ; After execution: R15 = H'FFFF7F80
                    R0 = H'00000000, R1 = H'11111111
                    R2 = H'22222222, R3 = H'33333333
                    R4 = H'44444444, R5 = H'55555555
                    R6 = H'66666666, R7 = H'77777777
```
6.3.24 MOVUM.U. MOVe Multi-register Upper part Data Transfer Instruction
Rn-R14, PR Register Save/Restore Instruction
SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVUM.L Rm, @-R15</td>
<td>R15 - 4 → R15, PR → (R15) R15 - 4 → R15, R14 → (R15) R15 - 4 → R15, Rm → (R15)</td>
<td>0100mmmm1110000</td>
<td>1 to 16</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Note: When Rm = R15, read Rm as PR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVUM.L @R15+, Rn</td>
<td>(R15) → Rn, R15 + 4 → R15 (R15) → Rn + 1, R15 + 4 → R15 (R15) → R14, R15 + 4 → R15 (R15) → PR, R15 + 4 → R15</td>
<td>0100nnnn11110100</td>
<td>1 to 16</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Note: When Rn = R15, read Rn as PR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description
Transfers a source operand to a destination. This instruction performs transfer between a number of general registers (Rn/Rm to R14, PR) not lower than the specified register number and memory with the contents of R15 as its address.

If R15 is specified, PR is transferred instead of R15.

Operation

```c
MOVUMU (long m) /*MOVUM.L Rm, @-R15 */
{
    int i;

    Write_Long (R[15]-4, PR);
    R[15]=4;

    for (i = 14; i>=m; i--)
    {
        Write_Long (R[15]-4, R[i]);
        R[15]=4;
    }
}
```
PC+=2;
}

MOVLPMU (long n) /*MOVU.L @R15+, Rn*/
{
int i;

for (i=n; i≤14; i++)
{
    R[i] = Read_Long (R[15]);
    R[15]+=4;
}
PR=Read_Long (R[15]);
R[15]+=4;
PC+=2;
}
Examples:

**MOVMU. L R8,@-R15**

; Before execution:

R15 = H'FFFF7F80
R8 = H'88888888, R9 = H'99999999
R10 = H'AAAAAAAA, R11 = H'BBBBBBBB
R12 = H'CCCCCCCC, R13 = H'BBBBBBBB
R14 = H'EEEEEEEE, PR = H'FFFFFFF0

; After execution:

R15 = H'FFFF7F60
@(H'FFFF7F60) = H'88888888
@(H'FFFF7F64) = H'99999999
@(H'FFFF7F68) = H'AAAAAAAA
@(H'FFFF7F6C) = H'BBBBBBBB
@(H'FFFF7F70) = H'CCCCCCCC
@(H'FFFF7F74) = H'DDDDDDDD
@(H'FFFF7F78) = H'EEEEEEEE
@(H'FFFF7F7C) = H'FFFFFFF0

**MOVMU. L @R15+,R8**

; Before execution:

R15 = H'FFFF7F60
@(H'FFFF7F60) = H'88888888
@(H'FFFF7F64) = H'99999999
@(H'FFFF7F68) = H'AAAAAAAA
@(H'FFFF7F6C) = H'BBBBBBBB
@(H'FFFF7F70) = H'CCCCCCCC
@(H'FFFF7F74) = H'DDDDDDDD
@(H'FFFF7F78) = H'EEEEEEEE
@(H'FFFF7F7C) = H'FFFFFFF0

; After execution:

R15 = H'FFFF7F80
R8 = H'88888888, R9 = H'99999999
R10 = H'AAAAAAAA, R11 = H'BBBBBBBB
R12 = H'CCCCCCCC, R13 = H'DDDDDDDD
R14 = H'EEEEEEEE, PR = H'FFFFFFF0
6.3.25 MOVRT MOVe Reverse Tbit Data Transfer Instruction
T Bit Reverse Rn Transfer SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVRT</td>
<td>Rn</td>
<td>~ T → Rn</td>
<td>0000nnnn001111001</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Reverses the T bit and then stores the resulting value in general register Rn. The value of Rn is 0 when T = 1 and 1 when T = 2.

**Operation**

```c
MOVRT (long n) /*MOVRT Rn */
{
    if (T ==1) R[n] = 0x00000000;
    else R[n] = 0x00000001;
    PC+=2;
}
```

**Examples:**

- `XOR R2, R2` ; R2 = 0
- `CMP/PZ R2` ; T = 1
- `MOVRT R0` ; R0 = 0
- `CLRT` ; T = 0
- `MOVRT R1` ; R1 = 1
### 6.3.26 MOVU

**MOVe structure data as Unsigned Data Transfer Instruction**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVU.B @(disp12,Rm), Rn</td>
<td>(disp+Rm) → zero extension → Rn</td>
<td>0011nnnnnnnn0001000ddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOVU.W @(disp12,Rm), Rn</td>
<td>(disp×2+Rm) → zero extension → Rn</td>
<td>0011nnnnnnnn0001001ddddddddddd</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Transfers a source operand to a destination, performing unsigned data transfer. This instruction is ideal for data access in a structure or the stack.

**Note**

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×1, ×2) as displacement values.
Operation

MOVBU12 (long d, long m, long n) /* MOVU.B @(disp12,Rm), Rn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    R[n]=Read_Byte(R[m]+disp);
    R[n] &=0x000000FF;
    PC+=4;
}

MOVWUL12 (long d, long m, long n) /* MOVU.W @(disp12,Rm), Rn */
{
    long disp;

    disp = (0x00000FFF & (long)d);
    R[n]=Read_Word(R[m]+(disp<<1));
    R[n] &=0x0000FFFF;
    PC+=4;
}

Examples:

MOVU.B @(2,R0),R1 ; Before execution: @(R0 + 2) = H'FF
                    ; After execution: R1 = H'000000FF

MOVU.W @(2,R0),R1 ; Before execution: @(R0 + 4) = H'FFFF
                    ; After execution: R1 = H'0000FFFF
6.3.27 MULR MULtiply to Register
Rn Result Storage Signed Multiplication

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULR R0,Rn</td>
<td>R0 × Rn → Rn</td>
<td>0100nnnn10000000</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

Description
Performs 32-bit multiplication of the contents of general register R0 by Rn, and stores the lower 32 bits of the result in general register Rn.

Operation

```c
MULR (long n) /* MULR R0, Rn */
{
    R[n] = R[0]*R[n];
    PC+=2;
}
```

Examples:

```
MULR R0,R1 ; Before execution: R0 = H'FFFFFFFFFFE, R1 = H'00005555
            ; After execution: R1 = H'FFFF5556
```
### 6.3.28 NOTT NOT Tbit Data Transfer Instruction

**T Bit Inversion and Transfer**

**SH-2A/SH2A-FPU (New)**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTT</td>
<td>~ T → T</td>
<td>0000000011101000</td>
<td>1</td>
<td>Operation result</td>
</tr>
</tbody>
</table>

**Description**

Inverts the T bit, then stores the resulting value in the T bit.

**Operation**

```plaintext
NOTT (long n)  /*NOTT Rn */
{
    if (T ==1) T=0;
    else T=1;
    PC+=2;
}
```

**Examples:**

- SETT ;T = 1
- NOTT ;T = 0
- NOTT ;T = 1
### PREFetch data to cache

#### Data Transfer Instruction

SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREF @Rn</td>
<td>Prefetch cache block</td>
<td>00000nnnn10000011</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

#### Description

Reads a 16-byte data block starting at a 16-byte boundary into the operand cache.

Address related errors are not generated for this instruction. In the event of an error, this instruction is handled as an NOP (no operation) instruction.

#### Note

On products with no cache, this instruction is handled as a NOP instruction.

#### Operation

```
PREF (long n) /* PREF @Rn */
{
    PC+=2;
}
```

#### Examples:

```
MOV.L SOFT_PF,R1  ; R1 address is SOFT_PF
PREF    @R1       ; Load SOFT_PF data into internal data cache

.align 16

SOFT_PF:    .data.w H'1234
            .data.w H'5678
            .data.w H'9ABC
            .data.w H'DEF0
```
6.3.30 RESBANK REStore from registerBANK System Control Instruction
Register Restoration from Register Bank
SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESBANK</td>
<td>Restoration from register bank</td>
<td>0000000001011011</td>
<td>9*</td>
<td>—</td>
</tr>
</tbody>
</table>

Note: * 19 when a bank overflow has occurred and the register is restored from the stack

Description
Restores the last register saved to a register bank.

Operation
```c
RESBANK( ) /*RESBANK */
    /*m = (Number of register bank to which a save was last performed)*/
    {
        int m;

        if(BO==0)
        {
            PR = Register_Bank[m].PR_BANK;
            GBR = Register_Bank[m].GBR_BANK;
            MACL = Register_Bank[m].MACL_BANK;
            MACH = Register_Bank[m].MACH_BANK;
            for (i=14; i≥14; i++)
                i≥0; i--
                {
                    R[i] = Register_Bank[m].R_BANK[i];
                }
        }
        else
        {
            for (i=0; i≤14; i++)
                {
                    R[i] = Read_Long(R[15]);
                    R[15] += 4;
                }
            if(BO==1)
                {
```
Examples:

RESBANK ; Recover register from register bank.
RTE ; Return to original routine.
ADD #8,R14 ; Executed before branch.
6.3.31 RTS/N ReTurn from Subroutine with No delay slot Branch Instruction

Return from Subroutine Procedure with No Delay Slot SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS/N</td>
<td>PR → PC</td>
<td>0000000011101011</td>
<td>3</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Performs a return from a subroutine procedure. That is, the PC is restored from PR, and processing is resumed from the address indicated by the PC. This instruction enables a return to be made from a subroutine procedure called by a BSR or JSR instruction to the origin of the call.

**Note**

This is not a delayed branch instruction.

**Operation**

```c
RTSN ( ) /* RTS/N */
{
    PC=PR+4;
}
```

**Examples:**

```assembly
MOV.L TABLE, R3 ; R0 = TRGET address
JSR/N @R3 ; Branch to TRGET.
ADD R0, R1 ; ← Procedure return destination
            (PR contents)

TABLE: .data .i TRGET ; Jump table

TRGET: NOP ; ← Entry to procedure

MOV R2, R3 ;
RTS/N ; Return to above ADD instruction.
```
6.3.32 RTV/N Return to Value and from subroutine with No delay slot Branch Instruction

Return from Subroutine Procedure with Register Value Transfer and with No Delay Slot

SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTV/N Rm</td>
<td>Rm → R0, PR → PC</td>
<td>0000mmmm01111011</td>
<td>3</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Performs a return from a subroutine procedure after a transfer from specified general register Rm to R0. That is, after the Rm value is stored in R0, the PC is restored from PR, and processing is resumed from the address indicated by the PC. This instruction enables a return to be made from a subroutine procedure called by a BSR or JSR instruction to the origin of the call.

**Note**

This is not a delayed branch instruction.

**Operation**

```c
RTVN (int m) /* RTV/N Rm */
{
    R[0]=R[m];
    PC=PR+4;
}
```
Examples:

```
MOV.L TABLE, R3 ; R0 = TRGET address
JSR/N @R3 ; Branch to TRGET.
ADD R0, R1 ; ← Procedure return destination
            (PR contents)

TABLE: .data.1 TRGET ; Jump table

TRGET: NOP ; ← Entry to procedure
        MOV #12, R3 ; R3 = H'00000012
        RTV/N R3 ; Return to above ADD instruction.
            ; R0 = H'00000012
```
### 6.3.33 SHAD Shift Arithmetic Dynamically

**Dynamic Arithmetic Shift**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAD Rm, Rn</td>
<td>When Rm ≥ 0, Rn&lt;&lt;Rm → Rn</td>
<td>0100nnnnmmmm1100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>When Rm &lt; 0, Rn&gt;&gt;</td>
<td>Rm</td>
<td>→ [MSB → Rn]</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Shifts the contents of general register Rn arithmetically. General register Rm specifies the shift direction and number of bits to be shifted.

A left shift is performed when the Rm register value is positive, and a right shift when negative. In a right shift, the MSB is added at the upper end.

The number of bits to be shifted is specified by the lower 5 bits (bits 4 to 0) of register Rm. If the value is negative (MSB = 1), the Rm register value is expressed as a two's complement. Therefore, the shift amount in a right shift is the value obtained by adding 1 to the inverse of the lower 5 bits of register Rm. The shift amount is 0 to 31 in a left shift, and 1 to 32 in a right shift.
Operation

```
SHAD (int m,n) /* SHAD Rm,Rn */
{
    int sgn = R[m] & 0x80000000;
    if  (sgn == 0)
        R[n] <<= (R[m] & 0x0000001F);
    else if  ((R[m] & 0x0000001F) == 0)
    {
        if ((R[n] & 0x80000000) == 0)
            R[n] = 0;
        else
            R[n]=0xFFFFFFFF;
    }
    else
        R[n]=(long)R[n] >> ((~R[m] & 0x0000001F)+1);
    PC+=2;
}
```

Examples:

```
SHAD R1, R2
; Before execution: R1 = H'FFFFFFEC, R2 = H'80180000
; After execution: R1 = H'FFFFFFEC, R2 = H'FFFFF801

SHAD R3, R4
; Before execution: R3 = H'000000014, R2 = H'FFFFF801
; After execution: R3 = H'000000014, R2 = H'80100000
```
### 6.3.34 SHLD

**Shift Logical Dynamically**

**Shift Instruction**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLD Rm, Rn</td>
<td>When Rm ≥ 0, Rn&lt;&lt;Rm → Rn</td>
<td>0100nnnnnnnnn1101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>When Rm &lt; 0, Rn&gt;&gt;</td>
<td>Rm</td>
<td>→ [0 → Rn]</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Shifts the contents of general register Rn logically. General register Rm specifies the shift direction and number of bits to be shifted.

A left shift is performed when the Rm register value is positive, and a right shift when negative. In a right shift, 0 is added at the upper end.

The number of bits to be shifted is specified by the lower 5 bits (bits 4 to 0) of register Rm. If the value is negative (MSB = 1), the Rm register value is expressed as a two's complement. Therefore, the shift amount in a right shift is the value obtained by adding 1 to the inverse of the lower 5 bits of register Rm. The shift amount is 0 to 31 in a left shift, and 1 to 32 in a right shift.
Operation

\texttt{SHLD (int m,n) // SHLD Rm,Rn */}
\begin{verbatim}
  
  int  sgn = R[m] & 0x80000000;
  if  (sgn == 0)
      R[n] <<= (R[m] & 0x0000001F);
  else if ((R[m] & 0x0000001F) == 0)
      R[n] = 0;
  else
      R[n]=\text{(unsigned)}R[n] >> (\text{~R[m]} & 0x0000001F)+1);
  PC+=2;
\end{verbatim}

Examples:

\texttt{SHLD R1, R2}  ; Before execution:  R1 = H'FFFFFFEC,  R2 = H'80180000  
           ; After execution:  R1 = H'FFFFFECE,  R2 = H'00080000
\texttt{SHLD R3, R4}  ; Before execution:  R3 = H'00000014,  R2 = H'FFFFF801  
           ; After execution:  R3 = H'00000014,  R2 = H'80100000
### 6.3.35 STBANK STore register BANK System Control Instruction

**Register Save to Specified Bank Entry**

**SH-2A/SH2A-FPU (New)**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBANK R0, @Rn</td>
<td>R0 → (specified register bank entry)</td>
<td>0100nnnn11100001</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

#### Description

R0 is transferred to the register bank entry indicated by the contents of general register Rn. The register bank number and register stored in the bank are specified by general register Rn.

---

**BN**

- BN: Bank number field
- EN: Entry number field

**Register Bank**

<table>
<thead>
<tr>
<th>BN</th>
<th>Register Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>Bank 0</td>
</tr>
<tr>
<td>0000000001</td>
<td>Bank 1</td>
</tr>
<tr>
<td>0000000010</td>
<td>Bank 2</td>
</tr>
<tr>
<td>0000000111</td>
<td>Bank 3</td>
</tr>
<tr>
<td>0000001000</td>
<td>Bank 4</td>
</tr>
<tr>
<td>0000001010</td>
<td>Bank 5</td>
</tr>
<tr>
<td>0000001100</td>
<td>Bank 6</td>
</tr>
<tr>
<td>0000001111</td>
<td>Bank 7</td>
</tr>
<tr>
<td>0000010000</td>
<td>Bank 8</td>
</tr>
<tr>
<td>0000010010</td>
<td>Bank 9</td>
</tr>
<tr>
<td>0000010100</td>
<td>Bank 10</td>
</tr>
<tr>
<td>0000010111</td>
<td>Bank 11</td>
</tr>
<tr>
<td>0000011000</td>
<td>Bank 12</td>
</tr>
<tr>
<td>0000011011</td>
<td>Bank 13</td>
</tr>
<tr>
<td>0000011100</td>
<td>Bank 14</td>
</tr>
</tbody>
</table>

**Entry in Register Bank**

<table>
<thead>
<tr>
<th>EN</th>
<th>Entry in Register Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>R0</td>
</tr>
<tr>
<td>00001</td>
<td>R1</td>
</tr>
<tr>
<td>00010</td>
<td>R2</td>
</tr>
<tr>
<td>00011</td>
<td>R3</td>
</tr>
<tr>
<td>00100</td>
<td>R4</td>
</tr>
<tr>
<td>00101</td>
<td>R5</td>
</tr>
<tr>
<td>00110</td>
<td>R6</td>
</tr>
<tr>
<td>00111</td>
<td>R7</td>
</tr>
<tr>
<td>01000</td>
<td>R8</td>
</tr>
<tr>
<td>01001</td>
<td>R9</td>
</tr>
<tr>
<td>01010</td>
<td>R10</td>
</tr>
<tr>
<td>01011</td>
<td>R11</td>
</tr>
<tr>
<td>01100</td>
<td>R12</td>
</tr>
<tr>
<td>01101</td>
<td>R13</td>
</tr>
<tr>
<td>01110</td>
<td>R14</td>
</tr>
<tr>
<td>01111</td>
<td>MACH</td>
</tr>
<tr>
<td>10000</td>
<td>Interrupt vector offset</td>
</tr>
<tr>
<td>10001</td>
<td>PR</td>
</tr>
<tr>
<td>10010</td>
<td>GBR</td>
</tr>
<tr>
<td>10011</td>
<td>MACL</td>
</tr>
</tbody>
</table>

---

**Diagram:**

- (Rn) 0 ...................... 0
- BN: Bank number field
- EN: Entry number field
- R0
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12
- R13
- R14
- MACH
- Interrupt vector offset
- PR
- GBR
- MACL
Note
The architecture supports a maximum of 512 banks. However, the number of banks differs depending on the product.

Operation

```
STBANK (long n) /*STBANK R0, @Rn */
{
    Write_Bank_Long (R[n], R[0])
    PC+=2;
}
```

Examples:

```
STBANK R0, @R1 ; Before execution: R1 = H'00000108, R0 = H'FFFFFFFF
 ; After execution: Contents of R2 stored   R2 = H'FFFFFFFF
```
6.3.36  STC STORE Control register  System Control Instruction

Store from Control Register  SH-2A/SH2A-FPU (New)

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>STC TBR, Rn</td>
<td>TBR → Rn</td>
<td>0000nnnn01001010</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Stores data in control register TBR in a destination.

**Operation**

```c
STCTBR(long n) /* STC TBR, Rn*/
{
    R[n]=TBR;
    PC+=2;
}
```

**Examples:**

- Before execution: R0 = H'12345678, TBR = H'00000000
- After execution: R0 = H'00000000
### 6.4 SH-2E CPU Instructions

#### 6.4.1 ADD

**ADD Binary Addition**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rm,Rn</td>
<td>Rm + Rn → Rn</td>
<td>0011nnnnnnnnnn1100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>ADD #imm,Rn</td>
<td>Rn + imm → Rn</td>
<td>0111nnnnnnnnnnnn</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

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

**Operation**

```
ADD(long m,long n) /* ADD Rm,Rn */
{
    R[n]+=R[m];
    PC+=2;
}
ADDI(long i,long n) /* ADD #imm,Rn */
{
    if ((i&0x80)==0) R[n]+=(0x000000FF & (long)i);
    else R[n]+=(0xFFFFFF00 | (long)i);
    PC+=2;
}
```

**Examples:**

- `ADD R0,R1` ; Before execution: R0 = H'7FFFFFFF, R1 = H'00000001  
  ; After execution: R1 = H'80000000
- `ADD #H'01,R2` ; Before execution: R2 = H'00000000  
  ; After execution: R2 = H'00000001
- `ADD #H'FE,R3` ; Before execution: R3 = H'00000001  
  ; After execution: R3 = H'FFFFFFFE`
### 6.4.2 ADDC ADD with Carry

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDC</td>
<td>Rm,Rn</td>
<td>Rn + Rm + T → Rn, carry → T</td>
<td>0011nnnnnnmmm1110</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Description

Adds Rm data and the T bit to general register Rn data, and stores the result in Rn. The T bit changes according to the result. This instruction can add data that has more than 32 bits.

#### Operation

```c
ADDC (long m,long n) /* ADDC Rm,Rn */
{
    unsigned long tmp0,tmp1;

    tmp1=R[n]+R[m];
    tmp0=R[n];
    R[n]=tmp1+T;
    if (tmp0>=tmp1) T=1;
    else T=0;
    if (tmp1>R[n]) T=1;
    PC+=2;
}
```

#### Examples:

- **CLRT**; R0:R1 (64 bits) + R2:R3 (64 bits) = R0:R1 (64 bits)
- **ADDC R3,R1**; Before execution: T = 0, R1 = H'00000001, R3 = H'FFFFFFFF; After execution: T = 1, R1 = H'00000000
- **ADDC R2,R0**; Before execution: T = 1, R0 = H'00000000, R2 = H'00000000; After execution: T = 0, R0 = H'00000001
6.4.3 ADDV ADD with (V flag) overflow check Arithmetic Instruction

Binary Addition with Overflow Check

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV</td>
<td>Rm,Rn</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rn + Rm → Rn, overflow → T</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011nnnnnnnnnnnnnnnn111111100</td>
<td>1</td>
<td>Overflow</td>
</tr>
</tbody>
</table>

**Description**

Adds general register Rn data to Rm data, and stores the result in Rn. If an overflow occurs, the T bit is set to 1.

**Operation**

ADDV(long m,long n) /*ADDV Rm,Rn */
{
    long dest,src,ans;

    if ((long)R[n]>=0) dest=0;
    else dest=1;
    if ((long)R[m]>=0) src=0;
    else src=1;
    src+=dest;
    R[n]+=R[m];
    if ((long)R[n]>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (src==0 || src==2) {
        if (ans==1) T=1;
        else T=0;
    }
    else T=0;
    PC+=2;
}
Examples:

ADDV R0, R1
; Before execution:  R0 = H'00000001, R1 = H'7FFFFFFE, T = 0
; After execution:  R1 = H'7FFFFFFF, T = 0

ADDV R0, R1
; Before execution:  R0 = H'00000002, R1 = H'7FFFFFFE, T = 0
; After execution:  R1 = H'80000000, T = 1
### Logical AND

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND Rm,Rn</td>
<td>Rn &amp; Rm → Rn</td>
<td>0010nnnnnnn1001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>AND #imm,R0</td>
<td>R0 &amp; imm → R0</td>
<td>1100100111111111</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>AND.B #imm, @(R0,GBR)</td>
<td>(R0 + GBR) &amp; imm → (R0 + GBR)</td>
<td>1100111011111111</td>
<td>3</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Logically ANDs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can be ANDed with zero-extended 8-bit immediate data. 8-bit memory data pointed to by GBR relative addressing can be ANDed with 8-bit immediate data.

**Note**

After AND #imm, R0 is executed and the upper 24 bits of R0 are always cleared to 0.
Operation

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

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

ANDM(long i) /* AND.B #imm,@(R0,GBR) */
{
    long temp;
    temp = (long)Read_Byte(GBR + R[0]);
    temp &= (0x000000FF & (long)i);
    Write_Byte(GBR + R[0], temp);
    PC += 2;
}

Examples:

AND R0, R1 ; Before execution: R0 = H'AAAAAAAA, R1 = H'55555555
    ; After execution: R1 = H'00000000

AND #H'0F, R0 ; Before execution: R0 = H'FFFFFFFF
    ; After execution: R0 = H'0000000F

AND.B #H'80, @(R0, GBR) ; Before execution: @(R0, GBR) = H'A5
    ; After execution: @(R0, GBR) = H'80
6.4.5 BF Branch if False Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF label</td>
<td>When T = 0, disp × 2 + PC → PC; When T = 1, nop</td>
<td>10001011ddddddd</td>
<td>3/1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Reads the T bit, and conditionally branches. If T = 0, it branches to the branch destination address. If T = 1, BF executes the next instruction. The branch destination is an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is −256 to +254 bytes. If the displacement is too short to reach the branch destination, use BF with the BRA instruction or the like.

**Note**

When branching, three cycles; when not branching, one cycle.

**Operation**

```c
BF(long d) /* BF disp */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==0) PC=PC+(disp<<1);
    else PC+=2;
}
```
Example:

```
CLRT ; T is always cleared to 0
BT TRGET_T ; Does not branch, because T = 0
BF TRGET_F ; Branches to TRGET_F, because T = 0
NOP ;
NOP ; ← The PC location is used to calculate the branch destination
............ address of the BF instruction
TRGET_F: ; ← Branch destination of the BF instruction
```
6.4.6 BF/S Branch if False with delay Slot Branch Instruction

**Conditional Branch with Delay**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF/S label</td>
<td>When T = 0, disp × 2+ PC → PC; When T = 1, nop</td>
<td>10001111dddddddd</td>
<td>2/1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Reads the T bit and conditionally branches. If T = 0, it branches after executing the next instruction. If T = 1, BF/S executes the next instruction. The branch destination is an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –256 to +254 bytes. If the displacement is too short to reach the branch destination, use BF with the BRA instruction or the like.

**Note**

Since this is a delay branch instruction, the instruction immediately following is executed before the branch. No interrupts and address errors are accepted between this instruction and the next instruction. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction. When branching, this is a two-cycle instruction; when not branching, one cycle.
Section 6  Instruction Descriptions

Operation

BFS(long d) /* BFS disp */
{
    long disp;
    unsigned long temp;

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==0) {
        PC=PC+(disp<<1);
        Delay_Slot(temp+2);
    }
    else PC+=2;
}

Example:

CLRT ; T is always 0
BT/S TRGET_T ; Does not branch, because T = 0
NOP ;
BF/S TRGET_F ; Branches to TRGET_F, because T = 0
ADD R0,R1 ; Executed before branch.
NOP ; ← The PC location is used to calculate the branch destination
............ address of the BF/S instruction
TRGET_F: ; ← Branch destination of the BF/S instruction

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
### 6.4.7 BRA BRAnch Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA</td>
<td>label disp × 2 + PC → PC</td>
<td>1010ddddddddddd</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

#### Description

Branches unconditionally after executing the instruction following this BRA instruction. The branch destination is an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. The 12-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –4096 to +4094 bytes. If the displacement is too short to reach the branch destination, this instruction must be changed to the JMP instruction. Here, a MOV instruction must be used to transfer the destination address to a register.

#### Note

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

#### Operation

```c
BRA(long d) /* BRA disp */
{
    unsigned long temp;
    long disp;

    if ((d&0x800)==0) disp=(0x00000FFF & (long) d);
    else disp=(0xFFFFF000 | (long) d);
    temp=PC;
    PC=PC+(disp<<1);
    Delay_Slot(temp+2);
}
```
Example:

```
BRA TRGET ; Branches to TRGET
ADD R0, R1 ; Executes ADD before branching
NOP ← The PC location is used to calculate the branch destination
........... address of the BRA instruction
TRGET: ← Branch destination of the BRA instruction
```

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
### 6.4.8 BRAF BRAnch Far Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAF</td>
<td>Rm + PC → PC</td>
<td>0000nnnn00100011</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Branches unconditionally. The branch destination is PC + the 32-bit contents of the general register Rm. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction.

**Note**

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

**Operation**

```c
BRAF(long m) /* BRAF Rm */
{
    unsigned long temp;

    temp=PC;
    PC=PC+R[m];
    Delay_Slot(temp+2);
}
```
Example:

```
MOV.L #(TARGET-BSRF_PC), R0       ; Sets displacement.
BRA  TRGET                       ; Branches to TARGET
ADD  R0, R1                      ; Executes ADD before branching
BRAF_PC:                         ; ← The PC location is used to calculate the branch
destination address of the BRAF instruction
NOP
............................
TARGET:                          ; ← Branch destination of the BRAF instruction
```

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
### 6.4.9 BSR Branch to SubRoutine Procedure

Branch to SubRoutine Delayed Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR</td>
<td>label</td>
<td>PC → PR, disp × 2+ PC → PC</td>
<td>1011dddddddddddd</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Description

Branches to the subroutine procedure at a specified address. The PC value is stored in the PR, and the program branches to an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. The 12-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is –4096 to +4094 bytes. If the displacement is too short to reach the branch destination, the JSR instruction must be used instead. With JSR, the destination address must be transferred to a register by using the MOV instruction. This BSR instruction and the RTS instruction are used together for a subroutine procedure call.

#### Note

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

#### Operation

```c
BSR(long d) /* BSR disp */
{
    long disp;

    if (((d&0x800)==0) disp=(0x00000FFF & (long) d);
    else disp=(0xFFFFF000 | (long) d);
    PR=PC+Is_32bit_Inst(PR+2);
    PC=PC+(disp<<1);
    Delay_Slot(PR+2);
}
```
Example:

```
BSR TRGET ; Branches to TRGET
MOV R3,R4 ; Executes the MOV instruction before branching
ADD R0,R1 ; ← The PC location is used to calculate the branch destination address of the BSR instruction (return address for when the subroutine procedure is completed (PR data))

......
......

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

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
6.4.10 BSRF  
Branch to SubRoutine Far  
Branch to Subroutine Procedure  
Branch Instruction  
Delayed Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSRF Rm</td>
<td>PC → PR, Rm + PC → PC</td>
<td>00000000000011</td>
<td>2</td>
</tr>
</tbody>
</table>

**Description**

Branches to the subroutine procedure at a specified address after executing the instruction following this BSRF instruction. The PC value is stored in the PR. The branch destination is PC + the 32-bit contents of the general register Rm. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. Used as a subroutine procedure call in combination with RTS.

**Note**

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

**Operation**

```
BSRF(long m) /* BSRF Rm */
{
    PR=PC
    PC=PC+R[m];
    Delay_Slot(PR+2);
}
```
Example:

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

BSRF_PC:
ADD R0,R1 ; ← The PC location is used to calculate the branch
.... ; destination with BSRF.
.... ;

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

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delay slot instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
### 6.4.11 BT Branch if True

**Conditional Branch**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT label</td>
<td>When $T = 1$, disp $\times 2 + PC \rightarrow PC$; When $T = 0$, nop</td>
<td>10001001ddddddddd</td>
<td>3/1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Reads the T bit, and conditionally branches. If $T = 1$, BT branches. If $T = 0$, BT executes the next instruction. The branch destination is an address specified by $PC +$ displacement. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is $-256$ to $+254$ bytes. If the displacement is too short to reach the branch destination, use BT with the BRA instruction or the like.

**Note**

When branching, requires three cycles; when not branching, one cycle.

**Operation**

```c
BT(long d) /* BT disp */
{
    long disp;

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);

    if (T==1) PC=PC+(disp<<1);
    else PC+=2;
}
```
Example:

```assembly
SETT ; T is always 1
BF TRGET_F ; Does not branch, because T = 1
BT TRGET_T ; Branches to TRGET_T, because T = 1
NOP
NOP ; ← The PC location is used to calculate the branch destination
............
TRGET_T: ; ← Branch destination of the BT instruction
```
### 6.4.12 BT/S Branch if True with delay Slot

Conditional Branch with Delay

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT/S</td>
<td>label When T = 1, disp \times 2 + PC → PC;</td>
<td>10001101ddddd</td>
<td>2/1</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>When T = 0, nop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Description

Reads the T bit and conditionally branches. If T = 1, BT/S branches after the following instruction executes. If T = 0, BT/S executes the next instruction. The branch destination is an address specified by PC + displacement. However, in this case it is used for address calculation. The PC is the address 4 bytes after this instruction. The 8-bit displacement is sign-extended and doubled. Consequently, the relative interval from the branch destination is −256 to +254 bytes. If the displacement is too short to reach the branch destination, use BT/S with the BRA instruction or the like.

### Note

Since this is a delay branch instruction, the instruction immediately following is executed before the branch. No interrupts and address errors are accepted between this instruction and the next instruction. When the immediately following instruction is a branch instruction, it is recognized as an illegal slot instruction. When branching, requires two cycles; when not branching, one cycle.
Operation

BTS(long d) /* BTS disp */
{
    long disp;
    unsigned long temp;

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==1) {
        PC=PC+(disp<<1);
        Delay_Slot(temp+2);
    }
    else PC+=2;
}

Example:

SETT ; T is always 1
BF/S TARGET_F ; Does not branch, because T = 1
NOP ;
BT/S TARGET_T ; Branches to TARGET, because T = 1
ADD R0,R1 ; Executes before branching.
NOP ; ← The PC location is used to calculate the branch destination
.......... ; ← The location is used to calculate the branch destination
TARGET_T: ; ← Branch destination of the BT/S instruction

Note: When a delayed branch instruction is used, the branching operation takes place after the
slot instruction is executed, but the execution of instructions (register update, etc.) takes
place in the sequence delayed branch instruction → delayed slot instruction. For example,
even if a delayed slot instruction is used to change the register where the branch
destination address is stored, the register content previous to the change will be used as the
branch destination address.
6.4.13  CLRMAC  CleaR MAC register  System Control Instruction

MAC Register Clear

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRMAC</td>
<td>0 → MACH, MACL</td>
<td>0000000000101000</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Clear the MACH and MACL Register.

Operation

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

Example:

```assembly
CLRMAC ; Clears and initializes the MAC register
MAC.W @R0+,@R1+ ; Multiply and accumulate operation
MAC.W @R0+,@R1+ ;
```
6.4.14 CLRT CleaR T bit System Control Instruction

**T Bit Clear**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRT</td>
<td>0 → T</td>
<td>0000000000001000</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Description**

Clears the T bit.

**Operation**

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

**Example:**

```plaintext
CLRT ; Before execution: T = 1
       ; After execution: T = 0
```
### 6.4.15 CMP/cond

**Compare conditionally Arithmetic Instruction**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP/EQ</td>
<td>$Rm, Rn$ When $Rn = Rm$, $1 \rightarrow T$</td>
<td>0011nnnnnnnnnnnn0000</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/GE</td>
<td>$Rm, Rn$ When signed and $Rn \geq Rm$, $1 \rightarrow T$</td>
<td>0011nnnnnnnnnnnn0011</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/GT</td>
<td>$Rm, Rn$ When signed and $Rn &gt; Rm$, $1 \rightarrow T$</td>
<td>0011nnnnnnnnnnnn0111</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/HI</td>
<td>$Rm, Rn$ When unsigned and $Rn &gt; Rm$, $1 \rightarrow T$</td>
<td>0011nnnnnnnnnnnn0110</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/HS</td>
<td>$Rm, Rn$ When unsigned and $Rn \geq Rm$, $1 \rightarrow T$</td>
<td>0011nnnnnnnnnnnn0010</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/PL</td>
<td>$Rn$ When $Rn &gt; 0$, $1 \rightarrow T$</td>
<td>0100nnnn00010101</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/PZ</td>
<td>$Rn$ When $Rn \geq 0$, $1 \rightarrow T$</td>
<td>0100nnnn00010001</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/STR</td>
<td>$Rm, Rn$ When a byte in $Rn$ equals a byte in $Rm$, $1 \rightarrow T$</td>
<td>0010nnnnnnnnnnn1100</td>
<td>1</td>
<td>Comparison result</td>
</tr>
<tr>
<td>CMP/EQ</td>
<td>#imm, R0 When $R0 = imm$, $1 \rightarrow T$</td>
<td>1000100011111111</td>
<td>1</td>
<td>Comparison result</td>
</tr>
</tbody>
</table>

**Description**

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

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP/EQ</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>CMP/GE</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>CMP/GT</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>CMP/HI</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>CMP/HS</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>CMP/PL</td>
<td>Rn</td>
</tr>
<tr>
<td>CMP/PZ</td>
<td>Rn</td>
</tr>
<tr>
<td>CMP/STR</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>CMP/EQ</td>
<td>#imm,R0</td>
</tr>
</tbody>
</table>

### Operation

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

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

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

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

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

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

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

CMPSTR(long m,long n) /* CMP_STR Rm,Rn */
{
    unsigned long temp;
    long HH,HL,LH,LL;

    temp=R[n]^R[m];
    HH=(temp>>24)&0x000000FF;
    HL=(temp>>16)&0x000000FF;
    LH=(temp>>8)&0x000000FF;
    LL=temp&0x000000FF;
    HH=HH&HL&LH&LL;
    if (HH==0) T=1;
    else T=0;
    PC+=2;
CMPIM(long i) /* CMP_EQ #imm,R0 */
{
    long imm;
    if ((i&0x80)==0) imm=(0x000000FF & (long i));
    else imm=(0xFFFFFF00 | (long i));
    if (R[0]==imm) T=1;
    else T=0;
    PC+=2;
}

Example:

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

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

### Operation

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

**Example:** See DIV1.
6.4.17 DIV0U DIVide (step 0) as Unsigned Arithmetic Instruction
Initialization for Unsigned Division

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV0U</td>
<td>0 → M/Q/T</td>
<td>0000000000011001</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**Operation**

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

**Example:** See DIV1.
6.4.18 DIV1 DIVide 1 step Arithmetic Instruction

**Description**

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

In one-step division, the dividend is shifted one bit left, the divisor is subtracted and the quotient bit reflected in the Q bit according to the status (positive or negative). To find the remainder in a division, first find the quotient using a DIV1 instruction, then find the remainder as follows:

\[(\text{dividend}) - (\text{divisor}) \times (\text{quotient}) = (\text{remainder})\]

Zero division, overflow detection, and remainder operation are not supported. Check for zero division and overflow division before dividing.

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

    DIV1(long m, long n) /* DIV1 Rm, Rn */
    {
        unsigned long tmp0;
        unsigned char old_q, tmp1;

        old_q = Q;
        Q = (unsigned char)((0x80000000 & R[n]) != 0);
        R[n] <<= 1;
        R[n] |= (unsigned long) T;
        switch (old_q) {
            case 0:
                switch (M) {
                    case 0:
                        tmp0 = R[n];
                        R[n] -= R[m];
                        tmp1 = (R[n] > tmp0);
                        switch (Q) {
                            case 0:
                                Q = tmp1;
                                break;
                            case 1:
                                Q = (unsigned char)(tmp1 == 0);
                                break;
                        }
                        break;
                    case 1:
                        tmp0 = R[n];
                        R[n] += R[m];
                        tmp1 = (R[n] < tmp0);
                        switch (Q) {
                            case 0:
                                Q = (unsigned char)(tmp1 == 0);
                                break;
                            case 1:
                                Q = tmp1;
                                break;
                        }
                        break;
                }
            break;
        }
    }
case 1: switch(M) {
    case 0: tmp0 = R[n];
        R[n] += R[m];
        tmp1 = R[n] < tmp0;
        switch(Q) {
            case 0: Q = tmp1;
                break;
            case 1: Q = (unsigned char)(tmp1 == 0);
                break;
        }
        break;
    case 1: tmp0 = R[n];
        R[n] -= R[m];
        tmp1 = R[n] > tmp0;
        switch(Q) {
            case 0: Q = (unsigned char)(tmp1 == 0);
                break;
            case 1: Q = tmp1;
                break;
        }
        break;
    }
}

T = (Q == M);
Pc += 2;
Example 1:

; R1 (32 bits) / R0 (16 bits) = R1 (16 bits):Unsigned

SHLL16 R0 ; Upper 16 bits = divisor, lower 16 bits = 0
TST R0,R0 ; Zero division check
BT ZERO_DIV ;
CMP/HS R0,R1 ; Overflow check
BT OVER_DIV ;
DIV0U ; Flag initialization
.arepeat 16 ;
DIV1 R0,R1 ; Repeat 16 times
.aendr ;
ROTCL R1 ;
EXTU.W R1,R1 ; R1 = Quotient

Example 2:

; R1:R2 (64 bits)/R0 (32 bits) = R2 (32 bits):Unsigned

TST R0,R0 ; Zero division check
BT ZERO_DIV ;
CMP/HS ;R0,R1 ; Overflow check
BT OVER_DIV ;
DIV0U ; Flag initialization
.arepeat 32 ;
ROTCL R2 ; Repeat 32 times
DIV1 R0,R1 ;
.aendr ;
ROTCL R2 ; R2 = Quotient
Example 3:

; R1 (16 bits)/R0 (16 bits) = R1 (16 bits):Signed
SHLL16 R0 ; Upper 16 bits = divisor, lower 16 bits = 0
EXTS.W R1,R1 ; Sign-extends the dividend to 32 bits
XOR R2,R2 ; R2 = 0
MOV R1,R3 ;
ROTC R3 ;
SUBC R2,R1 ; Decrement if the dividend is negative
DIV0S R0,R1 ; Flag initialization
. arepeat 16 ;
DIV1 R0,R1 ; Repeat 16 times
.aendr
EXTS.W R1,R1 ; R1 = quotient (one’s complement)
ROTC R1 ;
ADDC R2,R1 ; Increments and takes the two’s complement if the MSB of the quotient is 1
EXTS.W R1,R1 ; R1 = quotient (two’s complement)

Example 4:

; R2 (32 bits) / R0 (32 bits) = R2 (32 bits):Signed
MOV R2,R3 ;
ROTC R3 ;
SUBC R1,R1 ; Sign-extends the dividend to 64 bits (R1:R2)
XOR R3,R3 ; R3 = 0
SUBC R3,R2 ; Decrement and takes the one’s complement if the dividend is negative
DIV0S R0,R1 ; Flag initialization
. arepeat 32 ;
ROTC R2 ; Repeat 32 times
DIV1 R0,R1 ;
.aendr ;
ROTC R2 ; R2 = Quotient (one’s complement)
ADDC R3,R2 ; Increments and takes the two’s complement if the MSB of the quotient is 1.
R2 = Quotient (two’s complement)
6.4.19 **DMULS.L**

Signed Double-Length Multiply as Signed Arithmetic Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMULS.L</td>
<td>Rm, Rn</td>
<td>With sign, Rn × Rm → MACH, MACL</td>
<td>0011nnnnnnnnnnnn1101</td>
<td>4</td>
</tr>
</tbody>
</table>

**Description**

Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the 64-bit results in the MACL and MACH register. The operation is a signed arithmetic operation.

**Operation**

```c
DMULS(long m,long n) /* DMULS.L Rm,Rn */
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,temp1,temp2,temp3;
    long tempm,tempn,fnLmL;

    tempn=(long)R[n];
    tempm=(long)R[m];
    if (tempn<0) tempn=0-tempn;
    if (tempm<0) tempm=0-tempm;
    if ((long)(R[n]^R[m])<0) fnLmL=-1;
    else fnLmL=0;

    temp1=(unsigned long)tempn;
    temp2=(unsigned long)tempm;

    RnL=temp1&0x0000FFFF;
    RnH=(temp1>>16)&0x0000FFFF;
    RmL=temp2&0x0000FFFF;
    RmH=(temp2>>16)&0x0000FFFF;
```

Rev. 3.00  Jul 08, 2005  page 192 of 484
REJ09B0051-0300

Renesas
temp0=RmL*RnL;
temp1=RmH*RnL;
temp2=RmL*RnH;
temp3=RmH*RnH;

Res2=0
Res1=temp1+temp2;
if (Res1<temp1) Res2+=0x00010000;

temp1=(Res1<<16)&0xFFFF0000;
Res0=temp0+temp1;
if (Res0<temp0) Res2++;

Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

if (fnLmL<0) {
    Res2=~Res2;
    if (Res0==0)
        Res2++;           
    else
        Res0=(-Res0)+1;
}
MACH=Res2;
MACL=Res0;
PC+=2;
}

Example:
DMULS.L R0,R1 ; Before execution: R0 = H'FFFFFFFE, R1 = H'00005555
               ; After execution: MACH = H'FFFFFFFF, MACL = H'FFFF5556
STS MACH,R0 ; Operation result (top)
STS MACL,R0 ; Operation result (bottom)
6.4.20 DMULU.L Double-length MULtiply as Unsigned Arithmetic Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMULU.L</td>
<td>Rm, Rn</td>
<td>0011nnnnnn0101</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the 64-bit results in the MACL and MACH register. The operation is an unsigned arithmetic operation.

**Operation**

```c
DMULU(long m, long n) /* DMULU.L Rm,Rn */
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,temp1,temp2,temp3;

    RnL=R[n]&0x0000FFFF;
    RnH=(R[n]>>16)&0x0000FFFF;

    RmL=R[m]&0x0000FFFF;
    RmH=(R[m]>>16)&0x0000FFFF;

    temp0=RmL*RnL;
    temp1=RmH*RnL;
    temp2=RmL*RnH;
    temp3=RmH*RnH;

    Res2=0
    Res1=temp1+temp2;
    if (Res1<temp1) Res2+=0x00010000;
    temp1=(Res1<<16)&0xFFFF0000;
```
Res0=temp0+temp1;
if (Res0<temp0) Res2++;

Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

MACH=Res2;
MACL=Res0;
PC+=2;
}

Example:
DMULU.L R0,R1 ; Before execution: R0 = H'FFFFFFFE, R1 = H'00005555
               ; After execution: MACH = H'FFFFFFFF, MACL = H'FFFF5556
STS     MACH,R0 ; Operation result (top)
STS     MACL,R0 ; Operation result (bottom)
6.4.21 DT Decrement and Test

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT</td>
<td>Rn</td>
<td>0100nnnn00010000</td>
<td>1</td>
<td></td>
<td>Comparison result</td>
</tr>
</tbody>
</table>

**Description**

The contents of general register Rn are decremented by 1 and the result compared to 0 (zero). When the result is 0, the T bit is set to 1. When the result is not zero, the T bit is set to 0.

**Operation**

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

**Example:**

```assembly
MOV #4,R5 ; Sets the number of loops.
LOOP:
    ADD R0,R1 ;
    DT R5 ; Decrement the R5 value and checks whether it has become 0.
    BF LOOP ; Branches to LOOP is T=0. (In this example, loops 4 times.)
```
6.4.22 EXTS EXTend as Signed Arithmetic Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTS.B</td>
<td>Rm, Rn Sign-extend Rm from byte → Rn</td>
<td>0110nnnnnnnnnnnnnn1110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>EXTS.W</td>
<td>Rm, Rn Sign-extend Rm from word → Rn</td>
<td>0110nnnnnnnnnnnnnn1111</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Sign-extends general register Rm data, and stores the result in Rn. If byte length is specified, the bit 7 value of Rm is copied into bits 8 to 31 of Rn. If word length is specified, the bit 15 value of Rm is copied into bits 16 to 31 of Rn.

Operation

```c
EXTSB(long m, long n) /* EXTS.B Rm,Rn */
{
    R[n] = R[m];
    if ((R[m] & 0x00000080) == 0) R[n] &= 0x000000FF;
    else R[n] |= 0xFFFFFF00;
    PC += 2;
}
EXTSW(long m, long n) /* EXTS.W Rm,Rn */
{
    R[n] = R[m];
    if ((R[m] & 0x00008000) == 0) R[n] &= 0x0000FFFF;
    else R[n] |= 0xFFFF0000;
    PC += 2;
}
```

Examples:

- EXTS.B R0,R1 ; Before execution: R0 = H'000000080
  ; After execution: R1 = H'FFFFFF80
- EXTS.W R0,R1 ; Before execution: R0 = H'000008000
  ; After execution: R1 = H'FFFFFF8000
### 6.4.23 EXTU
Zero Extension
EXTend as Unsigned Arithmetic Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTU.B</td>
<td>Rm, Rn</td>
<td>Zero-extend Rm from byte → Rn</td>
<td>0110nnnnnnmm1100</td>
<td>1</td>
</tr>
<tr>
<td>EXTU.W</td>
<td>Rm, Rn</td>
<td>Zero-extend Rm from word → Rn</td>
<td>0110nnnnnnmm1101</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**
Zero-extends general register Rm data, and stores the result in Rn. If byte length is specified, 0s are written in bits 8 to 31 of Rn. If word length is specified, 0s are written in bits 16 to 31 of Rn.

**Operation**

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

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

**Examples:**

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

- **EXTU.W R0,R1**
  - Before execution: R0 = H'FFFF8000
  - After execution: R1 = H'00008000
6.4.24 JMP JuMP Branch Instruction

### Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP @Rm</td>
<td>Rm → PC</td>
<td>0100m00001011011</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

### Description

Branches unconditionally to the address specified by register indirect addressing. The branch destination is an address specified by the 32-bit data in general register Rm.

### Note

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

### Operation

```c
JMP(long m) /* JMP @Rm */
{
    unsigned long temp;

    temp=PC;
    PC=R[m]+4;
    Delay_Slot(temp+2);
}
```
Example:

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

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
6.4.25 JSR Jump to SubRoutine Branch Instruction

Branch to Subroutine Procedure

Delayed Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR @Rm</td>
<td>PC → PR, Rm → PC</td>
<td>0100mmmm00001011</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Branches to the subroutine procedure at the address specified by register indirect addressing. The PC value is stored in the PR. The jump destination is an address specified by the 32-bit data in general register Rm. The stored/saved PC is the address four bytes after this instruction. The JSR instruction and RTS instruction are used together for subroutine procedure calls.

**Note**

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

**Operation**

```c
JSR(long m) /* JSR @Rm */
{
    PR=PC;
    PC=R[m]+4;
    Delay_Slot(PR+2);
}
```
Example:

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

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
6.4.26  LDC
Load to Control Register
System Control Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC</td>
<td>Rm,SR</td>
<td>Rm → SR</td>
<td>0100mmmm00001110</td>
<td>3</td>
</tr>
<tr>
<td>LDC</td>
<td>Rm,GBR</td>
<td>Rm → GBR</td>
<td>0100mmmm00011110</td>
<td>1</td>
</tr>
<tr>
<td>LDC</td>
<td>Rm,VBR</td>
<td>Rm → VBR</td>
<td>0100mmmm00101110</td>
<td>1</td>
</tr>
<tr>
<td>LDC.L</td>
<td>@Rm+,SR</td>
<td>(Rm) → SR, Rm + 4 → Rm</td>
<td>0100mmmm00000111</td>
<td>5</td>
</tr>
<tr>
<td>LDC.L</td>
<td>@Rm+,GBR</td>
<td>(Rm) → GBR, Rm + 4 → Rm</td>
<td>0100mmmm00010111</td>
<td>1</td>
</tr>
<tr>
<td>LDC.L</td>
<td>@Rm+,VBR</td>
<td>(Rm) → VBR, Rm + 4 → Rm</td>
<td>0100mmmm00100111</td>
<td>1</td>
</tr>
</tbody>
</table>

Description
Store the source operand into control register SR, GBR, or VBR.

Operation
LDCSR(long m) /* LDC Rm,SR */
{
    SR=R[m] & 0x000063F3;
    PC+=2;
}
LDCGBR(long m) /* LDC Rm,GBR */
{
    GBR=R[m];
    PC+=2;
}
LDCVBR(long m) /* LDC Rm,VBR */
{
    VBR=R[m];
    PC+=2;
}
LDCMSR(long m) /* LDC.L @Rm+,SR */
{
    SR=Read_Long(R[m]) & 0x000063F3;
Section 6 Instruction Descriptions

LDCMGBR(long m) /* LDC.L @Rm+,GBR */
{
    GBR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

LDCMVBR(long m) /* LDC.L @Rm+,VBR */
{
    VBR=Read_Long(R[m]);
    R[m]+=4;
    PC+=2;
}

Examples:

LDC R0,SR ; Before execution: R0 = H'FFFFFFFF, SR = H'00000000
; After execution: SR = H'0000063F3

LDC.L @R15+,GBR ; Before execution: R15 = H'10000000
; After execution: R15 = H'10000004, GBR = @H'10000000
6.4.27 LDS Load to System Round

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS</td>
<td>Rm,MACH</td>
<td>0100mmmm000001010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS</td>
<td>Rm,MACL</td>
<td>0100mmmm00011010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS</td>
<td>Rm,PR</td>
<td>0100mmmm00101010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS.L</td>
<td>@Rm+, MACH (Rm) → MACH, Rm + 4 → Rm</td>
<td>0100mmmm00000110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS.L</td>
<td>@Rm+, MACL (Rm) → MACL, Rm + 4 → Rm</td>
<td>0100mmmm00010110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS.L</td>
<td>@Rm+, PR (Rm) → PR, Rm + 4 → Rm</td>
<td>0100mmmm00100110</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Store the source operand into the system register MACH, MACL, or PR.

Operation

```c
LDSMACH(long m) /* LDS Rm,MACH */
{
    MACH=R[m];
    PC+=2;
}
LDSMACL(long m) /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}
LDSPR(long m) /* LDS Rm,PR */
{
    PR=R[m];
    PC+=2;
}
LDSMMACH(long m) /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
```
LDSMMACL(long m) /* LDS.L @Rm+,MACL */
{
    MACL = Read_Long(R[m]);
    R[m] += 4;
    PC += 2;
}

LDSMPR(long m) /* LDS.L @Rm+,PR */
{
    PR = Read_Long(R[m]);
    R[m] += 4;
    PC += 2;
}

Examples:

LDS R0,PR ; Before execution: R0 = H'12345678, PR = H'00000000
; After execution: PR = H'12345678

LDS.L @R15+,MACL ; Before execution: R15 = H'10000000
; After execution: R15 = H'10000004, MACL = @H'10000000
6.4.28 MAC.L Multiply and ACcumulate

Long Arithmetic Instruction

Double-Precision Multiply-and-Accumulate Operation

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC.L</td>
<td>@Rm+, @Rn+ signed operation,</td>
<td>0000nnnnmmmm1111</td>
<td>4</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>(Rn) × (Rm) + MAC → MAC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

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

When the S bit is cleared to 0, the 64-bit result is stored in the coupled MACH and MACL registers. When bit S is set to 1, addition to the MAC register is a saturation operation of 48 bits starting from the LSB. For the saturation operation, only the lower 48 bits of the MACL register are enabled and the result is limited to a range of H'FFFF800000000000 (minimum) and H'00007FFFFFFF (maximum).

**Operation**

```
MACL(long m, long n) /* MAC.L @Rm+,@Rn+*/
{
    unsigned long RnL, RnH, RmL, RmH, Res0, Res1, Res2;
    unsigned long temp0, temp1, temp2, temp3;
    long tempm, tempn, fnLmL;

    tempn = (long)Read_Long(R[n]);
    R[n] += 4;
    tempm = (long)Read_Long(R[m]);
    R[m] += 4;

    if ((long)(tempn ^ tempm) < 0) fnLmL = -1;
    else fnLmL = 0;
```
if (tempn<0) tempn=0-tempn;
if (tempm<0) tempm=0-tempm;

temp1=(unsigned long)tempn;
temp2=(unsigned long)tempm;

RnL=temp1&0x0000FFFF;
RnH=(temp1>>16)&0x0000FFFF;
RmL=temp2&0x0000FFFF;
RmH=(temp2>>16)&0x0000FFFF;

temp0=RmL*RnL;
temp1=RmH*RnL;
temp2=RmL*RnH;
temp3=RmH*RnH;

Res2=0;

Res1=temp1+temp2;
if (Res1<temp1) Res2+=0x00010000;

temp1=(Res1<<16)&0xFFFF0000;
Res0=temp0+temp1;
if (Res0<temp0) Res2++;

Res2=Res2+((Res1>>16)&0x0000FFFF)+temp3;

if(fnLmL<0){
  Res2=~Res2;
  if (Res0==0) Res2++;
  else Res0=(~Res0)+1;
}
if(S==1){
  Res0=MACL+Res0;
  if (MACL>Res0) Res2++;
if (MACH&0x00008000);
else Res2+=MACH|0xFFFF0000;
    Res2+=MACH&0x00007FFF;

if(((long)Res2<0)&&(Res2<0xFFFF8000)){
    Res2=0xFFFF8000;
    Res0=0x00000000;
}
if(((long)Res2>0)&&(Res2>0x00007FFF)){
    Res2=0x00007FFF;
    Res0=0xFFFFFFFF;
};

MACH=(Res2&0x0000FFFF)|(MACH&0xFFFF0000)
MACL=Res0;
}
else {
    Res0=MACL+Res0;
    if (MACL>Res0) Res2++;
    Res2+=MACH

    MACH=Res2;
    MACL=Res0;
}
PC+=2;
Example:

```
MOVA   TBLM, R0   ; Table address
MOV    R0, R1    ;
MOVA   TBLN, R0   ; Table address
CLRMAC ; MAC register initialization
MAC.L  @R0+, @R1+ ;
MAC.L  @R0+, @R1+ ;
STS    MACL, R0   ; Store result into R0

..................
.align    2    ;
TBLM    .data.l H'1234ABCD    ;
          .data.l H'5678EF01    ;
TBLN    .data.l H'0123ABCD    ;
          .data.l H'4567DEF0    ;
```
6.4.29 MAC.W Multiply and ACcumulate Word Arithmetic Instruction

Single-Precision Multiply-and-Accumulate Operation

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC.W @Rm+, @Rn+</td>
<td>With sign, (Rn) × (Rm) + MAC → MAC</td>
<td>0100nnnnnnnn1111</td>
<td>3</td>
<td>—</td>
</tr>
<tr>
<td>MAC</td>
<td>@Rm+, @Rn+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Does signed multiplication of 16-bit operands obtained using the contents of general registers Rm and Rn as addresses. The 32-bit result is added to contents of the MAC register, and the final result is stored in the MAC register. Rm and Rn data are incremented by 2 after the operation.

When the S bit is cleared to 0, the operation is $16 \times 16 + 64 \rightarrow 64$-bit multiply and accumulate and the 64-bit result is stored in the coupled MACH and MACL registers.

When the S bit is set to 1, the operation is $16 \times 16 + 32 \rightarrow 32$-bit multiply and accumulate and addition to the MAC register is a saturation operation. For the saturation operation, only the MACL register is enabled and the result is limited to a range of $H'80000000$ (minimum) and $H'7FFFFFFF$ (maximum).

If an overflow occurs, the MACH register is set to $H'00000001$. The result is stored in the MACL register. The result is limited to a value between $H'80000000$ (minimum) for overflows in the negative direction and $H'7FFFFFFF$ (maximum) for overflows in the positive direction.

**Operation**

```c
MACW(long m,long n) /* MAC.W @Rm+,@Rn+*/
{
    long tempm,tempn,dest,src,ans;
    unsigned long templ;
    tempn=(long)Read_Word(R[n]);
    R[n]+=2;
    tempm=(long)Read_Word(R[m]);
    R[m]+=2;
```
templ=MACL;
tempm=((long)(short)tempn*(long)(short)tempm);
if ((long)MACL>=0) dest=0;
else dest=1;
if ((long)tempm>=0 {
    src=0;
    tempn=0;
} else {
    src=1;
    tempn=0xFFFFFFFF;
}
src+=dest;
MACL+=tempm;
if ((long)MACL>=0) ans=0;
else ans=1;
ans+=dest;
if (S==1) {
    if (ans==1) {
        MACH=0x00000001;
        if (src==0) MACL=0x7FFFFFFF;
        if (src==2) MACL=0x80000000;
    }
} else {
    MACH+=tempn;
    if (templ>MACL) MACH+=1;
}
PC+=2;
Example:

```
MOVA TBLM,R0 ; Table address
MOV R0,R1 ;
MOVA TBLN,R0 ; Table address
CLRMAC ; MAC register initialization
MAC.W @R0+,@R1+ ;
MAC.W @R0+,@R1+ ;
STS MACL,R0 ; Store result into R0
...............
.align 2 ;
TBLM .data.w H'1234 ;
       .data.w H'5678 ;
TBLN .data.w H'0123 ;
       .data.w H'4567 ;
```
### 6.4.30 MOV MOVe data Data Transfer Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Rm,Rn</td>
<td>Rm → Rn</td>
<td>0110nnnnmmmm0011</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B Rm,@Rn</td>
<td>Rm → (Rn)</td>
<td>0010nnnnmmmm0000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W Rm,@Rn</td>
<td>Rm → (Rn)</td>
<td>0010nnnnmmmm0001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L Rm,@Rn</td>
<td>Rm → (Rn)</td>
<td>0010nnnnmmmm0010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B @Rm,Rn</td>
<td>(Rm) → sign extension → Rn</td>
<td>0110nnnnmmmm0000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @Rm,Rn</td>
<td>(Rm) → sign extension → Rn</td>
<td>0110nnnnmmmm0001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @Rm,Rn</td>
<td>(Rm) → Rn</td>
<td>0110nnnnmmmm0010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B Rm,@–Rn</td>
<td>Rn – 1 → Rn, Rm → (Rn)</td>
<td>0010nnnnmmmm0100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W Rm,@–Rn</td>
<td>Rn – 2 → Rn, Rm → (Rn)</td>
<td>0010nnnnmmmm0101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L Rm,@–Rn</td>
<td>Rn – 4 → Rn, Rm → (Rn)</td>
<td>0010nnnnmmmm0110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B @Rm+,Rn</td>
<td>(Rm) → sign extension → Rn, Rm + 1 → Rm</td>
<td>0110nnnnmmmm0100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @Rm+,Rn</td>
<td>(Rm) → sign extension → Rn, Rm + 2 → Rm</td>
<td>0110nnnnmmmm0101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @Rm+,Rn</td>
<td>(Rm) → Rn, Rm + 4 → Rm</td>
<td>0110nnnnmmmm0110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B Rm,@(R0,Rn)</td>
<td>Rm → (R0 + Rn)</td>
<td>0000nnnnmmmm0100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W Rm,@(R0,Rn)</td>
<td>Rm → (R0 + Rn)</td>
<td>0000nnnnmmmm0101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L Rm,@(R0,Rn)</td>
<td>Rm → (R0 + Rn)</td>
<td>0000nnnnmmmm0110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B @(R0,Rm),Rn</td>
<td>(R0 + Rm) → sign extension → Rn</td>
<td>0000nnnnmmmm1100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @(R0,Rm),Rn</td>
<td>(R0 + Rm) → sign extension → Rn</td>
<td>0000nnnnmmmm1101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @(R0,Rm),Rn</td>
<td>(R0 + Rm) → Rn</td>
<td>0000nnnnmmmm1110</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

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

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

MOVBS(long m, long n) /* MOV.B Rm, @Rn */
{
    Write_Byte(R[n], R[m]);
    PC += 2;
}

MOVWS(long m, long n) /* MOV.W Rm, @Rn */
{
    Write_Word(R[n], R[m]);
    PC += 2;
}

MOVLS(long m, long n) /* MOV.L Rm, @Rn */
{
    Write_Long(R[n], R[m]);
    PC += 2;
}

MOVBL(long m, long n) /* MOV.B @Rm, Rn */
{
    R[n] = (long) Read_BYTE(R[m]);
    if (((R[n] & 0x80) == 0) R[n] &= 0x000000FF;
    else R[n] |= 0xFFFFFF00;
    PC += 2;
}

MOVWL(long m, long n) /* MOV.W @Rm, Rn */
{
    R[n] = (long) Read_Word(R[m]);
    if (((R[n] & 0x8000) == 0) R[n] &= 0x0000FFFF;
    else R[n] |= 0xFFFF0000;
    PC += 2;
}
MOVLL(long m, long n) /* MOV.L @Rm, Rn */
{
    R[n]=Read_Long(R[m]);
    PC+=2;
}

MOVBM(long m, long n) /* MOV.B Rm, @–Rn */
{
    Write_BYTE(R[n]-1, R[m]);
    R[n]=1;
    PC+=2;
}

MOVWM(long m, long n) /* MOV.W Rm, @–Rn */
{
    Write_Word(R[n]-2, R[m]);
    R[n]=2;
    PC+=2;
}

MOVLM(long m, long n) /* MOV.L Rm, @–Rn */
{
    Write_Long(R[n]-4, R[m]);
    R[n]=4;
    PC+=2;
}

MOVBP(long m, long n) /* MOV.B @Rm+, Rn */
{
    R[n]=(long)Read_BYTE(R[m]);
    if ((R[n]&0x80)==0) R[n]&0x000000FF;
    else R[n]|=0xFFFFFF00;
    if (n!=m) R[m]+=1;
    PC+=2;
}

MOVWP(long m, long n) /* MOV.W @Rm+, Rn */
{
    R[n]=(long)Read_Word(R[m]);
    if ((R[n]&0x8000)==0) R[n]&0x0000FFFF;

else R[n]|=0xFFFF0000;
if (n!=m) R[m]+=2;
PC+=2;
}

MOVLP(long m, long n) /* MOV.L @Rm+,Rn */
{
    R[n]=Read_Long(R[m]);
    if (n!=m) R[m]+=4;
    PC+=2;
}

MOVBS0(long m, long n) /* MOV.B Rm, @(R0,Rn) */
{
    Write_Byte(R[n]+R[0],R[m]);
    PC+=2;
}

MOVWS0(long m, long n) /* MOV.W Rm, @(R0,Rn) */
{
    Write_Word(R[n]+R[0],R[m]);
    PC+=2;
}

MOVLS0(long m, long n) /* MOV.L Rm, @(R0,Rn) */
{
    Write_Long(R[n]+R[0],R[m]);
    PC+=2;
}

MOVBL0(long m, long n) /* MOV.B @(R0,Rm), Rn */
{
    R[n]=(long)Read(Byte(R[m]+R[0]));
    if (((R[n]&0x80)==0) R[n]|=0x000000FF;
else R[n]|=0xFFFFFF00;
    PC+=2;
}

MOVWL0(long m, long n) /* MOV.W @(R0,Rm), Rn */
{
    R[n]=(long)Read_Word(R[m]+R[0]);
}
if ((R[n] & 0x8000) == 0) R[n] &= 0x0000FFFF;
else R[n] |= 0xFFFF0000;
PC+=2;
}

MOVLL0(long m, long n) /* MOV.L @(R0,Rm),Rn */
{
    R[n] = Read_Long(R[m] + R[0]);
    PC+=2;
}

Example:

MOV R0, R1
    ; Before execution:  R0 = H'FFFFFFFF, R1 = H'00000000
    ; After execution:   R1 = H'FFFFFFFF

MOV.W R0, @R1
    ; Before execution:  R0 = H'FFFF7F80
    ; After execution:   @R1 = H'7F80

MOV.B @R0, R1
    ; Before execution:  @R0 = H'80, R1 = H'00000000
    ; After execution:   R1 = H'FFFFFF80

MOV.W R0, @R1
    ; Before execution:  R0 = H'AAAAAAAA, R1 = H'FFFF7F80
    ; After execution:   R1 = H'FFFF7F7E, @R1 = H'AAAA

MOV.L @R0+, R1
    ; Before execution:  R0 = H'12345670
    ; After execution:   R0 = H'12345674, R1 = @H'12345670

MOV.B R1, @(R0,R2)
    ; Before execution:  R2 = H'00000004, R0 = H'10000000
    ; After execution:   R1 = @H'10000004

MOV.W @(R0,R2), R1
    ; Before execution:  R2 = H'00000004, R0 = H'10000000
    ; After execution:   R1 = @H'10000004
6.4.31  MOV
Immediate Data
Transfer

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #imm,Rn</td>
<td>imm → sign extension → Rn</td>
<td>1110nnnniiiiii</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @(disp, PC),Rn</td>
<td>(disp × 2 + PC) → sign extension → Rn</td>
<td>1001nnnnnnnnnnn</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @(disp, PC),Rn</td>
<td>(disp × 4 + PC) → Rn</td>
<td>1101nnnnnnnnnnn</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

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

If the data is a word or longword, table data stored in the address specified by PC + displacement is accessed. If the data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, the relative interval from the table can be up to PC + 510 bytes. The PC points to the starting address of the fourth byte after this MOV instruction. If the data is a longword, the 8-bit displacement is zero-extended and quadrupled. Consequently, the relative interval from the table can be up to PC + 1020 bytes. The PC points to the starting address of the fourth byte after this MOV instruction, but the lowest two bits of the PC are corrected to B'00.

**Note**

The optimum table assignment is at the rear end of the module or one instruction after the unconditional branch instruction. If the optimum assignment is impossible for the reason of no unconditional branch instruction in the 510 byte/1020 byte or some other reason, means to jump past the table by the BRA instruction are required. By assigning this instruction immediately after the delayed branch instruction, the PC becomes the "first address + 2".

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×2, ×4) as displacement values.
Operation

MOVI(long i, long n)  /* MOV #imm,Rn */
{
    if ((i&0x80)==0) R[n]=(0x000000FF & (long)i);
    else R[n]=(0xFFFFFF00 | (long)i);
    PC+=2;
}

MOVWI(long d, long n)  /* MOV.W @(disp,PC),Rn */
{
    long disp;
    disp=(0x000000FF & (long)d);
    R[n]=(long)Read_Word(PC+(disp<<1));
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}

MOVLI(long d, long n)  /* MOV.L @(disp,PC),Rn */
{
    long disp;
    disp=(0x000000FF & (long)d);
    R[n]=Read_Long((PC&0xFFFFFFFC)+(disp<<2));
    PC+=2;
}
Example:

Address

1000  MOV  #H'80,R1  ; R1 = H'FFFFFFFF80
1002  MOV.W IMM,R2  ; R2 = H'FFFF9ABC, IMM means @(H'08,PC)
1004  ADD  #-1,R0  ;
1006  TST  R0,R0  ; ← PC location used for address calculation for the MOV.W instruction
1008  MOVT  R13  ;
100A  BRA  NEXT  ; Delayed branch instruction
100C  MOV.L @(4,PC),R3  ; R3 = H'12345678
100E  IMM  .data.w  H'9ABC  ;
1010  .data.w  H'1234  ;
1012  NEXT  JMP  @R3  ; Branch destination of the BRA instruction
1014  CMP/EQ  #0,R0  ; ← PC location used for address calculation for the MOV.L instruction
1018  .data.l  H'12345678  ;
### 6.4.32 MOV MOVe peripheral Data Data Transfer

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.B @(disp,GBR),R0</td>
<td>(disp + GBR) → sign extension → R0</td>
<td>11000100dddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @(disp,GBR),R0</td>
<td>(disp × 2 + GBR) → sign extension → R0</td>
<td>11000101dddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @(disp,GBR),R0</td>
<td>(disp × 4 + GBR) → R0</td>
<td>11000110dddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B R0,@(disp,GBR)</td>
<td>R0 → (disp + GBR)</td>
<td>11000000dddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W R0,@(disp,GBR)</td>
<td>R0 → (disp × 2 + GBR)</td>
<td>11000001dddddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L R0,@(disp,GBR)</td>
<td>R0 → (disp × 4 + GBR)</td>
<td>11000010dddddddd</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

#### Description

Transfers the source operand to the destination. This instruction is optimum for accessing data in the peripheral module area. The data can be a byte, word, or longword, but only the R0 register can be used.

A peripheral module base address is set to the GBR. When the peripheral module data is a byte, the only change made is to zero-extend the 8-bit displacement. Consequently, an address within +255 bytes can be specified. When the peripheral module data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, an address within +510 bytes can be specified. When the peripheral module data is a longword, the 8-bit displacement is zero-extended and is quadrupled. Consequently, an address within +1020 bytes can be specified. If the displacement is too short to reach the memory operand, the above @(R0,Rn) mode must be used after the GBR data is transferred to a general register. When the source operand is in memory, the loaded data is stored in the register after it is sign-extended to a longword.

#### Note

The destination register of a data load is always R0. R0 cannot be accessed by the next instruction until the load instruction is finished. The instruction order shown in figure 6.1 will give better results.

```
MOV.B @(12, GBR), R0
AND #80, R0
ADD #20, R1
ADD #20, R1
AND #80, R0
```

**Figure 6.1** Using R0 after MOV
For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×1, ×2, ×4) as displacement values.

**Operation**

```c
MOVBLG(long d) /* MOV.B @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Byte(GBR+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}

MOVWLG(long d) /* MOV.W @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Word(GBR+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLLG(long d) /* MOV.L @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=Read_Long(GBR+(disp<<2));
    PC+=2;
}

MOVBSG(long d) /* MOV.B R0,@(disp,GBR) */
{
    long disp;
```
disp=(0x000000FF & (long)d);
Write_BYTE(GBR+disp,R[0]);
PC+=2;
}

MOVWSG(long d) /* MOV.W R0,@(disp,GBR) */
{
  long disp;

disp=(0x000000FF & (long)d);
Write_Word(GBR+(disp<<1),R[0]);
PC+=2;
}

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

disp=(0x000000FF & (long)d);
Write_Long(GBR+(disp<<2),R[0]);
PC+=2;
}

Examples:

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

MOV.B R0,@(1,GBR) ; Before execution: R0 = H'FFFF7F80
                      ; After execution: @(GBR + 1) = H'FFFF7F80
6.4.33 MOV

MOV structure data Data Transfer Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.B R0, @(disp, Rn)</td>
<td>R0 → (disp + Rn)</td>
<td>10000000nnnndddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W R0, @(disp, Rn)</td>
<td>R0 → (disp × 2 + Rn)</td>
<td>10000001nnnndddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L Rm, @(disp, Rn)</td>
<td>Rm → (disp × 4 + Rn)</td>
<td>0001nnnnmmmmdddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.B @(disp, Rm), R0</td>
<td>(disp + Rm) → sign extension → R0</td>
<td>10000100mmmmdddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.W @(disp, Rm), R0</td>
<td>(disp × 2 + Rm) → sign extension → R0</td>
<td>10000101mmmmdddddd</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>MOV.L @(disp, Rm), Rn</td>
<td>disp × 4 + Rm → Rn</td>
<td>0101nnnnmmmmdddddd</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

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

Note

When byte or word data is loaded, the destination register is always R0. R0 cannot be accessed by the next instruction until the load instruction is finished. The instruction order in figure 6.2 will give better results.

```
MOV.B @(2, R1), R0
AND #80, R0
ADD #20, R1
```

For the Renesas Technology SuperH RISC engine assembler, declarations should use scaled values (×1, ×2, ×4) as displacement values.
Operation

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

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

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

    disp=(0x0000000F & (long)d);
    Write_Word(R[n]+(disp<<1),R[0]);
    PC+=2;
}

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

    disp=(0x0000000F & (long)d);
    Write_Long(R[n]+(disp<<2),R[m]);
    PC+=2;
}

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

    disp=(0x0000000F & (long)d);
    R[0]=Read_Byte(R[m]+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}
MOVWL4(long m,long d) /* MOV.W @(disp,Rm),R0 */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[0]=Read_Word(R[m]+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLL4(long m,long d,long n) /* MOV.L @(disp,Rm),Rn */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[n]=Read_Long(R[m]+(disp<<2));
    PC+=2;
}

Examples:

MOVL @(2,R0),R1 ; Before execution: @(R0 + 8) = H'12345670
    ; After execution:    R1 = H'12345670
MOVL R0,@(H'F,R1) ; Before execution:    R0 = H'FFFF7F80
    ; After execution:    @(R1 + 60) = H'FFFF7F80
### 6.4.34 MOVA

#### MOV
effective Address
Transfer

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVA @(disp,PC),R0</td>
<td>disp x 4 + PC → R0</td>
<td>11000111ddddddd</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Stores the effective address of the source operand into general register R0. The 8-bit displacement is zero-extended and quadrupled. Consequently, the relative interval from the operand is PC + 1020 bytes. The PC is the address four bytes after this instruction, but the lowest two bits of the PC are corrected to B'00.

**Note**

If this instruction is placed immediately after a delayed branch instruction, the PC must point to an address specified by (the starting address of the branch destination) + 2.

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×4) as displacement values.

**Operation**

```c
MOVA(long d) /* MOVA @(disp,PC),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(PC&0xFFFFFFFC)+(disp<<2);
    PC+=2;
}
```
Example:
Address .org H'1006
1006 MOVA STR,R0 ; Address of STR → R0
1008 MOV.B @R0,R1 ; R1 = “X” ← PC location after correcting the lowest two bits
100A ADD R4,R5 ; ← Original PC location for address calculation for the
               MOVA instruction
                  .align 4
100C STR: .sdata “XYZP12”

 .............
2002 BRA TRGET ; Delayed branch instruction
2004 MOVA @(0,PC),R0 ; Address of TRGET + 2 → R0
2006 NOP ;
6.4.35 MOVT MOVe T bit Data Transfer Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVT</td>
<td>Rn</td>
<td>T → Rn</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000nnnn00101001</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

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

**Operation**

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

**Example:**

```c
XOR R2,R2 ;R2 = 0
CMP/PZ R2 ;T = 1
MOVT R0 ;R0 = 1
CLRT ;T = 0
MOVT R1 ;R1 = 0
```
6.4.36 MUL.L Multiply Long Arithmetic Instruction

**Double-Precision Multiplication**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.L</td>
<td>Rm,Rn</td>
<td>Rn × Rm → MACL</td>
<td>0000nnnnnnnnmm0111</td>
<td>2</td>
</tr>
</tbody>
</table>

**Description**
Performs 32-bit multiplication of the contents of general registers Rn and Rm, and stores the bottom 32 bits of the result in the MACL register. The MACH register data does not change.

**Operation**

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

**Example:**

```c
MULL R0,R1 ; Before execution: R0 = H'FFFFFFFE, R1 = H'00005555
 ; After execution: MACL = H'FFFF5556
STS MACL,R0 ; Operation result
```
### 6.4.37 MULS.W

**MULtiply as Signed Word**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULS.W</td>
<td>Rm,Rn</td>
<td>Signed operation, $Rn \times Rm \rightarrow MACL$</td>
<td>0010nnnnnnmm1111</td>
<td>1</td>
</tr>
<tr>
<td>MULS</td>
<td>Rm,Rn</td>
<td>Signed operation, $Rn \times Rm \rightarrow MACL$</td>
<td>0010nnnnnnmm1111</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

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

**Operation**

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

**Example:**

```
MULS R0,R1 ; Before execution: R0 = H'FFFFFFFE, R1 = H'00005555
 ; After execution: MACL = H'FFFF5556
STS MACL,R0 ; Operation result
```
### 6.4.38 MULU.W

**MULtiply as Unsigned Word**

**Arithmetic Instruction**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULU.W</td>
<td>Rm,Rn</td>
<td>Unsigned, Rm × Rn → MACL</td>
<td>0010nnnnnnnmm1110</td>
<td>1</td>
</tr>
<tr>
<td>MULU</td>
<td>Rm,Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

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

**Operation**

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

**Example:**

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

---

**REJ09B0051-0300**

Rev. 3.00 Jul 08, 2005 page 233 of 484

REJ09B0051-0300
### 6.4.39 NEG NEGate Arithmetic Instruction

#### Format

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>Rm,Rn</td>
<td>0 – Rm → Rn</td>
<td>0110nnnnnnnnm1011</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Description

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

#### Operation

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

#### Example:

```
NEG R0,R1 ; Before execution: R0 = H'00000001
; After execution: R1 = H'FFFFFFFF
```
6.4.40 NEGC  NEGate with Carry  Arithmetic Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEGC</td>
<td>Rm,Rn</td>
<td>0 – Rm – T → Rn, Borrow → T</td>
<td>0110nnnnnnnn1010</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

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

**Operation**

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

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

**Examples:**

- CLRT ; Sign inversion of R1 and R0 (64 bits)
- NEGC R1,R1 ; Before execution: R1 = H'00000001, T = 0
  ; After execution: R1 = H'FFFFFFFF, T = 1
- NEGC R0,R0 ; Before execution: R0 = H'00000000, T = 1
  ; After execution: R0 = H'FFFFFFFF, T = 1
6.4.41 NOP  No OPeration  System Control Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>0000000000001001</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Increments the PC to execute the next instruction.

**Operation**

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

**Example:**

```c
NOP ; Executes in one cycle
```
6.4.42  NOT  NOT-logical complement  Logical Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT Rm,Rn</td>
<td>~Rm → Rn</td>
<td>0110nnnnnnnnn0111</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Description**

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

**Operation**

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

**Example:**

```
NOT R0,R1 ; Before execution:  R0 = H'AAAAAAAA
; After execution:  R1 = H'55555555
```
6.4.43 OR Logical OR

### Logical OR

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th></th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR Rm,Rn</td>
<td>Rn</td>
<td>Rm → Rn</td>
<td>0010nnnnnnnnnnnnnnn1011</td>
<td>1</td>
</tr>
<tr>
<td>OR #imm,R0</td>
<td>R0</td>
<td>imm → R0</td>
<td>1100011111111111</td>
<td>1</td>
</tr>
<tr>
<td>OR.B #imm,@(R0,GBR)</td>
<td>(R0 + GBR)</td>
<td>imm → (R0 + GBR)</td>
<td>1100011111111111</td>
<td>3</td>
</tr>
</tbody>
</table>

**Description**

Logically ORs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can also be ORed with zero-extended 8-bit immediate data, or 8-bit memory data accessed by using indirect indexed GBR addressing can be ORed with 8-bit immediate data.

**Operation**

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

ORI(long i) /* OR #imm,R0 */
{
    R[0]=(0x000000FF & (long)i);
    PC+=2;
}

ORM(long i) /* OR.B #imm,@(R0,GBR) */
{
    long temp;

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

```
OR    R0,R1          ; Before execution:  R0 = H'AAAA5555,  R1 = H'55550000
       ; After execution:    R1 = H'FFFF5555

OR    #H'F0,R0       ; Before execution:  R0 = H'00000008
       ; After execution:    R0 = H'000000F8

OR.B  #H'50,@(R0,GBR) ; Before execution: @(R0,GBR) = H'A5
       ; After execution:    @(R0,GBR) = H'F5
```
6.4.44 ROTCL

ROTate with Carry Left
through T Bit

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTCL</td>
<td>Rn</td>
<td>T ← Rn ← T</td>
<td>0100nnnn001000100</td>
<td>1</td>
</tr>
</tbody>
</table>

Description

Rotates the contents of general register Rn and the T bit to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.3).

Operation

\[
\text{ROTCL}(\text{long } n)/* \text{ ROTCL Rn } */
\]

\{
    \text{long temp;}

    if ((R[n]\&0x80000000)==0) temp=0;
    else temp=1;
    R[n]<<=1;
    if (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
\}

Example:

\[
\text{ROTCL R0} ; \text{Before execution: } R0 = H'80000000, T = 0
\]

\[
\text{After execution: } R0 = H'00000000, T = 1
\]
6.4.45 ROTCR

ROTate with Carry Right Shift Instruction

One-Bit Right Rotation through T Bit

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTCR</td>
<td>Rn</td>
<td>T → Rn → T</td>
<td>0100nnnn00100101</td>
<td>1</td>
</tr>
</tbody>
</table>

Description

Rotates the contents of general register Rn and the T bit to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.4).

![Figure 6.4 Rotate with Carry Right](image)

Operation

ROTCR(long n) /* ROTCR Rn */
{
    long temp;

    if ((R[n]&0x00000001)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (T==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
}

Examples:

ROTCR R0  ; Before execution: R0 = H'00000001, T = 1
            ; After execution: R0 = H'80000000, T = 1
6.4.46 ROTL ROTate Left Shift Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract Code</th>
<th>Cycle</th>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTL</td>
<td>Rn T ← Rn ← MSB</td>
<td>0100nnnn0000100</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Rotates the contents of general register Rn to the left by one bit, and stores the result in Rn (figure 6.5). The bit that is shifted out of the operand is transferred to the T bit.

![Figure 6.5 Rotate Left](image)

**Operation**

ROTL(long n) /* ROTL Rn */
{
   if ((R[n] & 0x80000000) == 0) T = 0;
   else T = 1;
   R[n] <<= 1;
   if (T == 1) R[n] |= 0x00000001;
   else R[n] &= 0xFFFFFFFE;
   PC += 2;
}

**Examples:**

ROTL R0
; Before execution: R0 = H'80000000, T = 0
; After execution: R0 = H'00000001, T = 1
6.4.47  ROTR  ROTate Right  Shift Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTR</td>
<td>Rn</td>
<td>0100nnnn00000101</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Rotates the contents of general register Rn to the right by one bit, and stores the result in Rn (figure 6.6). The bit that is shifted out of the operand is transferred to the T bit.

**Operation**

```
ROTR(long n) /* ROTR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    if (T==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

**Examples:**

ROTR R0  ; Before execution: R0 = H'00000001, T = 0
; After execution: R0 = H'80000000, T = 1
6.4.48 RTE ReTurn from Exception System Control Instruction
Return from Exception Handling Delayed Branch Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTE</td>
<td>Delayed branch, Stack area → PC/SR</td>
<td>0000000000101011</td>
<td>4</td>
<td>LSB</td>
</tr>
</tbody>
</table>

**Description**

Returns from an interrupt routine. The PC and SR values are restored from the stack, and the program continues from the address specified by the restored PC value. The T bit is used as the LSB bit in the SR register restored from the stack area.

**Note**

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

**Operation**

```c
RTE() /* RTE */
{
    unsigned long temp;

    temp=PC;
    PC=Read_Long(R[15])+4;
    R[15]+=4;
    SR=Read_Long(R[15])&0x000063F3;
    R[15]+=4;
    Delay_Slot(temp+2);
}
```
Example:

RTE ; Returns to the original routine
ADD #8, R14 ; Executes ADD before branching

Note: When a delayed branch instruction is used, the branching operation takes place after the
slot instruction is executed, but the execution of instructions (register update, etc.) takes
place in the sequence delayed branch instruction → delayed slot instruction. For example,
even if a delayed slot instruction is used to change the register where the branch
destination address is stored, the register content previous to the change will be used as the
branch destination address.
Description

Returns from a subroutine procedure. The PC values are restored from the PR, and the program continues from the address specified by the restored PC value. This instruction is used to return to the program from a subroutine program called by a BSR, BSRF, or JSR instruction.

Note

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

Operation

```c
RTS() /* RTS */
{
    unsigned long temp;

    temp=PC;
    PC=PR+4;
    Delay_Slot(temp+2);
}
```
Example:

```asm
MOV.L   TABLE,R3  ; R3 = Address of TRGET
JSR    @R3       ; Branches to TRGET
NOP     ; Executes NOP before branching
ADD     R0,R1    ; ← Return address for when the subroutine procedure is completed (PR data)

............

TABLE:  .data.l TRGET;

............
TRGET:   MOV     R1,R0  ; ← Procedure entrance
         RTS       ; PR data → PC
         MOV     #12,R0 ;

Executes MOV before branching
```

Note: When a delayed branch instruction is used, the branching operation takes place after the slot instruction is executed, but the execution of instructions (register update, etc.) takes place in the sequence delayed branch instruction → delayed slot instruction. For example, even if a delayed slot instruction is used to change the register where the branch destination address is stored, the register content previous to the change will be used as the branch destination address.
6.4.50 SETT T Bit Setting

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETT</td>
<td>( 1 \rightarrow T )</td>
<td>0000000000011000</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Sets the T bit to 1.

**Operation**

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

**Example:**

```
SETT ; Before execution: T = 0
        ; After execution: T = 1
```
6.4.51 SHAL SHift Arithmetic Left Shift Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAL Rn</td>
<td>T ← Rn ← 0</td>
<td>0100nnnn00100000</td>
<td>1</td>
<td>MSB</td>
</tr>
</tbody>
</table>

Description

Arithmetically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.7).

```
Figure 6.7 Shift Arithmetic Left
```

Operation

```
SHAL(long n) /* SHAL Rn (Same as SHLL) */
{
   if ((R[n]&0x80000000)==0) T=0;
   else T=1;
   R[n]<<=1;
   PC+=2;
}
```

Example:

```
SHAL R0 ; Before execution: R0 = H'80000001, T = 0
; After execution: R0 = H'00000002, T = 1
```
### 6.4.52 SHAR SHift Arithmetic Right

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAR Rn</td>
<td>MSB → Rn → T</td>
<td>0100nnnn00100001</td>
<td>1</td>
<td>LSB</td>
</tr>
</tbody>
</table>

#### Description
Arithmetically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.8).

#### Operation
```
SHAR(long n) /* SHAR Rn */
{
    long temp;

    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]>>=1;
    if (temp==1) R[n]|=0x80000000;
    else R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

#### Example:
```
SHAR R0 ; Before execution: R0 = H'80000001, T = 0
          ; After execution: R0 = H'C0000000, T = 1
```
6.4.53 SHLL

SHift Logical Left
Logical Shift

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLL</td>
<td>Rn</td>
<td>T ← Rn ← 0</td>
<td>0100nnnn00000000</td>
<td>1</td>
</tr>
</tbody>
</table>

Description

Logically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.9).

```
Figure 6.9 Shift Logical Left
```

Operation

```
SHLL(long n) /* SHLL Rn (Same as SHAL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

Examples:

```
SHLL R0  ; Before execution: R0 = H'80000001, T = 0
          ; After execution: R0 = H'00000002, T = 1
```
6.4.54 **SHLLn**

n-Bit Left Logical Shift

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLL2</td>
<td>Rn  Rn &lt;&lt; 2 → Rn</td>
<td>0100nnnn00001000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>SHLL8</td>
<td>Rn  Rn &lt;&lt; 8 → Rn</td>
<td>0100nnnn00011000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>SHLL16</td>
<td>Rn  Rn &lt;&lt; 16 → Rn</td>
<td>0100nnnn00101000</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Logically shifts the contents of general register Rn to the left by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 6.10).

![Shift Logical Left n Bits](#)

---

Rev. 3.00 Jul 08, 2005 page 252 of 484

REJ09B0051-0300

RENESAS
**Operation**

```
SHLL2(long n) /* SHLL2 Rn */
{
    R[n] <<= 2;
    PC += 2;
}
SHLL8(long n) /* SHLL8 Rn */
{
    R[n] <<= 8;
    PC += 2;
}
SHLL16(long n) /* SHLL16 Rn */
{
    R[n] <<= 16;
    PC += 2;
}
```

**Examples:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLL2 R0</td>
<td>R0 = H’12345678</td>
<td>R0 = H’48D159E0</td>
</tr>
<tr>
<td>SHLL8 R0</td>
<td>R0 = H’12345678</td>
<td>R0 = H’34567800</td>
</tr>
<tr>
<td>SHLL16 R0</td>
<td>R0 = H’12345678</td>
<td>R0 = H’56780000</td>
</tr>
</tbody>
</table>
6.4.55 SHLR SHift Logical Right

One-Bit Right
Logical Shift

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLR</td>
<td>Rn</td>
<td>0 → Rn → T</td>
<td>0100nnnn00000001</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Logically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 6.11).

![Figure 6.11 Shift Logical Right](image)

**Operation**

```c
SHLR(long n) /* SHLR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

**Examples:**

- Before execution: R0 = H'80000001, T = 0
- After execution: R0 = H'40000000, T = 1
### 6.4.56 SHLRn n-Bit Right Logical Shift

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLR2</td>
<td>Rn Rn&gt;&gt;2 → Rn</td>
<td>0100nnnn00001001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>SHLR8</td>
<td>Rn Rn&gt;&gt;8 → Rn</td>
<td>0100nnnn00011001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>SHLR16</td>
<td>Rn Rn&gt;&gt;16 → Rn</td>
<td>0100nnnn00101001</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Logically shifts the contents of general register Rn to the right by 2, 8, or 16 bits, and stores the result in Rn. Bits that are shifted out of the operand are not stored (figure 6.12).

![Figure 6.12 Shift Logical Right n Bits](image-url)
Operation

SHLR2(long n) /* SHLR2 Rn */
{
    R[n] >>= 2;
    R[n] &= 0x3FFFFFFF;
    PC += 2;
}

SHLR8(long n) /* SHLR8 Rn */
{
    R[n] >>= 8;
    R[n] &= 0x00FFFFFF;
    PC += 2;
}

SHLR16(long n) /* SHLR16 Rn */
{
    R[n] >>= 16;
    R[n] &= 0x0000FFFF;
    PC += 2;
}

Examples:

SHLR2 R0  ; Before execution:  R0 = H'12345678
           ; After execution:    R0 = H'048D159E

SHLR8 R0  ; Before execution:  R0 = H'12345678
           ; After execution:    R0 = H'00123456

SHLR16 R0 ; Before execution:  R0 = H'12345678
           ; After execution:    R0 = H'00001234
6.4.57 SLEEP SLEEP System Control Instruction

Transition to Power-Down Mode

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEEP</td>
<td>Sleep</td>
<td>0000000000011011</td>
<td>5</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Sets the CPU into power-down mode. In power-down mode, instruction execution stops, but the CPU internal status is maintained, and the CPU waits for an interrupt request. If an interrupt is requested, the CPU exits the power-down mode and begins exception processing.

Note

The number of cycles given is for the transition to sleep mode.

Operation

```c
SLEEP() /* SLEEP */
{
    wait_for_exception;
}
```

Example:

```c
SLEEP ; Enters power-down mode
```
## 6.4.58 STC STore Control register System Control Instruction

### Store from Control Register

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>STC SR,Rn</td>
<td>SR → Rn</td>
<td>0000nnnn00000010</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>STC GBR,Rn</td>
<td>GBR → Rn</td>
<td>0000nnnn00010010</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>STC VBR,Rn</td>
<td>VBR → Rn</td>
<td>0000nnnn00100010</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>STCL SR,@-Rn</td>
<td>Rn – 4 → Rn, SR → (Rn)</td>
<td>0100nnnn00000011</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>STCL GBR,@-Rn</td>
<td>Rn – 4 → Rn, GBR → (Rn)</td>
<td>0100nnnn00010011</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>STCL VBR,@-Rn</td>
<td>Rn – 4 → Rn, VBR → (Rn)</td>
<td>0100nnnn00100011</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Description

Stores control register SR, GBR, or VBR data into a specified destination.

### Operation

```c
STCSR(long n) /* STC SR,Rn */
{
    R[n]=SR;
    PC+=2;
}
STCGBR(long n) /* STC GBR,Rn */
{
    R[n]=GBR;
    PC+=2;
}
STCVBR(long n) /* STC VBR,Rn */
{
    R[n]=VBR;
    PC+=2;
}
STCMSR(long n) /* STC.L SR,@-Rn */
{
    R[n]=4;
    Write_Long(R[n],SR);
}
```

Rev. 3.00 Jul 08, 2005 page 258 of 484
REJ09B0051-0300

RENESAS
PC+=2;
}
STCMGBR(long n) /* STC.L GBR,\$-Rn */
{
    R[n]-=4;
    Write_Long(R[n],GBR);
    PC+=2;
}
STCMVBR(long n) /* STC.L VBR,\$-Rn */
{
    R[n]-=4;
    Write_Long(R[n],VBR);
    PC+=2;
}

Examples:
STC SR,R0 ; Before execution: R0 = H'FFFFFFFF, SR = H'00000000
    ; After execution: R0 = H'00000000
STC.L GBR,\$-R15 ; Before execution: R15 = H'10000004
    ; After execution: R15 = H'10000000, @R15 = GBR
6.4.59 STS STore System register System Control Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS</td>
<td>MACH,Rn</td>
<td>MACH → Rn</td>
<td>0000nnnn00001010</td>
<td>1</td>
</tr>
<tr>
<td>STS</td>
<td>MACL,Rn</td>
<td>MACL → Rn</td>
<td>0000nnnn00011010</td>
<td>1</td>
</tr>
<tr>
<td>STS</td>
<td>PR,Rn</td>
<td>PR → Rn</td>
<td>0000nnnn00101010</td>
<td>1</td>
</tr>
<tr>
<td>STS.L</td>
<td>MACH,@–Rn</td>
<td>Rn – 4 → Rn, MACH → (Rn)</td>
<td>0100nnnn00000010</td>
<td>1</td>
</tr>
<tr>
<td>STS.L</td>
<td>MACL,@–Rn</td>
<td>Rn – 4 → Rn, MACL → (Rn)</td>
<td>0100nnnn00010010</td>
<td>1</td>
</tr>
<tr>
<td>STS.L</td>
<td>PR,@–Rn</td>
<td>Rn – 4 → Rn, PR → (Rn)</td>
<td>0100nnnn00100010</td>
<td>1</td>
</tr>
</tbody>
</table>

Description
Stores data from system register MACH, MACL, or PR into a specified destination.

Operation

```c
STSMACH(long n) /* STS MACH,Rn */
{
    R[n]=MACH;
    PC+=2;
}
STSMACL(long n) /* STS MACL,Rn */
{
    R[n]=MACL;
    PC+=2;
}
STSPR(long n) /* STS PR,Rn */
{
    R[n]=PR;
    PC+=2;
}
STSMMMACH(long n) /* STS.L MACH,@–Rn */
{
    R[n]=–4;
}
```
Write_Long(R[n], MACH);
PC+=2;
}

STSMMACL(long n) /* STS.L MAACL,@–Rn */
{
   R[n]−=4;
   Write_Long(R[n], MACL);
   PC+=2;
}

STSMPR(long n) /* STS.L PR,@–Rn */
{
   R[n]−=4;
   Write_Long(R[n], PR);
   PC+=2;
}

Example:

STS    MACH, R0    ; Before execution: R0 = H'FFFFFFFF, MACH = H'00000000
        ; After execution: R0 = H'00000000
STS.L  PR, @R15   ; Before execution: R15 = H'10000004
        ; After execution: R15 = H'10000000, @R15 = PR
### 6.4.60 SUB SUBtract binary Arithmetic Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>Rm,Rn</td>
<td>Rn – Rm → Rn</td>
<td>0011nnnnnnnn1000</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Subtracts general register Rm data from Rn data, and stores the result in Rn. To subtract immediate data, use ADD #imm,Rn.

**Operation**

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

**Example:**

```c
SUB R0,R1 ; Before execution: R0 = H'00000001, R1 = H'80000000
; After execution: R1 = H'7FFFFFFF
```
6.4.61  SUBC  SUBtract with Carry  
Binary Subtraction with Borrow

Format  Abstract  Code  Cycle  T Bit
---  ---  ---  ---  ---
SUBC  Rm,Rn  Rn – Rm→ T → Rn, Borrow → T  0011nnnnnnnn1010  1  Borrow

Description
Subtracts Rm data and the T bit value from general register Rn data, and stores the result in Rn. The T bit changes according to the result. This instruction is used for subtraction of data that has more than 32 bits.

Operation
```c
SUBC(long m,long n) /* SUBC Rm,Rn */
{
    unsigned long tmp0,tmp1;

    tmp1=R[n]-R[m];
    tmp0=R[n];
    R[n]=tmp1-T;
    if (tmp0<tmp1) T=1;
    else T=0;
    if (tmp1<R[n]) T=1;
    PC+=2;
}
```

Examples:
- CLRT ; R0:R1(64 bits) – R2:R3(64 bits) = R0:R1(64 bits)
- SUBC R3,R1 ; Before execution: T = 0, R1 = H'00000000, R3 = H'00000001
  ; After execution: T = 1, R1 = H'FFFFFFFF
- SUBC R2,R0 ; Before execution: T = 1, R0 = H'00000000, R2 = H'00000000
  ; After execution: T = 1, R0 = H'FFFFFFFF
6.4.62 SUBV SUBtract with (V flag) underflow check Arithmetic Instruction

Binary Subtraction with Underflow Check

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBV</td>
<td>Rm,Rn</td>
<td>Rn – Rm → Rn, underflow → T</td>
<td>0011nnnnnnnnm1011</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Subtracts Rm data from general register Rn data, and stores the result in Rn. If an underflow occurs, the T bit is set to 1.

**Operation**

```c
SUBV(long m,long n) /* SUBV Rm,Rn */
{
    long dest,src,ans;

    if ((long)R[n]>=0) dest=0;
    else dest=1;
    if ((long)R[m]>=0) src=0;
    else src=1;
    src+=dest;
    R[n]-=R[m];
    if ((long)R[n]>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (src==1) {
        if (ans==1) T=1;
        else T=0;
    } else T=0;
    PC+=2;
}
```
Examples:

```
SUBV R0, R1 ; Before execution: R0 = H'00000002, R1 = H'80000001
            ; After execution:  R1 = H'7FFFFFFF, T = 1
SUBV R2, R3 ; Before execution: R2 = H'FFFFFFFE, R3 = H'7FFFFFFE
            ; After execution:  R3 = H'80000000, T = 1
```
Section 6  Instruction Descriptions

6.4.63  SWAP   Upper-/Lower-Half Swap

### Format

- **SWAP.B**  
  - **Abstract:** Rm → Swap upper and lower halves of lower 2 bytes → Rn
  - **Code:** 0110nnnnnnnn1000
  - **Cycle:** 1
  - **T Bit:** —

- **SWAP.W**  
  - **Abstract:** Rm → Swap upper and lower word → Rn
  - **Code:** 0110nnnnnnnn1001
  - **Cycle:** 1
  - **T Bit:** —

### Description

Swaps the upper and lower bytes of the general register Rm data, and stores the result in Rn. If a byte is specified, bits 0 to 7 of Rm are swapped for bits 8 to 15. The upper 16 bits of Rm are transferred to the upper 16 bits of Rn. If a word is specified, bits 0 to 15 of Rm are swapped for bits 16 to 31.

### Operation

**SWAPB(long m,long n) /* SWAP.B Rm,Rn */**

```c
{ 
    unsigned long temp0,temp1;

    temp0=R[m] & 0xffff0000;
    temp1=(R[m] & 0x000000ff) << 8;
    R[n] = (R[m] >> 8) & 0x000000ff;
    R[n] = R[n] | temp1 | temp0;
    PC+=2;
}
```

**SWAPW(long m,long n) /* SWAP.W Rm,Rn */**

```c
{ 
    unsigned long temp;
    temp=(R[m] >> 16) & 0x0000FFFF;
    R[n] = R[m] << 16;
    R[n] |= temp;
    PC+=2;
}
```
Examples:

```
SWAP.B R0,R1  ; Before execution: R0 = H'12345678
              ; After execution:  R1 = H'12347856
SWAP.W R0,R1  ; Before execution: R0 = H'12345678
              ; After execution:  R1 = H'56781234
```
### 6.4.64 TAS Test And Set Logical Instruction

#### Memory Test and Bit Setting

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAS.B @Rn</td>
<td>When (Rn) is 0 → T, 1 → MSB of (Rn)</td>
<td>0100nnnn00011011</td>
<td>3</td>
<td>Test results</td>
</tr>
</tbody>
</table>

**Description**

Reads byte data from the address specified by general register Rn, and sets the T bit to 1 if the data is 0, or clears the T bit to 0 if the data is not 0. Then, data bit 7 is set to 1, and the data is written to the address specified by Rn. During this operation, the bus is not released.

**Operation**

```c
TAS(long n) /* TAS.B @Rn */
{
    long temp;

    temp=(long)Read_Byte(R[n]);      /* Bus Lock enable */
    if (temp==0) T=1;
    else T=0;
    temp|=0x00000080;
    Write_Byte(R[n],temp);           /* Bus Lock disable */
    PC+=2;
}
```

**Example:**

```
_LOOP TAS.B @R7 ; R7 = 1000
BF _LOOP ; Loops until data in address 1000 is 0
```
## 6.4.65 TRAPA TRAP Always System Control Instruction

### Trap Exception Handling

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAPA</td>
<td>#imm PC/SR → Stack area, (imm × 4 + VBR) → PC</td>
<td>11000011iiiiiiii</td>
<td>5</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

Starts the trap exception processing. The PC and SR values are stored on the stack, and the program branches to an address specified by the vector. The vector is a memory address obtained by zero-extending the 8-bit immediate data and then quadrupling it. The PC is the start address of the next instruction. TRAPA and RTE are both used together for system calls.

**Note**

For the Renesas Technology Super H RISC engine assembler, declarations should use scaled values (×4) as displacement values.

**Operation**

```c
TRAPA(long i) /* TRAPA #imm */
{
    long imm;

    imm=(0x000000FF & i);
    R[15]=4;
    Write_Long(R[15],SR);
    R[15]=4;
    Write_Long(R[15],PC–2);
    PC=Read_Long(VBR+(imm<<2))+4;
}
```
Example:

Address
VBR+H'80 .data.1 10000000;
........
    TRAPA #H'20 ; Branches to an address specified by data in address VBR + H'80
    TST #0,R0  ; ← Return address from the trap routine (stacked PC value)
........

........
100000000 XOR R0,R0  ; ← Trap routine entrance
100000002 RTE         ; Returns to the TST instruction
100000004 NOP         ; Executes NOP before RTE
### 6.4.66 TST

**TeST logical**

**Logical Instruction**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST</td>
<td>Rm,Rn</td>
<td>0010nnnnnnnnm1000</td>
<td>1</td>
<td>Test results</td>
</tr>
<tr>
<td>TST</td>
<td>#imm,R0</td>
<td>11001000iiiiiiii</td>
<td>1</td>
<td>Test results</td>
</tr>
<tr>
<td>TST.B</td>
<td>#(imm, @(R0,GBR))</td>
<td>11001100iiiiiiii</td>
<td>3</td>
<td>Test results</td>
</tr>
</tbody>
</table>

**Description**

Logically ANDs the contents of general registers Rn and Rm, and sets the T bit to 1 if the result is 0 or clears the T bit to 0 if the result is not 0. The Rn data does not change. The contents of general register R0 can also be ANDed with zero-extended 8-bit immediate data, or the contents of 8-bit memory accessed by indirect indexed GBR addressing can be ANDed with 8-bit immediate data. The R0 and memory data do not change.

**Operation**

```c
TST(long m,long n) /* TST Rm,Rn */
{
    if ((R[n]&R[m])==0) T=1;
    else T=0;
    PC+=2;
}
TSTI(long i) /* TEST #imm,R0 */
{
    long temp;

    temp=R[0]&(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}
TSTM(long i) /* TST.B #imm,@(R0,GBR) */
```
{  

    long temp;

    temp=(long)Read_Bar(GBR+R[0]);
    temp&=(0x000000FF & (long)i);
    if (temp==0) T=1;
    else T=0;
    PC+=2;
}

Examples:

TST    R0,R0          ; Before execution: R0 = H'00000000
        ; After execution: T = 1
TST    #H'80,R0        ; Before execution: R0 = H'FFFFFFFF
        ; After execution: T = 1
TST.B  #H'A5,@(R0,GBR) ; Before execution: @(R0,GBR) = H'A5
        ; After execution: T = 0
6.4.67 XOR eXclusive OR logical Logical Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR Rm,Rn</td>
<td>Rn ^ Rm → Rn</td>
<td>0010nnnnnnnn1010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>XOR #imm,R0</td>
<td>R0 ^ imm → R0</td>
<td>11001010iiiiiiii</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>XOR.B #imm, @(R0,GBR)</td>
<td>(R0 + GBR) ^ imm → (R0 + GBR)</td>
<td>11001110iiiiiiii</td>
<td>3</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

Exclusive ORs the contents of general registers Rn and Rm, and stores the result in Rn. The contents of general register R0 can also be exclusive ORed with zero-extended 8-bit immediate data, or 8-bit memory accessed by indirect indexed GBR addressing can be exclusive ORed with 8-bit immediate data.
Operation

XOR(long m,long n) /* XOR Rm,Rn */
{
    R[n]^=R[m];
    PC+=2;
}

XORI(long i) /* XOR #imm,R0 */
{
    R[0]^=(0x000000FF & (long)i);
    PC+=2;
}

XORM(long i) /* XOR.B #imm,@(R0,GBR) */
{
    long temp;

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

Examples:

XOR R0,R1 ; Before execution: R0 = H'AAAAAAAA, R1 = H'55555555
            ; After execution: R1 = H'FFFFFFFF

XOR #H'F0,R0 ; Before execution: R0 = H'FFFFFFFF
               ; After execution: R0 = H'FFFFFF0F

XOR.B #H'A5,@(R0,GBR) ; Before execution: @(R0,GBR) = H'A5
                       ; After execution: @(R0,GBR) = H'00
### 6.4.68 XTRCT eXTRaCT Data Transfer Instruction

#### Middle Extraction from Linked Registers

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTRCT</td>
<td>Rm,Rn</td>
<td>Rm: Center 32 bits of Rn → Rn</td>
<td>0010nnnnnnnnnnm1101</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**

Extracts the middle 32 bits from the 64 bits of coupled general registers Rm and Rn, and stores the 32 bits in Rn (figure 6.13).

**Operation**

```
XTRCT(long m,long n) /* XTRCT Rm,Rn */
{
    unsigned long temp;

    temp=(R[m]<<16)&0xFFFF0000;
    R[n]=(R[n]>>16)&0x0000FFFF;
    R[n] |= temp;
    PC+=2;
}
```

**Example:**

```
XTRCT R0,R1 ; Before execution: R0 = H'01234567, R1 = H'89ABCDEF
; After execution: R1 = H'456789AB
```
6.5 Floating-Point Instructions and FPU-Related CPU Instructions

6.5.1 FABS Floating-point ABSolute value Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FABS</td>
<td>FRn</td>
<td>1111nnnn01011101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FABS</td>
<td>DRn</td>
<td>1111nnnn001011101</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description
This instruction clears the most significant bit of the contents of floating-point register FRn/DRn to 0, and stores the result in FRn/DRn.

The cause and flag fields in FPSCR are not updated.

Operation

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

Possible Exceptions:
None
6.5.2 FADD Floating-point ADD Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FADD</td>
<td>FRm,FRn FRn+FRm → FRn</td>
<td>1111nnnmmm0000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FADD</td>
<td>DRm,DRn DRn+DRm → DRn</td>
<td>1111nnn0mmm00000</td>
<td>6</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0: Arithmetically adds the two single-precision floating-point numbers in FRm and FRn, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically adds the two double-precision floating-point numbers in DRm and DRn, and stores the result in DRn.

When FPSCR.enable.O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

**Operation**

```c
void FADD (int m,n)
{
    pc += 2;
    clear_cause();
    if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN)) invalid(n);
    else if((data_type_of(m) == qNaN) ||
        (data_type_of(n) == qNaN)) qnan(n);
    else if((data_type_of(m) == DENORM) ||
        (data_type_of(n) == DENORM)) set_E();
    else switch (data_type_of(m)){
        case NORM: switch (data_type_of(n)){
            case NORM: normal_faddsub(m,n,ADD); break;
            case FZERO:
```

```c
```
case NZERO: register_copy(m,n); break;
    default: break;
  }
  break;
}
case PZERO: switch (data_type_of(n)){
  case NZERO:    zero(n,0); break;
  default:      break;
  }
  break;
}
case NZERO:    break;
case PINF:    switch (data_type_of(n)){
  case NINF: invalid(n); break;
  default:    inf(n,0); break;
  }
  break;
}
case NINF:    switch (data_type_of(n)){
  case PINF: invalid(n); break;
  default:    inf(n,1); break;
  }
  break;
}

FADD Special Cases

<table>
<thead>
<tr>
<th>FRm,DRm</th>
<th>FRn,DRn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NORM</td>
</tr>
<tr>
<td>NORM</td>
<td>ADD</td>
</tr>
<tr>
<td></td>
<td>+0</td>
</tr>
<tr>
<td></td>
<td>-0</td>
</tr>
<tr>
<td></td>
<td>+INF</td>
</tr>
<tr>
<td></td>
<td>-INF</td>
</tr>
<tr>
<td>qNaN</td>
<td></td>
</tr>
<tr>
<td>sNaN</td>
<td></td>
</tr>
</tbody>
</table>

Note: When DN = 1, the value of a denormalized number is treated as 0.
Possible Exceptions:
- Invalid operation
- Overflow
- Underflow
- Inexact
6.5.3 FCMP Floating-point CoMPare Floating-Point Comparison

<table>
<thead>
<tr>
<th>No.</th>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>FCMP/EQ FRm,FRn</td>
<td>(FRn==FRm)?1:0 → T</td>
<td>1111nnnnmmmm0100</td>
<td>1</td>
<td>1/0</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>FCMP/EQ DRm,DRn</td>
<td>(DRn==DRm)?1:0 → T</td>
<td>1111nnnn00000100</td>
<td>2</td>
<td>1/0</td>
</tr>
<tr>
<td>3.</td>
<td>0</td>
<td>FCMP/GT FRm,FRn</td>
<td>(FRn&gt;FRm)?1:0 → T</td>
<td>1111nnnnmmmm0101</td>
<td>1</td>
<td>1/0</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>FCMP/GT DRm,DRn</td>
<td>(DRn&gt;DRm)?1:0 → T</td>
<td>1111nnnn000101</td>
<td>2</td>
<td>1/0</td>
</tr>
</tbody>
</table>

Description
1. When FPSCR.PR = 0: Arithmetically compares the two single-precision floating-point numbers in FRn and FRm, and stores 1 in the T bit if they are equal, or 0 otherwise.
2. When FPSCR.PR = 1: Arithmetically compares the two double-precision floating-point numbers in DRn and DRm, and stores 1 in the T bit if they are equal, or 0 otherwise.
3. When FPSCR.PR = 0: Arithmetically compares the two single-precision floating-point numbers in FRn and FRm, and stores 1 in the T bit if FRn > FRm, or 0 otherwise.
4. When FPSCR.PR = 1: Arithmetically compares the two double-precision floating-point numbers in DRn and DRm, and stores 1 in the T bit if DRn > DRm, or 0 otherwise.

Operation
```c
void FCMP_EQ(int m,n) /* FCMP/EQ  FRm,FRn */
{
    pc += 2;
    clear_cause();
    if(fcmp_chk (m,n) == INVALID) fcmp_invalid();
    else if(fcmp_chk (m,n) == EQ)  T = 1;
    else                           T = 0;
}
void FCMP_GT(int m,n) /* FCMP/GT  FRm,FRn */
{
    pc += 2;
    clear_cause();
    if ((fcmp_chk (m,n) == INVALID) ||
        (fcmp_chk (m,n) == UO)) fcmp_invalid();
```

Rev. 3.00  Jul 08, 2005  page 280 of 484
REJ09B0051-0300

Renesas
else if(fcmp_chk (m,n) == GT)  T = 1;
else                           T = 0;
}
int fcmp_chk (int m,n)
{
    if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN))        return(INVALID);
    else if((data_type_of(m) == qNaN) ||
            (data_type_of(n) == qNaN))      return(UO);
    else switch(data_type_of(m)){
        case NORM: switch(data_type_of(n)){
            case PINF     :return(GT);  break;
            case NINF     :return(LT);  break;
            default:                   break;
        }      break;
        case PZERO:
        case NZERO:  switch(data_type_of(n)){
            case PZERO   :
            case NZERO   :return(EQ);  break;
            default:                   break;
        }      break;
        case PINF :  switch(data_type_of(n)){
            case PINF    :return(EQ);  break;
            default:return(LT);       break;
        }      break;
        case NINF :  switch(data_type_of(n)){
            case NINF    :return(EQ);  break;
            default:return(GT);       break;
        }      break;
    }
    if(FPSCR_PR == 0) {
        if(FR[n] == FR[m])           return(EQ);
        else if(FR[n] > FR[m])       return(GT);
        else                         return(LT);
    }else {

if(DR[n>>1] == DR[m>>1]) return(EQ);
else if(DR[n>>1] > DR[m>>1]) return(GT);
else return(LT);
}
}

void fcmp_invalid()
{
    set_V();
    T = 0;
    if((FPSCR & ENABLE_V)==1) fpu_exception_trap();
}

**FCMP Special Cases**

<table>
<thead>
<tr>
<th>FCMP/EQ</th>
<th>FRn,DRn</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMP/EQ</td>
<td>NORM +0</td>
</tr>
<tr>
<td>FRm,DRm</td>
<td>NORM</td>
</tr>
<tr>
<td>NORM</td>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>qNaN</td>
<td></td>
</tr>
<tr>
<td>sNaN</td>
<td></td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

<table>
<thead>
<tr>
<th>FCMP/GT</th>
<th>FRn,DRn</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMP/GT</td>
<td>NORM +0</td>
</tr>
<tr>
<td>FRm,DRm</td>
<td>NORM</td>
</tr>
<tr>
<td>NORM</td>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>qNaN</td>
<td></td>
</tr>
<tr>
<td>sNaN</td>
<td></td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

UO means unordered. Unordered is treated as false (IGT).
Possible Exceptions:

Invalid operation
6.5.4 FCNVDS Floating-point CoNVert
Double to Single precision Floating-Point Instruction

Double-Precision to Single-Precision Conversion

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>FCNVDS DRm,FPUL (float)DRm → FPUL</td>
<td>1111mm010111101</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

When FPSCR.PR = 1, this instruction converts the double-precision floating-point number in DRm to a single-precision floating-point number, and stores the result in FPUL.

When FPSCR.enable.O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FPUL is not updated. Appropriate processing should therefore be performed by software.

If FPSCR.PR = 0, the instruction is handled as an illegal instruction.

Operation

```c
void FCNVDS(int m, float *FPUL){
    case((FPSCR.PR){
        0: undefined_operation(); /* reserved */
        1: fcnvds(m, *FPUL); break; /* FCNVDS */
    }
}
void fcnvds(int m, float *FPUL)
{
    pc += 2;
    clear_cause();
    case(data_type_of(m, *FPUL)){
        NORM :
        PZERO :
        NZERO : normal_fcnvds(m, *FPUL); break;
    }
}
```
PINF :  *FPUL = 0x7f800000; break;
NINF :  *FPUL = 0xff800000; break;
qNaN :  *FPUL = 0x7fbfffff; break;
sNaN :  set_V();
   if((FPSCR & ENABLE_V) == 0) *FPUL = 0x7fbfffff;
   else fpu_exception_trap(); break;
}
}
void normal_fcnvds(int m, float *FPUL)
{
int sign;
float abs;
union {
    float f;
    int l;
} dstf,tmpf;
union {
    double d;
    int l[2];
} dstd;
dstd.d = DR[m>>1];
if(dstd.l[1] & 0x1fffffff)) set_I();
if(FPSCR_RM == 1) dstd.l[1] &= 0xe0000000; /* round toward zero*/
dstf.f = dstd.d;
check_single_exception(FPUL, dstf.f);
}

**FCNVDS Special Cases**

<table>
<thead>
<tr>
<th>FRn</th>
<th>+NORM</th>
<th>–NORM</th>
<th>+0</th>
<th>–0</th>
<th>+INF</th>
<th>–INF</th>
<th>qNaN</th>
<th>sNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCNVDS(FRn FPUL)</td>
<td>FCNVDS</td>
<td>FCNVDS</td>
<td>+0</td>
<td>–0</td>
<td>+INF</td>
<td>–INF</td>
<td>qNaN</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

**Note:** The value of a denormalized number is treated as 0.
Possible Exceptions:

- Invalid operation
- Overflow
- Underflow
- Inexact
6.5.5 FCNVSD Floating-point CoNVert
Single to Double precision Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FCNVSD FPUL, DRn (double) FPUL → DRn</td>
<td>1111nnn010101101</td>
<td>2</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Description

When FPSCR.PR = 1, this instruction converts the single-precision floating-point number in FPUL to a double-precision floating-point number, and stores the result in DRn.

If FPSCR.PR = 0, the instruction is handled as an illegal instruction.

Operation

```c
void FCNVSD(int n, float *FPUL){
    pc += 2;
    clear_cause();
    case((FPSCR_PR){
        0:  undefined_operation();    /* reserved */
        1:  fcnvsd (n, *FPUL);  break;  /* FCNVSD */
    }
}
void fcnvsd(int n, float *FPUL)
{
    case(fpul_type(FPUL)){
        PZERO :         
        NZERO :         
        PINF :          
        NINF :          DR[n>>1] = *FPUL;  break;
        qNaN :          qnan(n);     break;
        sNaN :          invalid(n);  break;
    }
}  ```
int fpul_type(int *FPUL)
{
    int abs;
    abs = *FPUL & 0x7fffffff;
    if(abs < 0x00800000) {
        if((FPSCR_DN == 1) || (abs == 0x00000000)) {
            if(sign_of(src) == 0) return(PZERO);
        else return(NZERO);
        }
    else return(DENORM);

    else if(abs < 0x7f800000) return(NORM);
    else if(abs == 0x7f800000) {
        if(sign_of(src) == 0) return(PINF);
        else return(NINF);
    }
    else if(abs < 0x7fc00000) return(qNaN);
    else return(sNaN);
}

FCNVSD Special Cases

<table>
<thead>
<tr>
<th>FRn</th>
<th>+NORM</th>
<th>-NORM</th>
<th>+0</th>
<th>-0</th>
<th>+INF</th>
<th>-INF</th>
<th>qNaN</th>
<th>sNaN</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCNVSD(FPUL, FRn)</td>
<td>+NORM</td>
<td>-NORM</td>
<td>+0</td>
<td>-0</td>
<td>+INF</td>
<td>-INF</td>
<td>qNaN</td>
<td>sNaN</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

Possible Exceptions:
- Invalid operation
6.5.6 FDIV Floating-Point Division

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FDIV FRm,FRn</td>
<td>FRn/FRm → FRn</td>
<td>1111nnnnnnnnm011</td>
<td>10</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FDIV DRm,DRn</td>
<td>DRn/DRm → DRn</td>
<td>1111nnnn0mm011</td>
<td>23</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0: Arithmetically divides the single-precision floating-point number in FRn by the single-precision floating-point number in FRm, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically divides the double-precision floating-point number in DRn by the double-precision floating-point number in DRm, and stores the result in DRn.

When FPSCR.enable.O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

**Operation**

```c
void FDIV(int m,n)     /* FDIV FRm,FRn */
{
    pc += 2;
    clear_cause();
    if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN)) invalid(n);
    else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN)) qnan(n);
    else switch (data_type_of(m)){
        case NORM: switch (data_type_of(n)){
            case PINF:
            case NINF:    inf(n,sign_of(m)^sign_of(n));break;
            case PZERO:
            case NZERO:   zero(n,sign_of(m)^sign_of(n));break;
            default:      normal_fdiv(m,n);  break;
        }
    }
```
void normal_fdiv(int m, n)
{
union {
    float f;
    int l;
} dstf, tmpf;
union {
    double d;
    int l[2];
} dstd, tmpd;
union {
    int double x;
}
int l[4];

}  
tmpx;

if(FPSCR_PR == 0) {
    tmpf.f = FR[n]; /* save destination value */
    dstf.f /= FR[m]; /* round toward nearest or even */
    tmpd.d = dstf.f; /* convert single to double */
    tmpd *= FR[m];
    if(tmpf.f != tmpd.d) set_I();
    if((tmpf.f < tmpd.d) && (SPSCR_RM == 1))
        dstf.l -= 1; /* round toward zero */
    check_single_exception(&FR[n], dstf.f);
} else {
    tmpd.d = DR[n>>1]; /* save destination value */
    dstd.d /= DR[m>>1]; /* round toward nearest or even */
    tmpx.x = dstd.d; /* convert double to int double */
    tmpx *= DR[m>>1];
    if(tmpd.d != tmpx.x) set_I();
    if((tmpd.d < tmpx.x) && (SPSCR_RM == 1)) {
        dstd.l[1] -= 1; /* round toward zero */
        if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
    }
    check_double_exception(&DR[n>>1], dstd.d);
}
}
### FDIV Special Cases

<table>
<thead>
<tr>
<th>FRm,DRm</th>
<th>NORM</th>
<th>+0</th>
<th>–0</th>
<th>+INF</th>
<th>–INF</th>
<th>qNaN</th>
<th>sNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>DIV</td>
<td>0</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>DZ</td>
<td>Invalid</td>
<td>+INF</td>
<td>–INF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–0</td>
<td></td>
<td></td>
<td>–INF</td>
<td>+INF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>0</td>
<td>+0</td>
<td>–0</td>
<td></td>
<td></td>
<td>qNaN</td>
<td></td>
</tr>
<tr>
<td>–INF</td>
<td></td>
<td>–0</td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td>Invalid</td>
</tr>
<tr>
<td>qNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

### Possible Exceptions:
- Invalid operation
- Divide by zero
- Overflow
- Underflow
- Inexact
6.5.7  FLDI0  Floating-point Load Immediate 0.0  Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FLDI0</td>
<td>FRn 0x00000000 → FRn</td>
<td>l1l1nnnn10001101</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0, this instruction loads floating-point 0.0 (0x00000000) into FRn.

If FPSCR.PR = 1, the instruction is handled as an illegal instruction.

**Operation**

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

**Possible Exceptions:**

None
### 6.5.8 FLDI1 Floating-point Load

**Immediate 1.0 Floating-Point Instruction**

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract Code</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLDI1</td>
<td>FRn → FRn</td>
<td>0x3F800000 → FRn</td>
<td>1111nnnn10011101</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Description

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

If FPSCR.PR = 1, the instruction is handled as an illegal instruction.

#### Operation

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

#### Possible Exceptions:

None
6.5.9    FLDS    Floating-point
         Load to System register    Floating-Point Instruction

Transfer to System
Register

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLDS FRm,FPUL</td>
<td>FRm → FPUL</td>
<td>0111mmmm00011101</td>
<td>1</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Possible Exceptions:
None
6.5.10 FLOAT

Floating-point Instruction

Integer to Floating-Point Conversion

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FLOAT FPUL,FRn (float)FPUL → FRn</td>
<td>1111nnnn00101101</td>
<td>1</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FLOAT FPUL,DRn (double)FPUL → DRn</td>
<td>1111nnnn000101101</td>
<td>2</td>
<td></td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0: Taking the contents of FPUL as a 32-bit integer, converts this integer to a single-precision floating-point number and stores the result in FRn.

When FPSCR.PR = 1: Taking the contents of FPUL as a 32-bit integer, converts this integer to a double-precision floating-point number and stores the result in DRn.

When FPSCR.enable.I = 1, and FPSCR.PR = 0, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.
Operation

void FLOAT(int n, float *FPUL)
{
    union {
        double d;
        int l[2];
    } tmp;
    pc += 2;
    clear_cause();
    if(FPSCR.PR==0){
        FR[n] = *FPUL; /* convert from integer to float */
        tmp.d = *FPUL;
        if(tmp.l[1] & 0x1ffffff) inexact();
    } else {
        DR[n>>1] = *FPUL; /* convert from integer to double */
    }
}

Possible Exceptions:

Inexact: Not generated when FPSCR.PR = 1.
6.5.11 FMAC Floating-point Multiply and Accumulate Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FMAC FR0,FRm,FRn</td>
<td>FR0*FRm+FRn → FRn</td>
<td>1111nnnnnnnnnnnn1110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0, this instruction arithmetically multiplies the two single-precision floating-point numbers in FR0 and FRm, arithmetically adds the contents of FRn, and stores the result in FRn.

When FPSCR.enable.O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn is not updated. Appropriate processing should therefore be performed by software.

If FPSCR.PR = 1, the instruction is handled as an illegal instruction.

**Operation**

```c
void FMAC(int m, n)
{
    pc += 2;
    clear_cause();
    if(FPSCR_PR == 1) undefined_operation();
    else if((data_type_of(0) == sNaN) ||
             (data_type_of(m) == sNaN) ||
             (data_type_of(n) == sNaN)) invalid(n);
    else if((data_type_of(0) == qNaN) ||
             (data_type_of(m) == qNaN)) qnan(n);
    else if((data_type_of(0) == DENORM) ||
             (data_type_of(m) == DENORM)) set_E();
    else switch (data_type_of(0)){
        case NORM: switch (data_type_of(m)){
```
case PZERO:
  case NZERO: switch (data_type_of(n)){
    case qNaN:  qnan(n); break;
    case PZERO:
      case NZERO: zero(n, sign_of(0)^ sign_of(m)^sign_of(n));
      break;
    default:     break;
  }
  break;

case PINF:
  case NINF: switch (data_type_of(n)){
    case qNaN:    qnan(n); break;
    case PINF:
      case NINF: if(sign_of(0)^ sign_of(m)^sign_of(n)) invalid(n);
        else  inf(n, sign_of(0)^ sign_of(m)); break;
    default:      inf(n, sign_of(0)^ sign_of(m)); break;
  }
  break;

case NORM: switch (data_type_of(n)){
  case qNaN:   qnan(n);  break;
  case PINF:
    case NINF:   inf(n, sign_of(n)); break;
  case PZERO:
    case NZERO:
      case NORM: normal_fmac(m,n);  break;
      break;
  case PZERO:
    case NZERO: switch (data_type_of(m)){
      case PINF:\    invalid(n); break;
      case PZERO:
        case NZERO:
          case NORM: switch (data_type_of(n)){
            case qNaN: qnan(n);  break;
            case PZERO:
              case NZERO:
                case NZERO: zero(n, sign_of(0)^ sign_of(m)^sign_of(n));  break;
                default: break;
      case PZERO:
        case NZERO: zero(n, sign_of(0)^ sign_of(m)^sign_of(n));  break;
                default: break;
void normal_fmac(int m,n) {
union {
    int double x;
    int l[4];
} dstx,tmpx;
float dstf,srcf;
  if((data_type_of(n) == PZERO) || (data_type_of(n) == NZERO))
    srcf = 0.0; /* flush denormalized value */
  else    srcf = FR[n];
  tmpx.x = FR[0]; /* convert single to int double */
  tmpx.x *= FR[m]; /* exact product */
  dstx.x = tmpx.x + srcf;
  if(((dstx.x == srcf) && (tmpx.x != 0.0)) ||
    ((dstx.x == tmpx.x) && (srcf != 0.0))) {
    set_I();
    if(sign_of(0)^ sign_of(m)^ sign_of(n))  {
      dstx.l[3] -= 1; /* correct result */
      if(dstx.l[2] == 0xffffffff) dstx.l[1] -= 1;
      if(dstx.l[1] == 0xffffffff) dstx.l[0] -= 1;
    }
  }
}
else  dstx.l[3] |= 1;
}
if((dstx.l[1] & 0x01ffffff) || dstx.l[2] || dstx.l[3]) set_I();
if(FPSCR_RM == 1) {
    dstx.l[1] &= 0xfe000000; /* round toward zero */
    dstx.l[2] = 0x00000000;
    dstx.l[3] = 0x00000000;
}
dstf = dstx.x;
check_single_exception(&FR[n], dstf);
}
### FMAC Special Cases

<table>
<thead>
<tr>
<th>FRn</th>
<th>FR0</th>
<th>FRm</th>
<th>+Norm</th>
<th>–Norm</th>
<th>+0</th>
<th>–0</th>
<th>+INF</th>
<th>–INF</th>
<th>qNaN</th>
<th>sNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Norm</td>
<td>Norm</td>
<td>MAC</td>
<td>INF</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>INF</td>
<td>Invalid</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>Norm</td>
<td>MAC</td>
<td>+0</td>
<td>–0</td>
<td>+INF</td>
<td>–INF</td>
<td>qNaN</td>
<td>sNaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>INF</td>
<td>+0</td>
<td>Invalid</td>
<td>INF</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–0</td>
<td>+Norm</td>
<td>MAC</td>
<td>–0</td>
<td>+0</td>
<td>–INF</td>
<td>+INF</td>
<td>qNaN</td>
<td>sNaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>–0</td>
<td>INF</td>
<td>Invalid</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>+Norm</td>
<td></td>
<td>+INF</td>
<td>Invalid</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>INF</td>
<td>Invalid</td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–INF</td>
<td>+Norm</td>
<td></td>
<td>–INF</td>
<td>–INF</td>
<td></td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>INF</td>
<td>Invalid</td>
<td>INF</td>
<td></td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qNaN</td>
<td>0</td>
<td>INF</td>
<td>Invalid</td>
<td>qNaN</td>
<td></td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1sNaN</td>
<td>qNaN</td>
<td></td>
<td>INF</td>
<td>Invalid</td>
<td>qNaN</td>
<td></td>
<td>INF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** When DN = 1, the value of a denormalized number is treated as 0.
Possible Exceptions:
- Invalid operation
- Overflow
- Underflow
- Inexact
6.5.12 FMOV Floating-Point MOVe Floating-Point Instruction

<table>
<thead>
<tr>
<th>No.</th>
<th>SZ</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>FMOV FRm,FRn</td>
<td>FRm → FRn</td>
<td>1111nnnnnnnnmm1100</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>FMOV DRm,DRn</td>
<td>DRm → DRn</td>
<td>1111nnnnnnnnmm01000</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>3.</td>
<td>0</td>
<td>FMOV.S FRm,@Rn</td>
<td>FRm → (Rn)</td>
<td>1111nnnnnnnnmm1010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>FMOV.D DRm,@Rn</td>
<td>DRm → (Rn)</td>
<td>1111nnnnnnnnmm01000</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>5.</td>
<td>0</td>
<td>FMOV.S @Rm,FRn</td>
<td>(Rm) → FRn</td>
<td>1111nnnnnnnnmm1000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>6.</td>
<td>1</td>
<td>FMOV.D @Rm,DRn</td>
<td>(Rm) → DRn</td>
<td>1111nnnnnnnnmm1000</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>7.</td>
<td>0</td>
<td>FMOV.S @Rm+,FRn</td>
<td>(Rm) → FRn,Rm+=4</td>
<td>1111nnnnnnnnmm1000</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>8.</td>
<td>1</td>
<td>FMOV.D @Rm+,DRn</td>
<td>(Rm) → DRn,Rm+=8</td>
<td>1111nnnnnnnnmm1000</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>9.</td>
<td>0</td>
<td>FMOV.S FRm,@-Rn</td>
<td>Rn=4,FRm → (Rn)</td>
<td>1111nnnnnnnnmm1010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>10.</td>
<td>1</td>
<td>FMOV.D DRm,@-Rn</td>
<td>Rn=8,DRm → (Rn)</td>
<td>1111nnnnnnnnmm1010</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>11.</td>
<td>0</td>
<td>FMOV.S @(R0,Rm),FRn</td>
<td>(R0+Rm) → FRn</td>
<td>1111nnnnnnnnmm0110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>12.</td>
<td>1</td>
<td>FMOV.D @(R0,Rm),DRn</td>
<td>(R0+Rm) → DRn</td>
<td>1111nnnnnnnnmm0110</td>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>13.</td>
<td>0</td>
<td>FMOV.S FRm, @(R0,Rn)</td>
<td>FRm → (R0+Rn)</td>
<td>1111nnnnnnnnmm0111</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>14.</td>
<td>1</td>
<td>FMOV.D DRm, @(R0,Rn)</td>
<td>DRm → (R0+Rn)</td>
<td>1111nnnnnnnnmm0111</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

1. This instruction transfers FRm contents to FRn.
2. This instruction transfers DRm contents to DRn.
3. This instruction transfers FRm contents to memory at address indicated by Rn.
4. This instruction transfers DRm contents to memory at address indicated by Rn.
5. This instruction transfers contents of memory at address indicated by Rm to FRn.
6. This instruction transfers contents of memory at address indicated by Rm to DRn.
7. This instruction transfers contents of memory at address indicated by Rm to FRn, and adds 4 to Rm.
8. This instruction transfers contents of memory at address indicated by Rm to DRn, and adds 8 to Rm.
9. This instruction subtracts 4 from Rn, and transfers FRm contents to memory at address indicated by resulting Rn value.
10. This instruction subtracts 8 from Rn, and transfers DRm contents to memory at address indicated by resulting Rn value.
11. This instruction transfers contents of memory at address indicated by \((R0 + Rm)\) to \(FRn\).
12. This instruction transfers contents of memory at address indicated by \((R0 + Rm)\) to \(DRn\).
13. This instruction transfers \(FRm\) contents to memory at address indicated by \((R0 + Rn)\).
14. This instruction transfers \(DRm\) contents to memory at address indicated by \((R0 + Rn)\).

### Operation

```c
void FMOV(int m,n) /* FMOV FRm,FRn */
{
    FR[n] = FR[m];
    pc += 2;
}

void FMOV_DR(int m,n) /* FMOV DRm,DRn */
{
    DR[n>>1] = DR[m>>1];
    pc += 2;
}

void FMOV_STORE(int m,n) /* FMOV.S FRm,@Rn */
{
    store_int(FR[m],R[n]);
    pc += 2;
}

void FMOV_STORE_DR(int m,n) /* FMOV.D DRm,@Rn */
{
    store_quad(DR[m>>1],R[n]);
    pc += 2;
}

void FMOV_LOAD(int m,n) /* FMOV.S @Rm,FRn */
{
    load_int(R[m],FR[n]);
    pc += 2;
}

void FMOV_LOAD_DR(int m,n) /* FMOV.D @Rm,DRn */
{
    load_quad(R[m],DR[n>>1]);
    pc += 2;
}
```
Section 6  Instruction Descriptions

}  
void FMOV_RESTORE(int m,n)     /* FMOV.S @Rm+,FRn */  
{  
    load_int(R[m],FR[n]);
    R[m] += 4;
    pc += 2;
}  
void FMOV_RESTORE_DR(int m,n) /* FMOV.D @Rm+,DRn */  
{  
    load_quad(R[m],DR[n>>1]) ;
    R[m] += 8;
    pc += 2;
}  
void FMOV_SAVE(int m,n)        /* FMOV.S FRm,@–Rn */  
{  
    store_int(FR[m],R[n]-4);
    R[n] -= 4;
    pc += 2;
}  
void FMOV_SAVE_DR(int m,n)    /* FMOV.D DRm,@–Rn */  
{  
    store_quad(DR[m>>1],R[n]-8);
    R[n] -= 8;
    pc += 2;
}  
void FMOV_INDEX_LOAD(int m,n)  /* FMOV.S @(R0,Rm),FRn */  
{  
    load_int(R[0] + R[m],FR[n]);
    pc += 2;
}  
void FMOV_INDEX_LOAD_DR(int m,n) /*FMOV.D @(R0,Rm),DRn */  
{  
    load_quad(R[0] + R[m],DR[n>>1]);
    pc += 2;
}
void FMOV_INDEX_STORE(int m, n) /*FMOV.S FRm,@(R0,Rn)*/
{
    store_int(FR[m], R[0] + R[n]);
    pc += 2;
}

void FMOV_INDEX_STORE_DR(int m, n) /*FMOV.D DRm,@(R0,Rn)*/
{
    store_quad(DR[m>>1], R[0] + R[n]);
    pc += 2;
}

Possible Exceptions:
- Address error
6.5.13 FMUL Floating-point MULtiply Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FMUL FRm,FRn</td>
<td>FRn*FRm → FRn</td>
<td>111nnnnnnnn0010</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>FMUL DRm,DRn</td>
<td>DRn*DRm → DRn</td>
<td>111nnn00000010</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0: Arithmetically multiplies the two single-precision floating-point numbers in FRn and FRm, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically multiplies the two double-precision floating-point numbers in DRn and DRm, and stores the result in DRn.

When FPSCR.enable.O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

**Operation**

```c
void FMUL(int m,n)
{
    pc += 2;
    clear_cause();
    if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN)) invalid(n);
    else if((data_type_of(m) == qNaN) ||
        (data_type_of(n) == qNaN)) qnan(n);
    else switch (data_type_of(m)){
        case NORM: switch (data_type_of(n)){
            case PZERO: zero(n,sign_of(m)^sign_of(n)); break;
            case NZERO: zero(n,sign_of(m)^sign_of(n)); break;
            case PINF: inf(n,sign_of(m)^sign_of(n)); break;
            case NINF:      inf(n,sign_of(m)^sign_of(n)); break;
            default:        normal_fmul(m,n); break;
        }
    }
```


```c
break;

} break;
case PZERO:
case NZERO: switch (data_type_of(n)){
case PINF:
case NINF: invalid(n); break;
default: zero(n,sign_of(m)^sign_of(n));break;
} break;
case PINF :
case NINF : switch (data_type_of(n)){
case PZERO:  
case NZERO: invalid(n); break;
default: inf(n,sign_of(m)^sign_of(n));break
} break;
}
```

### FMUL Special Cases

<table>
<thead>
<tr>
<th>FRm,DRm</th>
<th>FRn,DRn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NORM</td>
</tr>
<tr>
<td>NORM</td>
<td>MUL</td>
</tr>
<tr>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>–0</td>
<td>–0</td>
</tr>
<tr>
<td>+INF</td>
<td>INF</td>
</tr>
<tr>
<td>–INF</td>
<td>INF</td>
</tr>
<tr>
<td>qNaN</td>
<td></td>
</tr>
<tr>
<td>sNaN</td>
<td></td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

### Possible Exceptions:
- Invalid operation
- Overflow
- Underflow
- Inexact
6.5.14 FNEG Floating-point NEGate value Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FNEG</td>
<td>FRn → FRn</td>
<td>1111nnnn01001101</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>FNEG</td>
<td>DRn → DRn</td>
<td>1111nnn001001101</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Description
This instruction inverts the most significant bit (sign bit) of the contents of floating-point register FRn/DRn, and stores the result in FRn/DRn.

The cause and flag fields in FPSCR are not updated.

Operation

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

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

Possible Exceptions:
None
6.5.15 FSCHG Sz-bit CHanGe Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FSCHG</td>
<td>FPSCR.SZ=¬FPSCR.SZ</td>
<td>1111001111111101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

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

If FPSCR.PR = 1, the instruction is handled as an illegal instruction.

**Operation**

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

**Possible Exceptions:**

None
### 6.5.16 FSQRT Floating-point SQuare RooT Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FSQRT</td>
<td>FRn → FRn</td>
<td>0F S Q R T F R n</td>
<td>1111nnnn01101101</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>FSQRT</td>
<td>DRn → DRn</td>
<td>1111nnnn01101101</td>
<td>22</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0: Finds the arithmetical square root of the single-precision floating-point number in FRn, and stores the result in FRn.

When FPSCR.PR = 1: Finds the arithmetical square root of the double-precision floating-point number in DRn, and stores the result in DRn.

When FPSCR.enable.I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

**Operation**

```c
void FSQRT(int n) {
    pc += 2;
    clear_cause();
    switch(data_type_of(n)){
        case NORM : if(sign_of(n) == 0) normal_fsqrt(n);
                     else invalid(n); break;
        case PZERO :
        case NZERO :
        case PINF  :
        case NINF  :
        case qNaN  :
        case sNaN  :
            invalid(n); break;
    }
}
```

---

*Rev. 3.00 Jul 08, 2005 page 312 of 484
REJ09B0051-0300*
void normal_fsqrt(int n)
{
union {
    float f;
    int l;
} dstf, tmpf;
union {
    double d;
    int l[2];
} dstd, tmpd;
union {
    int double x;
    int l[4];
} tmpx;
if(FPSCR_PR == 0) {
    tmpf.f = FR[n]; /* save destination value */
    dstf.f = sqrt(FR[n]); /* round toward nearest or even */
    tmpd.d = dstf.f; /* convert single to double */
    tmpd *= dstf.f;
    if(tmpf.f != tmpd.d) set_I();
    if((tmpf.f < tmpd.d) && (SPSCR_RM == 1))
        dstf.l -= 1; /* round toward zero */
    if(FPSCR & ENABLE_I) fpu_exception_trap();
    else FR[n] = dstf.f;
} else {
    tmpd.d = DR[n>>1]; /* save destination value */
    dstd.d = sqrt(DR[n>>1]); /* round toward nearest or even */
    tmpx.x = dstd.d; /* convert double to int double */
    tmpx *= dstd.d;
    if(tmpd.d != tmpx.x) set_I();
    if((tmpd.d < tmpx.x) && (SPSCR_RM == 1)) {
        dstd.l[1] -= 1; /* round toward zero */
        if(dstd.l[1] == 0xffffffff) dstd.l[0] -= 1;
    }
}
if(FPSCR & ENABLE_I) fpu_exception_trap();
else DR[n>>1] = dstd.d;
}

FSQRT Special Cases

<table>
<thead>
<tr>
<th>FRn</th>
<th>+NORM</th>
<th>-NORM</th>
<th>+0</th>
<th>-0</th>
<th>+INF</th>
<th>-INF</th>
<th>qNaN</th>
<th>sNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSQRT(FRn)</td>
<td>SQRT</td>
<td>Invalid</td>
<td>+0</td>
<td>-0</td>
<td>+INF</td>
<td>Invalid</td>
<td>qNaN</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

Possible Exceptions:
- Invalid operation
- Inexact
6.5.17 FSTS Floating-point STore  
System register  
Floating-Point Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSTS FPUL,FRn</td>
<td>FPUL \to FRn</td>
<td>1111nnnn00001101</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

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

Operation

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

Possible Exceptions:

None
6.5.18 FSUB Floating-point SUBtract Floating-Point Subtraction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FSUB</td>
<td>FRm,FRn FRn-FRm → FRn</td>
<td>1111nnnnmmmm0001</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FSUB</td>
<td>DRm,DRn DRn-DRm → DRn</td>
<td>1111nnn0mmmm00001</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

When FPSCR.PR = 0: Arithmetically subtracts the single-precision floating-point number in FRm from the single-precision floating-point number in FRn, and stores the result in FRn.

When FPSCR.PR = 1: Arithmetically subtracts the double-precision floating-point number in DRm from the double-precision floating-point number in DRn, and stores the result in DRn.

When FPSCR.enable.O/U/I is set, an FPU exception trap is generated regardless of whether or not an exception has occurred. When an exception occurs, correct exception information is reflected in FPSCR.cause and FPSCR.flag, and FRn or DRn is not updated. Appropriate processing should therefore be performed by software.

**Operation**

```c
void FSUB (int m,n)
{
    pc += 2;
    clear_cause();
    if((data_type_of(m) == sNaN) ||
        (data_type_of(n) == sNaN)) invalid(n);
    else if((data_type_of(m) == qNaN) ||
             (data_type_of(n) == qNaN)) qnan(n);
    else switch (data_type_of(m)){
        case NORM: switch (data_type_of(n)){
            case NORM: normal_faddsub(m,n,SUB); break;
            case PZERO: break;
            case NZERO: register_copy(m,n); FR[n] = -FR[n];break;
            default: break;
        }
    }
```
FSUB Special Cases

<table>
<thead>
<tr>
<th>FRm,DRm</th>
<th>FRn,DRn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NORM</td>
</tr>
<tr>
<td>NORM</td>
<td>SUB</td>
</tr>
<tr>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>–0</td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>–INF</td>
</tr>
<tr>
<td>–INF</td>
<td>+INF</td>
</tr>
<tr>
<td>qNaN</td>
<td></td>
</tr>
<tr>
<td>sNaN</td>
<td></td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

Possible Exceptions:

- Invalid operation
- Overflow
- Underflow
- Inexact
### 6.5.19 FTRC Floating-point TRuncate and Convert to integer Floating-Point Instruction

<table>
<thead>
<tr>
<th>PR</th>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FTRC FRm,FPUL</td>
<td>(long)FRm → FPUL</td>
<td>1111mmmm001111101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>FTRC DRm,FPUL</td>
<td>(long)DRm → FPUL</td>
<td>1111mmmm000111101</td>
<td>2</td>
<td>—</td>
</tr>
</tbody>
</table>

#### Description

When FPSCR.PR = 0: Converts the single-precision floating-point number in FRm to a 32-bit integer, and stores the result in FPUL.

When FPSCR.PR = 1: Converts the double-precision floating-point number in FRm to a 32-bit integer, and stores the result in FPUL.

The rounding mode is always truncation.

#### Operation

```c
#define N_INT_SINGLE_RANGE 0xcf000000 & 0x7fffffff /* -1.000000 * 2^31 */
#define P_INT_SINGLE_RANGE 0x4effffff /* 1.fffffe * 2^30 */
#define N_INT_DOUBLE_RANGE 0xc1e0000000200000 & 0x7fffffffffffffff
#define P_INT_DOUBLE_RANGE 0x41e0000000000000

void FTRC(int m, int *FPUL)
{
    pc += 2;
    clear_cause();
    if(FPSCR.PR==0){
        case(ftrc_single_type_of(m)){
            NORM:     *FPUL = FR[m];   break;
            PINF:     ftrc_invalid(0); break;
            NINF:     ftrc_invalid(1); break;
        }
    }
    else{        /* case FPSCR.PR=1 */
```
```c
int ftrc_signle_type_of(int m)
{
    if(sign_of(m) == 0)
    {
        if(FR_HEX[m] > 0x7f800000)    return(NINF);    /* NaN */
        else if(FR_HEX[m] > P_INT_SINGLE_RANGE)
            return(PINF);    /* out of range,+INF */
        else       return(NORM);    /* +0,+NORM          */
    } else {
        if((FR_HEX[m] & 0x7fffffff) > N_INT_SINGLE_RANGE)
            return(NINF);  /* out of range ,+INF,NaN*/
        else       return(NORM);   /* -0,-NORM              */
    }
}

int ftrc_double_type_of(int m)
{
    if(sign_of(m) == 0)
    {
        if((FR_HEX[m] > 0x7ff00000) ||
            (FR_HEX[m] >= 0x7ff00000) &&
            (FR_HEX[m+1] != 0x00000000))   return(NINF);     /* NaN */
        else if(DR_HEX[m>>1] >= P_INT_DOUBLE_RANGE)
            return(PINF);    /* out of range,+INF */
        else       return(NORM);    /* +0,+NORM          */
    } else {
        if((DR_HEX[m>>1] & 0x7fffffffffffffff) >= N_INT_DOUBLE_RANGE)
            return(NINF);    /* out of range ,+INF,NaN*/
        else       return(NORM);    /* -0,-NORM              */
    }
}
```

void ftrc_invalid(int sign, int *FPUL)
{
    set_V();
    if((FPSCR & ENABLE_V) == 0){
        if(sign == 0)     *FPUL = 0x7fffffff;
        else              *FPUL = 0x80000000;
    }
    else fpu_exception_trap();
}

FTRC Special Cases

<table>
<thead>
<tr>
<th>FRn,DRn</th>
<th>NORM</th>
<th>+0</th>
<th>–0</th>
<th>Positive Out of Range</th>
<th>Negative Out of Range</th>
<th>+INF</th>
<th>–INF</th>
<th>qNaN</th>
<th>sNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTRC (FRn,DRn)</td>
<td>TRC</td>
<td>0</td>
<td>0</td>
<td>Invalid</td>
<td>Invalid</td>
<td>+INF</td>
<td>–INF</td>
<td>qNaN</td>
<td>sNaN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+MAX</td>
<td>–MAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Invalid</td>
<td>Invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+MAX</td>
<td>–MAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Invalid</td>
<td>Invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+MAX</td>
<td>–MAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Invalid</td>
<td>Invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The value of a denormalized number is treated as 0.

Possible Exceptions:
- Invalid operation
6.5.20 LDS

Load to FPU System Register

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS</td>
<td>Rm,FPUL</td>
<td>0100mmmm01011010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS.L</td>
<td>@Rm+,FPUL</td>
<td>0100mmmm01010110</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS</td>
<td>Rm,FPSCR</td>
<td>0100mmmm01101010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>LDS.L</td>
<td>@Rm+,FPSCR</td>
<td>0100mmmm01100110</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

**Description**

This instruction loads the source operand into FPU system registers FPUL and FPSCR.

**Operation**

```c
#define FPSCR_MASK 0x003FFFFF

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

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

LDSFPSCR(int m) /* LDS Rm,FPSCR */
{
    FPSCR=R[m] & FPSCR_MASK;
    PC+=2;
}

LDSMFPSCR(int m) /* LDS.L @Rm+,FPSCR */
{
```
FPSCR=Read_Long(R[m]) & FPSCR_MASK;
R[m]+=4;
PC+=2;
}

Possible Exceptions:
- Address error
6.5.21 STS

STStore from FPU
System register
System Control Instruction

<table>
<thead>
<tr>
<th>Format</th>
<th>Abstract</th>
<th>Code</th>
<th>Cycle</th>
<th>T Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS FPUL,Rn</td>
<td>FPUL → Rn</td>
<td>0000nnnn01011010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>STS FPSCR,Rn</td>
<td>FPSCR → Rn</td>
<td>0000nnnn01101010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>STS.L FPUL,@-Rn</td>
<td>Rn-4 → Rn, FPUL → (Rn)</td>
<td>0100nnnn01010010</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>STS.L FPSCR,@-Rn</td>
<td>Rn-4 → Rn, FPSCR → (Rn)</td>
<td>0100nnnn01100010</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>

Description

This instruction stores FPU system register FPUL or FPSCR in the destination.

Operation

```c
STS(int n, int *FPUL)           /* STS FPUL,Rn */
{
    R[n]= *FPUL;
    PC+=2;
}

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

STS(int n)          /* STS FPSCR,Rn */
{
    R[n]=FPSCR&0x003FFFFF;
    PC+=2;
}

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

Possible Exceptions:

- Address error

Examples

- **STS**
  
  Example 1:
  
  ```assembly
  MOV.L #H'12ABCDEF, R12
  LDS R12, FPUL
  STS FPUL, R13
  ; After executing the STS instruction:
  ; R13 = 12ABCDEF
  ```

  Example 2:
  
  ```assembly
  STS FPSCR, R2
  ; After executing the STS instruction:
  ; The current content of FPSCR is stored in register R2
  ```

- **STS.L**
  
  Example 1:
  
  ```assembly
  MOV.L #H'0C700148, R7
  STS.L FPUL, @-R7
  ; Before executing the STS.L instruction:
  ; R7 = 0C700148
  ; After executing the STS.L instruction:
  ; R7 = 0C700144, and the content of FPUL is saved at memory location 0C700144.
  ```

  Example 2:
  
  ```assembly
  MOV.L #H'0C700154, R8
  STS.L FPSCR, @-R8
  ; After executing the STS.L instruction:
  ; The content of FPSCR is saved at memory location 0C700150.
  ```
Section 7 Register Banks

7.1 Overview

The SH-2A/SH2A-FPU has on-chip register banks to provide high-speed register save and retrieve performance during interrupt processing. The configuration of the register banks is shown in figure 7.1.
7.2 Register Banks and Bank Control Registers

7.2.1 Banked Data

The contents of general registers R0 to R14, the global register (GBR), the multiply and accumulate registers (MACH, MACL), the procedure register (PR), and the interrupt vector table address offsets (VTO) are banked.

7.2.2 Register Banks

The number of register banks is N, numbered from bank 0 to bank N – 1 (maximum 512 banks). Register banks are stacked in first in last out (FILO) sequence. Saves take place in order, beginning from bank 0, and retrieves take place in the reverse order, beginning from the last bank saved to. The number of banks, N, differs depending on the product. For details, refer to the Register Banks section of the hardware manual for the product in question.

7.2.3 Bank Control Registers

(1) Bank Control Register (IBCR) (16 bit, Initial value: H'0000)

This register is used to allow or prohibit the use of specific register banks, based on the interrupt priority level or the interrupt source. The register specifications and initial values differ depending on the product. For details, refer to the Interrupt Controller section of the hardware manual for the product in question.

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>E15</td>
<td>E14</td>
<td>E13</td>
<td>E12</td>
<td>E11</td>
<td>E10</td>
<td>E9</td>
<td>E8</td>
<td>E7</td>
<td>E6</td>
<td>E5</td>
<td>E4</td>
<td>E3</td>
<td>E2</td>
<td>E1</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Bits 15 to 1: E15 to E1
The setting of these bits is used to allow or prohibit use of register banks based on interrupt priority level (15 to 1).

<table>
<thead>
<tr>
<th>Bits 15 to 1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Register bank use is prohibited.</td>
</tr>
<tr>
<td>1</td>
<td>Register bank use is allowed.</td>
</tr>
</tbody>
</table>

Bit 0: Reserved Bit
This bit is always read as 0 and only a value of 0 should be written to it.
(2) Bank Number Register (IBNR) (16 bit, Initial value: H'0000)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE1</td>
<td>BE0</td>
<td>BOVE</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>BN3</td>
<td>BN2</td>
<td>BN1</td>
<td>BN0</td>
</tr>
</tbody>
</table>

The setting of the bank number register (IBNR) is used to allow or prohibit use of register banks and to allow or prohibit register bank overflow exceptions. In addition, bits BN3 to BN0 indicate the number of the next bank to be saved to. They are initialized to H'0000 by a power-on reset.

**Bits 15 and 14: BE1, BE0**

These bits specify whether register bank use is prohibited or allowed.

<table>
<thead>
<tr>
<th>BE1, BE0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Use of the bank is prohibited for all interrupts. The setting of IBCR is ignored. (Initial value)</td>
</tr>
<tr>
<td>01</td>
<td>Use of the bank is prohibited for all interrupts except NMI and UBC. The setting of IBCR is ignored.</td>
</tr>
<tr>
<td>10</td>
<td>Reserved. (Do not attempt to set this bit.)</td>
</tr>
<tr>
<td>11</td>
<td>Use of the bank is as specified by IBCR.</td>
</tr>
</tbody>
</table>

**Bit 13: BOVE**

This bit specify whether register bank overflow exceptions are prohibited or allowed.

<table>
<thead>
<tr>
<th>BOVE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Generation of register bank overflow exceptions is prohibited. (Initial value)</td>
</tr>
<tr>
<td>1</td>
<td>Generation of register bank overflow exceptions is allowed.</td>
</tr>
</tbody>
</table>

**Bits 12 to 4: Reserved Bits**

These bits are always read as 0 and only a value of 0 should be written to them.

**Bits 3 to 0: BN3 to BN0**

These bits indicate the number of the next bank to be saved to. When an interrupt that uses a register bank is received, it is saved to the bank specified by BN3 to BN0 and BN is incremented by 1. Execution of a register bank retrieve instruction causes BN to be decremented by 1, after which the data is retrieved from the register bank. These bits are read-only and cannot be modified.
7.3 Bank Save and Retrieve Operations

7.3.1 Save to Bank

Figure 7.2 illustrates the register bank save operations. The following operations are performed when an interrupt for which register bank use is allowed by IBCR is received by the CPU.

(a) Assume that the IBNR bank number value, BN, is i before the interrupt is generated.
(b) The contents of registers R0 to R14, GBR, MACH, MACL, PR, and the interrupt vector table address offset (VTO) are saved to the bank indicated by the BN, bank i.
(c) The BN value is incremented by 1.

Figure 7.2 Bank Save Operations

Figure 7.3 illustrates the register bank save timing. Saving to the bank takes place between the start of interrupt exception processing and the start of the fetch of the first instruction in the exception service routine.
Figure 7.3  Bank Save Timing

7.3.2  Retrieve from Bank

The retrieve from bank instruction, RESBANK, is used to retrieve data stored in a bank. After retrieving the data from the bank with the RESBANK instruction at the end of the interrupt service routine, use the RTE instruction to return from exception processing.

7.3.3  Save and Retrieve Operations after Saving to All Banks

If, after data has been saved to all of the register banks, an interrupt for which register bank use is allowed is received by the CPU, data is saved automatically to the stack instead of a register bank. This is possible by masking the register bank overflow exception using the interrupt controller. If a register bank overflow exception were generated it would not be possible to save to the stack. For details, refer to the Interrupt Controller section of the hardware manual for the product in question. The automatic save to and retrieve from stack operations are described below.

(1) Save to Stack

(a) When interrupt exception processing occurs, the status register (SR) and program counter (PC) are saved on the stack.
(b) The contents of the banked registers (R0 to R14, GBR, MACH, MACL, and PR) are saved to the stack. The order in which the contents of these registers are saved is MACL, MACH, GBR, PR, R14, R13, … R1, R0.
(c) The register bank overflow bit in SR is set to 1.
(d) The bank number (BN) bits in the bank number register (IBNR) remain set to the maximum value, N.
(2) Retrieve from Stack

If the retrieve from bank instruction, RESBANK, is executed when the register bank overflow bit in SR is set to 1, the following operations occur.

(a) The contents of the banked registers (R0 to R14, GBR, MACH, MACL, and PR) are retrieved from the stack. The order in which the contents of these registers are retrieved is R0, R1, … R13, R14, PR, GBR, MACH, MACL.

(b) The bank number (BN) bits in the bank number register (IBNR) remain set to the maximum value, N.

7.4 Register Bank Data Send Instructions

The LDBANK and STBANK instructions can be used to send user-defined register bank data to and from general register R0 for debugging purposes.

7.4.1 Description of Instructions

(1) LDBANK (Load Data from Register Bank to R0)

Format: LDBANK @Rm,R0
Operation: Sends 4 bytes of data from the register bank address indicated by Rm to R0.

(2) STBANK (Store Data from R0 to Register Bank)

Format: STBANK R0,@Rn
Operation: Sends the contents of R0 to the register bank address indicated by Rn.

7.4.2 Register Bank Addressing

Figure 7.4 illustrates the correlation between register bank send command address values (Rm in the case of LDBANK and Rn in the case of STBANK) and register bank entries. The bank number is specified by address bits 15 to 7 (BN), and the entry within the bank (R0 to R14, GBR, MACH, MACL, PR, VTO) is specified by address bits 6 to 2 (EN). Address bits 31 to 16 and 1 to 0 should all be cleared to 0. If the value of these bits is not all 0 operation cannot be guaranteed in cases where a nonexistent bank is specified by address bits 15 to 7 or a nonexistent entry is specified by address bits 6 to 2.
Register bank send instruction address (Rm, Rn)

Register banks (overall)

<table>
<thead>
<tr>
<th>Address</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>Bank 0</td>
</tr>
<tr>
<td>0000000001</td>
<td>Bank 1</td>
</tr>
<tr>
<td>0000000010</td>
<td>Bank 2</td>
</tr>
<tr>
<td>0000000011</td>
<td>Bank 3</td>
</tr>
<tr>
<td>1111111110</td>
<td>Bank N-2</td>
</tr>
<tr>
<td>1111111111</td>
<td>Bank N-1</td>
</tr>
</tbody>
</table>

Single register bank

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>R0</td>
</tr>
<tr>
<td>00001</td>
<td>R1</td>
</tr>
<tr>
<td>00010</td>
<td>R2</td>
</tr>
<tr>
<td>00011</td>
<td>R3</td>
</tr>
<tr>
<td>00100</td>
<td>R4</td>
</tr>
<tr>
<td>00101</td>
<td>R5</td>
</tr>
<tr>
<td>00110</td>
<td>R6</td>
</tr>
<tr>
<td>00111</td>
<td>R7</td>
</tr>
<tr>
<td>01000</td>
<td>R8</td>
</tr>
<tr>
<td>01001</td>
<td>R9</td>
</tr>
<tr>
<td>01010</td>
<td>R10</td>
</tr>
<tr>
<td>01011</td>
<td>R11</td>
</tr>
<tr>
<td>01100</td>
<td>R12</td>
</tr>
<tr>
<td>01101</td>
<td>R13</td>
</tr>
<tr>
<td>01110</td>
<td>R14</td>
</tr>
<tr>
<td>01111</td>
<td>MACH</td>
</tr>
<tr>
<td>10000</td>
<td>VTO</td>
</tr>
<tr>
<td>10001</td>
<td>PR</td>
</tr>
<tr>
<td>10010</td>
<td>GBR</td>
</tr>
<tr>
<td>10011</td>
<td>MACL</td>
</tr>
</tbody>
</table>

N = 512

Figure 7.4 Register Bank Addressing
7.5  **Register Bank Exceptions**

There are two types of register bank exception (register bank error): register bank overflow and register bank underflow.

7.5.1  **Register Bank Error Sources**

(1)  **Register Bank Overflow**

This exception occurs if, after data has been saved to all of the register banks, an interrupt for which register bank use is allowed is received by the CPU, and the register bank overflow exception is not masked by the interrupt controller. In this case the bank number (BN) bits in the bank number register (IBNR) remain set to the maximum value, N, and no data is saved to the register bank.

(2)  **Register Bank Underflow**

This exception occurs if the RESBANK instruction is executed when no data has been saved to the register banks. In this case the values of R0 to R14, GBR, MACH, MACL, and PR do not change. In addition, the bank number (BN) bits in the bank number register (IBNR) remain set to 0.

7.5.2  **Register Bank Error Exception Processing**

If a register bank error is generated, register bank error exception processing begins. When this happens the CPU performs the following operations.

1.  The contents of the status register (SR) are saved to the stack.
2.  The value of the program counter (PC) is saved to the stack. The PC value that is saved when a register bank overflow occurs is the starting address of the next instruction after the last executed instruction. The PC value that is saved when a register bank underflow occurs is the starting address of the relevant RESBANK instruction. To prevent multiple interrupts from occurring when a bank overflow occurs, the level of the interrupt that caused the overflow is written to the interrupt mask bits (I3 to I0) of the status register (SR).
3.  The exception service routine start address is extracted from the exception processing vector table corresponding to the register bank error, and the program is run beginning from that address.
7.6 SR Register Bank Overflow Bit (BO Bit)

The BO bit is modified when the contents of the SR register are retrieved by the RTE instruction. The BO bit is not modified when a RESBANK instruction is executed. The BO bit is set to 1 if exception generation by the interrupt controller is not enabled in cases where a bank overflow occurs during an interrupt. If exception generation by the interrupt controller is enabled for cases when a bank overflow occurs during an interrupt, the BO bit is not modified. The BO bit is modified by the LDC Rm.SR and LDC.L @Rmt.SR instructions.
Section 8 Pipeline Operation

This section describes the pipeline operation of the various instructions. This is information for calculating the number of CPU instruction execution states (number of system clock cycles).

The SH-2A/SH2A-FPU is a 2-ILP (2-Instruction-Level-Parallelism) super-scalar pipelining microprocessor. Instruction execution is pipelined, and two instructions can be executed in parallel. A Harvard architecture is used, and there is no contention between memory accesses and instruction fetches. As an instruction fetch unit is provided, the CPU core does not stop during an instruction fetch.

8.1 Basic Pipeline Configuration

The SH-2A/SH2A-FPU has the following pipelines (see figure 8.1).

- Integer pipelines 1 and 2: Process integer operations.
- Memory access pipeline: Processes memory accesses and the loading of data to the FPU.
- Multiplier pipeline: Processes multiply instructions and the storing of data from the FPU.
- Branch pipeline: Processes branch instructions.
- Shift pipeline: Processes shift instructions.
- FPU arithmetic operation pipeline: Processes FPU arithmetic operations.
- FPU division/square root extraction pipeline: Processes FPU division and square root extraction.

All instructions are first processed by an integer pipeline, and are also passed to another pipeline if necessary. These pipelines can all operate independently of each other. Therefore, if there is no contention, two instructions can always continue to be issued.

Instructions that perform memory access and instructions that load data from the CPU to the FPU use the memory access pipeline.

Multiply instructions and multiplication result register access instructions use the multiplier pipeline. In addition, inspections that store data from the FPU use the WB stage of the multiplier pipeline.

Branch instructions use the branch pipeline. Shift instructions use the shift pipeline.

Instructions that perform FPU internal register moves or data exchange from the FPU to memory or the CPU use the FPU load/store pipeline.
Instructions that perform FPU arithmetic operations use the FPU arithmetic operation pipeline.

Of the FPU arithmetic operations, FDIV and FSQRT use the FPU arithmetic operation pipeline and FPU division/square root extraction pipeline.

See section 8.9, Pipeline Operations for Each Instruction, for details.

The CPU pipeline stages are described in detail below.

- **IF**: Instruction fetch
  An instruction is fetched from memory in which the program is stored.

- **ID**: Instruction decoding
  The fetched instruction is decoded.

- **EX**: Instruction execution
  A data operation or address calculation is performed in accordance with the result of decoding.

- **MA**: Memory access
  A memory data access is performed.
  Generated by an instruction accompanying a memory access or an instruction that performs data exchange between the CPU and FPU.

- **mm**: Multiplier access
  A multiplier access is performed.
  Generated by an instruction accompanying a memory access or an instruction that loads data from the CPU to the FPU.

- **WB**: Write-back
  The result (data) accessed by a memory access or multiplier access is returned to the register.

The FPU pipeline stages are described in detail below. CPU and FPU pipelines share the first-stage instruction fetch (IF).

- **DF**: FPU decoding
  The fetched instruction is decoded.

- **E1**: FPU execution stage 1
  A floating-point operation is initialized.

- **E2**: FPU execution stage 2
  The floating-point operation is executed.
• SF: FPU store
  The floating-point operation is completed, and the result is written to an FPU register.

• ED: FPU division and square root calculation
  Used only for FDIV and FSQRT.

• EX: FPU load/store stage 1
  Floating-point load/store instruction data preparation is performed.

• NA: FPU load/store stage 2
  Floating-point load/store instruction data exchange is performed.

The length of all stages after ID and DF is the same. Only IF may be extended due to a wait for data, but as the instruction fetch unit and pipelines operate independently, pipelining can be continued in this case, also, for instructions that have already been fetched.

As shown in figure 8.2, instruction stages continue to flow together with instruction execution, forming a pipeline. The basic pipeline flow is shown in figure 8.1. The interval during which one stage is executed is called a slot, and is indicated by “↔”. Each instruction has at least a 3-stage structure.

The three stages IF, ID, and EX (integer pipeline) are present for each instruction. Thereafter, instruction processing is performed with the necessary pipelines operating simultaneously.
Figure 8.1 SH-2A/SH2A-FPU Pipelines

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction 5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction 6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
</tbody>
</table>

Figure 8.2 Basic Pipeline Configuration
8.2 Slots and Pipeline Flow

The interval during which one stage is executed is called a slot. The following rules apply to a slot.

(1) Each stage of an instruction (IF, ID, EX, MA, WB, mm, E1, E2, DF, ED, SF, NA) is always executed in one slot. Two or more stages are never executed in one slot (see figure 8.3). The ED stage operates without regard to a slot.

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
</tbody>
</table>

Note: ID and EX of instruction 1 are executed in one slot.

![Figure 8.3 Impossible Pipeline Flow (1)]

(2) The maximum number of different stages of different instructions set in one slot is two in the case of integer pipelines, and one in the case of other pipelines. Simultaneous pipeline execution never exceeds this number (see figure 8.4).

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Note: Three ID stages are executed in one slot.

![Figure 8.4 Impossible Pipeline Flow (2)]

(3) The number of states (number of system clock cycles) S required for execution of one slot is calculated using the following conditions.

(a) $S = \text{(maximum number of states among stages of each instruction contained in one slot)}$

That is to say, instructions that have other short stages are stalled by the longest stage.

(b) The number of execution states of each stage is as follows:

- **IF**: Number of memory access clocks for instruction fetch
  
  (As a fetch buffer is provided and instruction fetches are performed beforehand, pipeline stalling only occurs when a fetched instruction must be decoded immediately.)

- **ID**: Always 1 state

- **EX**: Always 1 state
- MA: Number of memory access clocks for data access
- WB: Always 1 state
- mm: Always 1 state
- DF: Always 1 state
- E1: Always 1 state
- E2: Always 1 state
- SF: Always 1 state
- ED: Always 1 state, but operates without regard to slots.
- NA: Always 1 state

For example, figure 8.5 shows the pipeline flow when IF (memory access for instruction fetch) of instructions 1 and 2 takes 2 cycles, MA (memory access for data access) of instruction 1 takes 3 cycles, and other stages take 1 cycle. “—” indicates stalling. For the sake of simplicity, this figure does not take super-scalar operation into consideration.

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>IF</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 2</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

Note: If IF requires more than one cycle, the slot is extended only if the instruction must be decoded immediately.

Figure 8.5 Slots Requiring a Number of Cycles
8.3 Instruction Execution and Parallel Execution Capability

The SH-2A/SH2A-FPU is a 2-ILP (2-Instruction-Level-Parallelism) super-scalar pipelining microprocessor. When two instructions are in the ID stage, two instructions can be executed simultaneously (see figure 8.6).

```
ADD R2,R3 IF  ID  EX
MOV.L @R0,R1 IF  ID  EX  MA  WB
ADD R4,R3 IF  ID  EX
FADD FR1,FR2 IF  DF  E1  E2  SF
```

Figure 8.6 Example of Parallel Execution

However, parallel execution is not possible in the following cases:

- When resource contention occurs (described in 8.3.1)
- When waiting for the result of a previously issued instruction (described in 8.3.2)
- When register contention or flag contention occurs (described in 8.3.3)
- When a multi-cycle instruction is executed as a preceding instruction (described in 8.3.4)
- When a 32-bit instruction is executed as a preceding instruction (described in 8.3.5)
- In the case of an instruction that uses FPSCR, an FPU instruction, or an FPU-related CPU instruction (described in 8.3.6)
- Delayed unconditional branch instruction at which a branch occurs, and delay slot (described in 8.3.7)

When IF stages are completed for two instructions without the occurrence of such contention, the SH-2A/SH2A-FPU can perform parallel execution of the two instructions.

The above cases are described in the following subsections. Terms used in the descriptions are as follows:

- Preceding instruction: Earlier instruction in the same slot
- Succeeding instruction: Later instruction in the same slot
- Previously issued instruction: Generic term for an instruction that has already been issued
8.3.1 Details of Resource Contention

As there is only one each of pipelines other than integer pipelines, if a preceding instruction and succeeding instruction attempt to use such a pipeline simultaneously, contention occurs and the succeeding instruction has to wait to be executed. Cases in which contention occurs are as follows.

(1) When the preceding instruction and succeeding instruction are both instructions accompanying a memory access (figure 8.8)

   Alternatively, in the case of a combination of a CPU → FPU data transfer instruction and memory write instruction (figure 8.8), or a combination with another FPU → CPU data transfer instruction.

   In these cases, memory access pipeline contention occurs.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Pipeline</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.L @R1+,R2</td>
<td>IF ID EX MA</td>
<td></td>
</tr>
<tr>
<td>MOV.L @R1+,R3</td>
<td>IF — ID EX MA</td>
<td></td>
</tr>
</tbody>
</table>

Note: There is a maximum of one memory access (MA) per slot.

Figure 8.8 Example of Memory Access Contention

LDS R0,FPUL IF ID EX : CPU pipeline
IF DF EX NA SF : FPU pipeline
MOV.L R1,@R3 IF — ID EX MA : CPU pipeline

Note: Contention between LDS instruction and memory write instruction

Figure 8.9 Example of Contention between LDS Instruction and Memory Write Instruction
Instructions that transfer data from the FPU to the CPU do not conflict with memory access instructions (figure 8.10). In addition, instructions that transfer data from the CPU to the FPU do not conflict with memory access instructions (figure 8.11).

<table>
<thead>
<tr>
<th>STS FPUL,R0</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
<th>CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>SF</td>
</tr>
<tr>
<td>MOV.L R1,@R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
</tbody>
</table>

Note: No contention between STS instruction and memory access instruction

**Figure 8.10  Example of Contention between STS and Memory Access**

<table>
<thead>
<tr>
<th>LDS R0,FPUL</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
</tr>
<tr>
<td>MOV.L @R1+,R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
</tbody>
</table>

Note: No contention between LDS instruction and memory read instruction

**Figure 8.11  Example of LDS Instruction and Memory Read Instruction**

(2) When the preceding instruction and succeeding instruction are both instructions that use the multiplier (figure 8.12).

With the multiplier, contention also occurs when a previously issued instruction is locked (figure 8.13).

In addition, instructions that read MACH or MACL, MULR instructions, and instructions that transfer the value of FPUL or FPSCR to the CPU cause contention because they share the read bus (figure 8.14).

| MULS.W R2,R1 | IF | ID | mm | mm |
| MULR R0,R3   | IF | —  | ID | mm | mm | mm | WB |

**Figure 8.12  Example of Multiplier Contention**

Multiplier locked

| LDS.L @R1+, MACH | IF | ID | EX | MA | WB |
| MULR R0,R3       | IF | —  | —  | ID | mm | mm | mm | WA |

**Figure 8.13  Example of Contention Due to Previously Issued Instruction**
Note: The two instructions using the multiplication result read bus conflict with each other.

**Figure 8.14  Example of Contention between Instructions Using Multiplication Result Read Bus**

(3) When the preceding instruction and succeeding instruction are both shift instructions or rotate instructions (figure 8.15)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAD R0,R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHAD R2,R3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.15  Example of Shift Instruction Contention**

(4) When the preceding instruction and succeeding instruction are both FPU arithmetic operation instructions (figure 8.16)

With regard to FPU arithmetic operation instructions, complex resource contention occurs with double-precision instructions or with FDIV or FSQRT instructions. See section 8.6, Contention Due to FPU, for details.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD FR0,FR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FADD FR2,FR3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.16  Example of FPU Arithmetic Operation Instruction Contention**

(5) When the preceding instruction and succeeding instruction are both FPU load/store instructions (figure 8.17)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>DF</th>
<th>EX</th>
<th>NA</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNEG FR0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMOV FR1,FR3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.17  Example of FPU Load/Store Instruction Contention**
8.3.2 Details of Contention Due to Wait for Result of Previously Issued Instruction

When the result of a previously issued instruction is used as a source, execution is performed after a wait equivalent to the latency of that instruction. Cases where this applies include the following:

- When waiting for the result of a memory access (see section 8.5, Effect of Memory Load Instruction on Pipeline, for details)
- When waiting for the result of an FPU operation (see section 8.6, Contention Due to FPU, for details)
- When waiting for the result of multiplication (see section 8.7, Contention Due to Multiplier, for details)

If the preceding instruction causes contention in these cases, the succeeding instruction must wait to be executed.

If the succeeding instruction causes contention, the preceding instruction is executed if there is no other contention.

8.3.3 Details of Register Contention and Flag Contention

In the following cases, register contention or flag contention occurs in the same slot.

(1) When the succeeding instruction uses the destination register or flag of the preceding instruction as a source register or flag (excluding a case where the preceding instruction is a zero-latency instruction) (figures 8.18 and 8.19)

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP/EQ R2,R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>BF</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>
```

**Figure 8.18 Example of Flag Contention between Preceding Destination and Succeeding Source**

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R3,R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>ADD R4,R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>
```

**Figure 8.19 Example of No Contention between Zero-Latency Instruction and Succeeding Instruction**
(2) When the succeeding instruction writes to the destination register or flag of the preceding instruction. (However, contention only occurs if an instruction other than a multiply instruction, divide instruction, LDBANK instruction, RESBANK instruction, MOVMU instruction, or MOVML instruction writes to registers and flags other than the FPU register and CS bit. No contention is detected with a multiply instruction, divide instruction, LDBANK instruction, or RESBANK instruction. In addition, contention is only detected for Rn with the MOVMU instruction and for R0 with the MOVML instruction. No contention occurs if either of these instructions write to other registers.) (Figures 8.20 to 8.25)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R3,R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>MOV R5,R4</td>
<td>IF</td>
<td>Id</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.20 Example of Contention Due to Instruction that Overwrites Destination of Preceding Instruction 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.L @R0,R1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
<tr>
<td>MOV.L @R2,R1</td>
<td>IF</td>
<td></td>
<td></td>
<td>ID</td>
</tr>
</tbody>
</table>

Figure 8.21 Example of Contention Due to Instruction that Overwrites Destination of Preceding Instruction 2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIPS.B R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>CLIPS.B R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.22 Example of No Contention in Case of CS Bit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R5,R6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>MULR R0,R6</td>
<td>IF</td>
<td>ID</td>
<td>mm</td>
<td>mm</td>
</tr>
</tbody>
</table>

Figure 8.23 Example of MULR No Contention

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R5,R6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVMUL.R15+,R13</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>MA</td>
</tr>
</tbody>
</table>

Figure 8.24 Example of MOVMUL.L No Contention
8.3.4 Details of Contention Due to Multi-Cycle Instruction

An instruction that does not have one execution state is called a “multi-cycle instruction.” The following rules apply to such instructions.

1. When a multi-cycle instruction is executed as a preceding instruction, it cannot be executed in parallel with the succeeding instruction.
2. During execution of a multi-cycle instruction, if the slot is not the last slot, the next instruction cannot be newly executed. “During execution” here refers to a slot not exceeding the number of execution state cycles counting from the instruction ID stage.
3. At the end of the execution states of a multi-cycle instruction (in the last slot: equivalent to the execution state cycle), parallel execution with the next instruction is possible. Parallel execution can be performed even if the next instruction is a 32-bit instruction.
4. A multi-cycle instruction can be executed in parallel with a preceding instruction that is a single-cycle instruction (an instruction with one execution state).

A relevant example is shown in figure 8.26.

(5) If a multicycle 32-bit instruction such as BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, or BXOR is followed on the next line by the instruction BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, or BXOR, the instruction on the second line is executed in parallel (figure 8.27).
8.3.5 Details of Contention Due to 32-Bit Instruction

The following rules apply to execution of 32-bit instructions.

(1) Parallel execution is not possible when the preceding instruction is a 32-bit instruction (figure 8.29).

(2) When the succeeding instruction is a 32-bit instruction, the preceding instruction can be executed but the succeeding instruction cannot (figure 8.29).

(3) The last slot of a multi-cycle instruction and a 32-bit instruction can be executed in parallel (figure 8.26).

(4) Only in cases where the preceding instruction in the last slot is a multicycle 32-bit instruction such as BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, or BXOR, and the instruction on the next line is BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, or BXOR, does parallel execution take place. Parallel execution does not occur in combinations with any other instructions (figures 8.27 and 8.28).

(5) A 32-bit instruction cannot be executed unless IF has been completed for the upper 16 bits and the lower 16 bits (figure 8.30).

Relevant examples are shown in figures 8.26 and 8.27.
8.3.6 Details of Contention Due to Instruction that Uses FPSCR

If an instruction uses FPSCR, parallel execution is not possible with any other instruction if this instruction precedes it. If this instruction follows, parallel execution is not possible with FPU instructions or FPU-related CPU instructions (figure 8.31).
8.3.7 Details of Contention Due to Branch Instruction

The following rules apply to contention due to a branch instruction.

(1) Parallel execution is possible when the branch instruction does not branch.
(2) When a branch instruction is supplied as a succeeding instruction, parallel execution with the preceding instruction is possible regardless of the branching situation.
(3) When a branch instruction is supplied as a preceding instruction, parallel execution with the succeeding instruction is not possible if a branch occurs. Parallel execution is not possible even if IF has already been completed for the delay slot (figure 8.32).
(4) For the delay slot, ID is performed in the next slot in which there is a branch instruction EX stage.
(5) Execution of a delayed branch instruction is delayed if a fetch has not been performed for the delay slot.

A relevant example is shown in figure 8.28.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R3,R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>JMP @R2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Delay slot</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
</tr>
<tr>
<td>Branch destination instruction</td>
<td>IF</td>
<td>ID</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.32 Example of Contention between Branch Instructions
8.4 Number of Instruction Execution States

The number of execution states of an instruction is counted in the EX stage execution interval. The number of states from the start of instruction 1 EX stage execution until the start of the EX stage of following instruction 2 constitutes the execution time of instruction 1.

For example, in the case of the pipeline flow shown in figure 8.33, the EX stage interval of instruction 1 and instruction 2 consists of 4 stages, and therefore the instruction 1 execution time is 4 states. Also, the EX stage interval of instruction 2 and instruction 3 consists of 1 states, and therefore the instruction 2 execution time is 1 state.

If the program ends at instruction 3, take instruction 4 as the next instruction after instruction 3 in virtual terms, and calculate the execution time of instruction 3 from the EX stages of instruction 3 and instruction 4 in MOV Rm,Rn. (In the example in figure 8.33, the execution time of instruction 3 is 1 state.)

The execution time from instruction 1 through instruction 3 in figure 8.33 is a total of $4 + 1 + 1 = 6$ states.

For the sake of simplicity, this figure does not take super-scalar operation into consideration.

---

Figure 8.33   Example of How to Count Number of Instruction Execution States
### 8.5 Effect of Memory Load Instruction on Pipeline

With an instruction that performs a load from memory, return of data to the destination register is performed in the WB stage at the end of the pipeline. Looking at such a load instruction (designated “load instruction 1” here) and the instruction immediately following it (designated “instruction 2”), the EX stage of instruction 2 comes before the WB stage of load instruction 1.

If, in this case, the destination register of load instruction 1 is used by instruction 2, since the contents of that register have not yet been prepared, execution of the ID stage is delayed for a period equivalent to the latency of instruction 1. The same also applies if the destination register of load instruction 1 is the same as the destination, rather than the source, of instruction 2.

Similarly, execution of the ID stage is stalled for an additional slot if the destination of load instruction 1 is the status register (SR) and a flag in SR is fetched and used by instruction 2 (such as ADDC, for example).

When this kind of register contention occurs, the slot in which the destination register can be used is the cycle after completion of the MA stage of instruction 1. This is illustrated in figure 8.34.

Therefore, if program is written in which an instruction that uses the result of a load instruction is placed immediately after that load instruction, execution speed will decrease. Generally, the latency of a load instruction is 2, and therefore speed will not decrease if an instruction that uses the result of a load is placed 3 or 4 instructions after the load instruction. If a memory access instruction is executed as a preceding instruction, the applicable number of instructions is 4 or more, and if executed as a succeeding instruction, 3 or more.

| Load instruction 1 (MOV.W @R0,R1) | IF | ID | EX | MA | WB |
| Instruction 2 (ADD R1,R3)        | IF | —  | —  | ID | EX |
|                                   | IF | —  | —  | ID | EX |
|                                   | IF | —  | —  | ID | EX |

**Figure 8.34 Effect of Memory Load Instruction on Pipeline**
8.6 Contention Due to FPU

When a register (FR0 to FR15, or FPUL) that stores the result of a floating-point arithmetic operation instruction, FMOV instruction, or floating-point load instruction is read (used as a source register) by a following floating-point arithmetic operation instruction or FMOV FRm,FRn instruction, the next instruction is issued after completion of the operation. As a result, that instruction is kept waiting for a period equivalent to the latency cycle of the preceding operation instruction (figure 8.35). A zero-latency instruction can be executed in parallel with the succeeding instruction even if the succeeding instruction uses the result register as its source (figure 8.36).

| Floating-point arithmetic operation instruction (single-precision) | IF | DF | E1 | E2 | SF |
| (FADD FR1,FR2) (latency 3) |    |    |    |    |    |
| Next floating-point instruction (single-precision) | IF | —  | —  | DF | EX | NA | SF |
| (FMOV FR2,FR3) |    |    |    |    |    |    |

Figure 8.35 Example of Use of FPU Operation Result by Succeeding Instruction

| Floating-point instruction (single-precision) | IF | DF | EX | NA | SF |
| (FMOV FR0,FR2) (latency 0) |    |    |    |    |    |
| Next floating-point arithmetic operation instruction (single-precision) | IF | DF | E1 | E2 | SF |
| (FADD FR2,FR3) |    |    |    |    |    |

Figure 8.36 Example of Use of Result of Zero-Latency Instruction as Source

When a register (FR0 to FR15) that stores the result of a floating-point arithmetic operation instruction is read (used as a source register) by a following FMOV or STS.L instruction, and the value is output to memory, latency is shortened by 1 cycle (figure 8.37).

| Floating-point arithmetic operation instruction (single-precision) | IF | DF | E1 | E2 | SF |
| (FADD FR0,FR2) |    |    |    |    |    |
| Next floating-point instruction (single-precision) | IF | —  | DF | EX | NA |
| (FMOV FR2,@R3) |    |    |    |    |    |

Figure 8.37 Example of Writing Result to Memory Immediately Following FPU Operation
When a register (FPUL) that stores the result of a floating-point arithmetic operation instruction is read (used as a source register) by a following STS instruction, and the value is output to the CPU, latency is shortened by 2 cycles (figure 8.38).

<table>
<thead>
<tr>
<th>Floating-point arithmetic operation instruction (single-precision)</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FTRC FR0,FPUL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next floating-point instruction (single-precision)</th>
<th>IF</th>
<th>DF</th>
<th>EX</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(STS FPUL,R3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.38   Example of Transferring Result to CPU Immediately Following FPU Operation

The time required for the result of an FCMP instruction to be reflected in the T bit is 2 cycles in the case of single-precision, and 3 cycles in the case of double-precision. As a result, if that instruction (the following instruction) references the T bit, execution is delayed by the above slot interval (figure 8.39).

<table>
<thead>
<tr>
<th>Instruction 1 (single-precision)</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FCMP FR0,FR1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction 2 (instruction that references T bit)</th>
<th>IF</th>
<th>—</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.39   Example of Referencing T Bit Immediately After FCMP Instruction

When the FPSCR value is changed using an LDS or LDS.L instruction, execution of the next instruction by a 3-slot interval (figure 8.40).

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>IF</th>
<th>DF</th>
<th>EX</th>
<th>NA</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(LDS R2,FPSCR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction 2</th>
<th>IF</th>
<th>—</th>
<th>—</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FADD FR4,FR5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.40   Example of Performing FPU Operation Immediately After FPSCR Load
When the FPSCR value is read using an STS or STS.L instruction, FPSCR is read after completion of the previously issued operation. As a result, execution is delayed by an interval of [latency of preceding operation + 1 slot] (figure 8.41).

<table>
<thead>
<tr>
<th>Instruction 1 (single-precision)</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FADD FR6,FR9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction 2</th>
<th>IF</th>
<th>—</th>
<th>—</th>
<th>—</th>
<th>DF</th>
<th>EX</th>
<th>NA</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(STS FPSCR,R3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.41** Example of Reading FPSCR

Double-precision floating-point arithmetic operation instructions (FADD, FSUB, FMUL) require 6 cycles for the E1 stage. Another floating-point arithmetic operation instruction will not enter the E1 stage during this interval. If another floating-point arithmetic operation instruction appears before a double-precision floating-point arithmetic operation instruction finishes the E1 stage, that floating-point arithmetic operation instruction has its execution delayed by a predetermined slot interval, and enters the E1 stage after the double-precision floating-point arithmetic operation instruction has finished the E1 stage. A floating-point load/store instruction arriving during this interval can be executed (figure 8.42).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E1</th>
<th>E1</th>
<th>E1</th>
<th>E1</th>
<th>E2</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD DR4,DR6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FABS DR0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STS FPUL,R0</td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL DR2,DR0</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>DF</td>
<td>E1</td>
<td>E2</td>
<td>SF</td>
</tr>
</tbody>
</table>

**Figure 8.42** Example of Double-Precision FPU Operation and Next FPU Instruction

With an FDIV or FSQRT instruction, after the E1 stage is used in initialization, operation is performed by an independent computer (ED stage), after which the operation result is written back. A floating-point arithmetic operation instruction following either of these instructions operates as described below. See section 8.9, Pipeline Operations for Each Instruction, for the kind of pipeline used with each instruction.

1. During E1 stage use in initialization, another floating-point arithmetic operation instruction will not enter the E1 stage. Other instructions enter the E1 stage after FDIV or FSQRT initialization ends.
2. After an FDIV or FSQRT instruction has progressed to the ED stage, an FPU instruction is executed without delay unless it uses the FDIV or FSQRT instruction result register (figure 8.40).
(3) At the end of an FDIV or FSQRT instruction, operation write-back occurs. The E1 stage is used again here, and therefore if an instruction requests E1 stage operation from just this point onward, the subsequent instruction is kept waiting until the FDIV or FSQRT instruction finishes using the E1 stage (figure 8.44).

(4) An FDIV or FSQRT instruction immediately following an FDIV or FSQRT instruction cannot enter the ED stage while the preceding FDIV or FSQRT instruction is using the ED stage.

| Instruction 1          | IF  DF  E1  ED  ED  ED  ED  ED  ED  ED  E1  E2  SF |
|------------------------|--------|---------------------------------|
| (single-precision)     |        | (FDIV  FR6,FR7)                 |
| Instruction 2          | IF  DF  E1  E2  SF               |
| (single-precision)     |        | (FADD  FR8,FR10)                |

**Figure 8.43  Example 1 of E1 Stage Contention Due to FDIV**

| Instruction 1          | IF  DF  E1  ED  ED  ED  ED  ED  ED  ED  E1  E2  SF |
|------------------------|--------|---------------------------------|
| (single-precision)     |        | (FDIV  FR6,FR7)                 |
| Other instruction      |        |                                 |
| Instruction 2          | IF  DF  E1  E2  SF               |
| (single-precision)     |        | (FADD  FR8,FR10)                |
| Instruction 3          | IF  DF  E1  E2  SF               |
| (single-precision)     |        | (FADD  FR9,FR11)                |

**Figure 8.44  Example 2 of E1 Stage Contention Due to FDIV**

If a write was performed by a previous instruction on a register used as a source register by a double-precision arithmetic operation instruction, and the latency of the previous instruction is 2 cycles or less, the latency of those instructions will be 2 (figure 8.45).
Floating-point load/store instruction (double-precision)  
(FMOV DR0,DR2)  
(latency 1 → latency 2)

Next floating-point arithmetic operation instruction (double-precision)  
(FADD DR2,DR4)

Figure 8.45  Example of 1-Latency Instruction Immediately Preceding Double-Precision Arithmetic Operation

If the destination register of a double-precision arithmetic operation instruction is used as a source register by the following instruction, if “n” of FRn is an odd number, latency will be reduced by 1 cycle (figure 8.46). However, latency will not be reduced if “n” of FRn is an even number (figure 8.47).

Floating-point arithmetic operation instruction (double-precision)  
(FADD DR0,DR2)  
(latency 8 → latency 7)

Next floating-point load/store instruction (single-precision)  
(FMOV FR3,FR5)

Figure 8.46  Example of Latency Reduction with Double-Precision Arithmetic Operation Instruction
Floating-point arithmetic operation instruction (double-precision) (FADD DR0,DR2) (remains at latency 8)

Next floating-point load/store instruction (single-precision) (FMOV FR2,FR4)

**Figure 8.47 Example of No Latency Reduction with Double-Precision Arithmetic Operation Instruction**

When a register (FR0 to FR15, or FPUL) that stores the result of a floating-point arithmetic operation instruction is written to (used as a destination register) by a following floating-point arithmetic operation instruction or floating-point load/store instruction, the next instruction is kept waiting before being executed. The number of cycles by which execution is delayed is \([\text{latency} - 1]\) cycles if the preceding operation was FDIV or FSQRT, and \([\text{latency} - 2]\) cycles otherwise (figures 8.48 and 8.49).

Floating-point arithmetic operation instruction (single-precision) (FDIV FR1,FR2) (latency 12 → latency 11)

Next floating-point load/store instruction (single-precision) (FMOV FR3,FR2)

**Figure 8.48 Example of Contention Due to Overwriting (FDIV, FSQRT)**
Floating-point arithmetic operation instruction (single-precision)
(FADD FR1,FR2) (latency 3 → latency 1)

Next floating-point instruction (single-precision)
(FMOV FR2,FR2)

---

Floating-point arithmetic operation instruction (double-precision)
(FADD DR0,DR2) (latency 0 → latency 2)

Next floating-point load/store instruction (single-precision)
(FMOV FR4,FR1)

Figure 8.49 Example of Contention Due to Overwriting (Except FDIV, FSQRT)

If a write is performed by the following instruction on the register used as a source register by a double-precision FADD, FSUB, or FMUL, the following will be kept waiting for 2 cycles (figure 8.50).

---

Figure 8.50 Example of Write to Double-Precision Instruction Source Immediately after Double-Precision Operation
8.7 Contention Due to Multiplier

Multiply instructions, multiply-and-accumulate instructions, and instructions that manipulate the registers for these instructions (MACH, MACL) use the multiplier. In addition, the STS FPUL,Rn, and STS FPSCR,Rn instructions use the multiplication result read bus. Details of pipelining and contention are given below, with instructions divided into the categories shown. The numbers immediately following the instructions, in the form (A/B/C), indicate (number of execution slots/latency/number of lock slots).

- Multiply-and-accumulate instructions
  - MAC.L (4/6/5) IF ID EX MA MA mm mm mm
  - MAC.W (3/5/4) IF ID EX MA MA mm mm

- Multiply instructions (I)
  - DMUL.S, DMUL.U, MUL.L (2/3/2) IF ID mm mm mm
  - MULS.W, MULU.W (1/2/1) IF ID mm mm

- Multiply instructions (II) (register return)
  - MULR (2/4/2) IF ID mm mm mm WB

- Register write instructions (I)
  - CLRMAC, LDS (1/2/1) IF ID mm mm

- Register write instructions (II)
  - LDS.L (1/3/2) IF ID EX MA WB

- Register read instructions (including STS FPUL,Rn and STS FPSCR,Rn)
  - STS (1/2/0) IF ID EX WB
  - STS.L (1/2/0) IF ID EX MA

Facts about Contention

Contention arises with multi-cycle instructions in the same way as with general instructions (figure 8.51). See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction, for details.

| MAC.L @R1+,@R2+ | IF | ID | EX | MA | MA | mm | mm | mm |
| MAC.L @R3+,@R4+ | IF | —  | —  | —  | ID | EX | MA | MA | mm | mm | mm |

Note: MAC.L is an instruction with an execution rate of 4.

Figure 8.51 Example of Multi-Cycle Instructions Using Multiplier

The following rules apply to instructions that use the multiplier.
(1) Execution of a instruction that uses a multiplication result as its source is delayed by an interval equivalent to the latency of that instruction (figure 8.52). If the following instruction is one that reads MACH or MACL, execution is delayed by [latency – 1] cycled (figure 8.53). If the following instruction is a multiply-and-accumulate instruction, execution is not delayed (figure 8.54).

```
MULR R0,R4    IF  ID  mm  mm  mm  WB
ADD  R4,R5    IF  —  —  —  —  ID  EX  WB
```

**Figure 8.52  Example of Referencing Result Register Immediately after Multiplication (1)**

```
MUL.L R2,R3    IF  ID  mm  mm  mm
STS  MACH,R4    IF  —  —  ID  EX  WB
```

**Figure 8.53  Example of Referencing Result Register Immediately after Multiplication (2)**

```
MAC.W @R1+,@R2+    IF  ID  EX  MA  MA  mm  mm
MAC.W @R3+,@R4+    IF  —  —  ID  EX  MA  MA  mm  mm
```

**Figure 8.54  Example of Referencing Result Register Immediately after Multiplication (3)**

(2) In the case of an instruction after an instruction that uses the multiplier, if the preceding instruction locked the multiplier, execution is delayed until the multiplier is unlocked (figure 8.55).

```
MULR1 lock interval  ←←←←→→→→
MULR1 R0,R1    IF  ID  mm  mm  mm  WB
MULR2 R0,R2    IF  —  —  ID  mm  mm  mm  WB
```

**Figure 8.55  Example of Multiplier Lock Contention**

However, if the following instruction is a multiply-and-accumulate instruction, it is executed after waiting for the same kind of state interval as with an ordinary multi-cycle instruction, rather than after waiting for the multiplier to be unlocked (figure 8.56).
If the following instruction is an instruction in category “Register write instructions (II),” it is executed when there is one slot remaining in the lock interval (figure 8.57).

STS and STS.L instructions do not lock the multiplier. Therefore, parallel execution is possible for an STS instruction and MUL.L instruction, etc.
(3) MULR instructions, STS instructions affecting MACH, MACL, FPUL, or FPSCR, and STS.L instructions affecting MACH or MACL share a result register read bus, causing resource contention (MA and WB stages). Therefore, parallel execution is not possible for STS and STS.L instructions (figure 8.59).

If an STS or STS.L is located immediately after a MULR instruction, WB stage contention occurs in the same way, and execution of the STS or STS.L instruction is delayed by 3 cycles (figure 8.60).
8.8 Programming Strategy

The following programming points should be noted in order to improve instruction execution speed.

1. A branch destination address should be at a longword boundary in memory. This enables parallel execution to be performed efficiently immediately after a branch.

2. The first 3 instructions immediately after an instruction that performs a load from memory should not include an instruction that uses the same register as the load instruction destination register. If possible, an instruction that uses the destination register should be no earlier than the fourth instruction after the load instruction.

3. The first 3 instructions immediately after a 32-bit multiply instruction should not include an instruction that uses the same register as the result register.

4. Instructions immediately following a floating-point arithmetic operation instruction, and having a latency between 1 and twice the latency of the floating-point arithmetic operation instruction, should not use the destination register of the floating-point arithmetic operation instruction.

8.9 Pipeline Operations for Each Instruction

Pipeline operations for each instruction are described below. In conjunction with the previously described rules and possibility of parallel execution, this information allows the program pipeline flow and number of instruction execution states to be calculated.

“Instruction A” in the following pipeline diagrams denotes the instruction being described.

The “Instruction Issuance” description indicates in particular how the instruction should be treated when taking resource contention into consideration.

The “Parallel Execution Capability” description indicates in particular how the instruction should be treated when taking parallel execution capability into consideration. Cases are described here in which there is no register contention.

The number of stages and number of execution states of an instruction are indicated using the format below. These tables show the number of states when the instruction is executed without register dependency.
### Format of Number of Instruction Stages and Execution States

<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type according to function</td>
<td>Instructions are categorized according to differences of operation.</td>
<td>Number of instruction stages</td>
<td>Number of execution states when there is no contention</td>
<td>Number of execution states until execution result is confirmed</td>
<td>Resource contention that occurs</td>
<td>Applicable instructions, indicated by mnemonic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transfer instructions</td>
<td>Register-register transfer instructions</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>MOV #imm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>MOV Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>MOVA @(disp,PC),R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV T Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV RT Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOTT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SWAP.B Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SWAP.W Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XTRCT Rm,Rn</td>
</tr>
</tbody>
</table>

• These are 32-bit instructions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV120</td>
<td>#imm,Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV120S #imm20,Rn</td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Data transfer instructions</td>
<td>Memory load instructions</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td></td>
<td>• These instructions use the memory access pipeline. MOV.W @(disp,PC),Rn MOV.L @(disp,PC),Rn MOV.B @Rm,Rn MOV.W @Rm,Rn MOV.L @Rm,Rn MOV.B @Rm+Rn,Rn MOV.W @Rm+Rn,Rn MOV.L @Rm+Rn,Rn MOV.B @Rm,R0 MOV.W @Rm,R0 MOV.L @Rm,R0 MOV.B @(disp,Rm),R0 MOV.W @(disp,Rm),R0 MOV.L @(disp,Rm),R0 MOV.B @(R0,Rm),Rn MOV.W @(R0,Rm),Rn MOV.L @(R0,Rm),Rn MOV.B @(disp,GBR),R0 MOV.W @(disp,GBR),R0 MOV.L @(disp,GBR),R0 MOV.B @(R15,Rm),Rn MOV.W @(R15,Rm),Rn MOV.L @(R15,Rm),Rn MOV.B @(disp12,Rm),Rn MOV.W @(disp12,Rm),Rn MOV.L @(disp12,Rm),Rn MOVU.B @(disp12,Rm),Rn MOVU.W @(disp12,Rm),Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 to 20</td>
<td>1 to 16</td>
<td>2 to 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Memory store</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>MOV.B Rm, @Rn</td>
</tr>
<tr>
<td>instructions</td>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W Rm, @Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L Rm, @Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.B Rm, @-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W Rm, @-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L Rm, @-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.B R0, @Rn+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W R0, @Rn+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L R0, @Rn+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.B R0, (disp, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W R0, (disp, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L R0, (disp, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.B Rm, (R0, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W Rm, (R0, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L Rm, (R0, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.B R0, (disp, GBR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W R0, (disp, GBR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L R0, (disp, GBR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 to 19</td>
<td>1 to 16</td>
<td>1 to 16</td>
<td></td>
<td>MOVML.L Rm, @-R15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOVMLU.L Rm, @-R15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>MOV.B Rm, (disp12, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.W Rm, (disp12, Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOV.L Rm, (disp12, Rn)</td>
</tr>
<tr>
<td>PREF instruction</td>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PREF @Rm</td>
</tr>
</tbody>
</table>

* These instructions use the memory access pipeline.

Data transfer instructions are 32-bit instructions.

PREF instruction uses the memory access pipeline.
<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operation instructions</td>
<td>Inter-register arithmetic operation instructions (excluding multiply instructions)</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>ADD Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADD #imm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADDC Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADDV Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMP/EQ #imm,R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMP/EQ Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMP/HS Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMP/GE Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMP/PI Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CMP/POS Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIV1 Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIV0S Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIV0U</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DT Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EXTS.B Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EXTS.W Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EXTU.B Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EXTU.W Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NEG Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NEG.C Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUB Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUBC Rm,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SUBV Rm,Rn</td>
</tr>
<tr>
<td>Inter-register arithmetic operations instructions (excluding multiply instructions and DIVU or DIVS instructions)</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td>—</td>
<td>CLIPU.B Rn</td>
</tr>
<tr>
<td>CLIP instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CLIPU.W Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CLIPS.B Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CLIPS.W Rn</td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Arithmetic operation instructions</td>
<td>Multiply-and-accumulate instruction</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>• This instruction locks the multiplier for 4 states.</td>
<td>MAC.W  @Rm+,@Rn+</td>
</tr>
<tr>
<td></td>
<td>Double-precision multiply-and-accumulate instruction</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>• This instruction locks the multiplier for 5 states.</td>
<td>MAC.L  @Rm+,@Rn+</td>
</tr>
<tr>
<td>Multiply instructions</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
<td>• These instructions lock the multiplier for 2 states.</td>
<td>MULS.W  Rm,Rn  MULU.W  Rm,Rn</td>
</tr>
<tr>
<td>Double-precision multiply instructions</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td></td>
<td>• These instructions lock the multiplier for 2 states.</td>
<td>DMULS.L  Rm,Rn  DMULU.L  Rm,Rn  MUL.L  Rm,Rn  MULR  R0,Rn</td>
</tr>
<tr>
<td>DIVU instruction</td>
<td>36</td>
<td>34</td>
<td>34</td>
<td></td>
<td>• These instructions use the shift register.</td>
<td>DIVU  R0,Rn</td>
</tr>
<tr>
<td>DIVS instruction</td>
<td>38</td>
<td>36</td>
<td>36</td>
<td></td>
<td>—</td>
<td>DIVS  R0,Rn</td>
</tr>
<tr>
<td>Logical operation instructions</td>
<td>Register-register logical operation instructions</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>AND  Rm,Rn  AND  #imm,R0  NOT  Rm,Rn  OR  Rm,Rn  OR  #imm,R0  TST  Rm,Rn  TST  #imm,R0  XOR  Rm,Rn  XOR  #imm,R0</td>
</tr>
<tr>
<td>Memory logical operation instructions</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td>• These instructions use the memory access pipeline.</td>
<td>AND.B  #imm,@(R0,GRB)  OR.B  #imm,@(R0,GRB)  TST.B  #imm,@(R0,GRB)  XOR.B  #imm,@(R0,GRB)</td>
</tr>
<tr>
<td>TAS instruction</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td></td>
<td>• This instruction uses the memory access pipeline.</td>
<td>TAS.B  @Rn</td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Bit manipulation</td>
<td>Register-register bit</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>BLD #imm3,Rn</td>
</tr>
<tr>
<td>instructions</td>
<td>operation instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BSET #imm3,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BCLR #imm3,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BST #imm3,Rn</td>
</tr>
<tr>
<td>Memory-T-bit bit</td>
<td></td>
<td>5</td>
<td>3</td>
<td>3</td>
<td></td>
<td>BAND.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BANDNOT.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BOR.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>Memory bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BORNOT.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>Manipulation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BLD.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BLDNOT.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BXOR.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>Shift</td>
<td></td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td>BST.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BCLR.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BSET.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• These are 32-bit instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• These instructions use the memory access pipeline.</td>
</tr>
</tbody>
</table>

- These are 32-bit instructions.
- These instructions use the shift pipeline.
<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch instructions</td>
<td>Conditional branch instructions</td>
<td>3</td>
<td>3/1*1</td>
<td>3/1*1</td>
<td>* These instructions use the branch pipeline.</td>
<td>BF label</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BT label</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delayed conditional branch</td>
<td>3</td>
<td>2/1*1</td>
<td>2/1*1</td>
<td>* These instructions use the branch pipeline.</td>
<td>BS/F label</td>
</tr>
<tr>
<td></td>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td>BT/S label</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unconditional branch instructions</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>* These instructions use the branch pipeline.</td>
<td>BRA label</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BRAF Rm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BSR label</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BSRF Rm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JMP @Rm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JSR @Rm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unconditional branch instructions with no delay</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>* These instructions use the branch pipeline.</td>
<td>JSR/N @Rm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RTS/N</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RTV/N Rm</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>* This instruction uses the branch pipeline.</td>
<td>JSR/N @@(disp,TBR)</td>
</tr>
<tr>
<td>System control instructions</td>
<td>System control</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>CLRT</td>
</tr>
<tr>
<td></td>
<td>ALU instructions</td>
<td></td>
<td></td>
<td></td>
<td>—</td>
<td>LDC Rm,SR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>—</td>
<td>LDC Rm,GBR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDC Rm,TBR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDC Rm,VBR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDS Rm,PR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SETT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>—</td>
<td>STC SR,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STC GBR,Rn</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>STC TBR,Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STC VBR,Rn</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STS PR,Rn</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>System control instructions</td>
<td>LDC.L instructions</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>LDC.L @Rm+,SR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDC.L @Rm+,GBR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDC.L @Rm+,VBR</td>
</tr>
<tr>
<td></td>
<td>STC.L instructions</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>STC.L SR,@-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td>STC.L GBR,@-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STC.L VBR,@-Rn</td>
</tr>
<tr>
<td></td>
<td>LDS.L instruction (PR)</td>
<td>5</td>
<td>1</td>
<td></td>
<td>2</td>
<td>LDS.L @Rm+,PR</td>
</tr>
<tr>
<td></td>
<td>STS.L instruction (PR)</td>
<td>4</td>
<td>1</td>
<td></td>
<td>1</td>
<td>STS.L PR,@-Rn</td>
</tr>
<tr>
<td>Register → MAC transfer instructions</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>• These instructions lock the multiplier for 1 state.</td>
<td>CLRMAC</td>
<td></td>
</tr>
<tr>
<td>Memory → MAC transfer instructions</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>• These instructions lock the multiplier for 2 states.</td>
<td>LDS Rm,MACH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDS Rm,MACL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDS Rm,MACL</td>
</tr>
<tr>
<td>MAC → register transfer instructions</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>• These instructions use the multiplication result read path.</td>
<td>STS MACH,Rn</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STS MACL,Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STS MACL,Rn</td>
</tr>
<tr>
<td>MAC → memory transfer instructions</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>• These instructions use the multiplication result read path.</td>
<td>STS.L MACH,@-Rn</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STS.L MACL,@-Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STS.L MACL,@-Rn</td>
</tr>
<tr>
<td>RTE instruction</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>RTE</td>
<td></td>
</tr>
<tr>
<td>RESBANK instruction</td>
<td>11/23&lt;sup&gt;2&lt;/sup&gt;</td>
<td>9/19&lt;sup&gt;2&lt;/sup&gt;</td>
<td>8/20&lt;sup&gt;2&lt;/sup&gt;</td>
<td>When the BO bit is 1, this instruction uses the memory access pipeline.</td>
<td>RESBANK</td>
<td></td>
</tr>
<tr>
<td>LDBANK instruction</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>—</td>
<td>LDBANK @Rm,R0</td>
<td></td>
</tr>
<tr>
<td>STBANK instruction</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>—</td>
<td>STBANK R0,@Rn</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>System control</td>
<td>TRAP instruction</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>—</td>
<td>TRAPA #imm</td>
</tr>
<tr>
<td></td>
<td>SLEEP instruction</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>—</td>
<td>SLEEP</td>
</tr>
<tr>
<td>FPU load/store instructions</td>
<td>LDS instruction</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>LDS Rm,FPUL</td>
</tr>
<tr>
<td></td>
<td>LDS instruction</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>LDS Rm,FPSCR</td>
</tr>
<tr>
<td></td>
<td>LDS instruction</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>LDS L @Rm+, FPUL</td>
</tr>
<tr>
<td></td>
<td>LDS instruction</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>LDS L @Rm+, FPSCR</td>
</tr>
<tr>
<td></td>
<td>LDS instruction</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>LDS L @Rm+, FPSCR</td>
</tr>
<tr>
<td></td>
<td>LDS instruction</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LDS L @Rm+, FPSCR</td>
</tr>
</tbody>
</table>

*These instructions use the memory access pipeline.*
<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-precision</td>
<td>Floating-point</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>• These instructions use the FPU load/store pipeline.</td>
<td>FLDS FRm,FPUL, FMOV FRm,FRn, FSTS FPUL,FRn</td>
</tr>
<tr>
<td>floating-point</td>
<td>register-transfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point</td>
<td>Floating-point</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>• These instructions use the FPU load/store pipeline.</td>
<td>FLDI0 FRn, FLDI1 FRn</td>
</tr>
<tr>
<td>register-immediate</td>
<td>register-transfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSCHG</td>
<td>FSCHG</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>• This instruction uses the FPU arithmetic operation pipeline.</td>
<td>FSCHG</td>
</tr>
<tr>
<td>instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point</td>
<td>Floating-point</td>
<td>5</td>
<td>1</td>
<td>0/2^*2</td>
<td>1/2^*2</td>
<td>0/2^*3</td>
</tr>
<tr>
<td>register load instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point</td>
<td>Floating-point</td>
<td>4</td>
<td>1</td>
<td>0/2^*2</td>
<td>1/0^*2</td>
<td>0</td>
</tr>
<tr>
<td>register store instructions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Single-precision floating-point instructions</td>
<td>Floating-point operation instructions (excluding FDIV)</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>• These instructions use the FPU arithmetic operation pipeline.</td>
<td>FADD FRm,FRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>• These instructions use the FPU load/store pipeline.</td>
<td>FABS FRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>1</td>
<td>11</td>
<td>• These instructions use the FPU arithmetic operation pipeline and FPU division/square root extraction pipeline.</td>
<td>FDIV FRm,FRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>1</td>
<td>11</td>
<td>• These instructions use the FPU arithmetic operation pipeline and FPU division/square root extraction pipeline.</td>
<td>FSQRT FRn</td>
</tr>
<tr>
<td></td>
<td>Floating-point compare instructions</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>• These instructions use the FPU arithmetic operation pipeline.</td>
<td>FCMP/EQ FRm,FRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FCMP/GT FRm,FRn</td>
</tr>
<tr>
<td>Type</td>
<td>Category</td>
<td>Number of Stages</td>
<td>Execution States</td>
<td>Latency</td>
<td>Contention</td>
<td>Instructions</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>----------</td>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>Double-precision floating-point instructions</td>
<td>Floating-point register-register transfer instructions</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>• These instructions use the FPU load/store pipeline.</td>
<td>FMOV DRm,DRn</td>
</tr>
<tr>
<td>Floating-point register-register transfer instructions</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0/2/3/4</td>
<td>• These instructions use the FPU arithmetic operation pipeline.</td>
<td>FCNVSD FPUL,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FCNVDS DRm,FPUL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D @Rm,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D @Rm+,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D @(R0,Rm),DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D @(disp12,Rm),DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D DRm,@Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D DRm,@-Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D DRm,@ (R0,Rn)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FMOV.D DRm,@(disp12,Rn)</td>
</tr>
</tbody>
</table>
### Section 8  Pipeline Operation

<table>
<thead>
<tr>
<th>Type</th>
<th>Category</th>
<th>Number of Stages</th>
<th>Execution States</th>
<th>Latency</th>
<th>Contention</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-precision floating-point instructions (excluding FDIV)</td>
<td>Floating-point operation instructions</td>
<td>10</td>
<td>1</td>
<td>0/8/7/8**</td>
<td>These instructions use the FPU arithmetic operation pipeline.</td>
<td>FADD DRm,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td>0/4*3</td>
<td></td>
<td>FMUL DRm,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td>0/4/3/4**</td>
<td></td>
<td>FSUB DRm,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td>0</td>
<td></td>
<td>FTRC DRm,FPUL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• These instructions use the FPU load/store pipeline.</td>
<td>FLOAT FPUL,DRn</td>
</tr>
<tr>
<td>Floating-point operation instructions (FDIV, FSQRT)</td>
<td></td>
<td>27</td>
<td>1</td>
<td>0/25/24/25**</td>
<td>These instructions use the FPU arithmetic operation pipeline.</td>
<td>FABS DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>26</td>
<td>1</td>
<td>0/24/23/24**</td>
<td>• These instructions use the FPU arithmetic operation pipeline and FPU division/square root extraction pipeline. Floating-point compare instructions</td>
<td>FNEG DRn</td>
</tr>
<tr>
<td>Floating-point compare instructions</td>
<td></td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>• These instructions use the FPU arithmetic operation pipeline.</td>
<td>FCMP/EQ DRm,DRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FCMP/GT DRm,DRn</td>
</tr>
</tbody>
</table>

**Notes:**
1. 1 state when a branch is not performed.
2. Number of stages, execution states, and latency are shown in BO bit = 0/BO bit = 1 order.
3. Latency is shown in CPU register/FPU register order.
4. Latency is shown in the following order: in case of use as CPU register/single-precision register; in case of use as FRn even number side/single-precision register; in case of use as FRn odd number side/double-precision register.
8.9.1 Data Transfer Instructions

(1) Register-Register Transfer Instructions (MOV Rm,Rn)

Instruction Type

\[
\text{MOV} \quad \text{Rm, Rn}
\]

Pipeline

\[
\begin{array}{c|c|c|c|c}
\text{Instruction A} & \text{IF} & \text{ID} & \text{EX} & \text{Slots} \\
\hline
\text{Next instruction} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\end{array}
\]

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, data transfer is performed via the ALU.

Instruction Issuance

This instruction does not cause resource contention.

Parallel Execution Capability

This is a zero-latency instruction. Parallel execution is possible even when this instruction is executed as a preceding instruction and the succeeding instruction uses Rn.
(2) Register-Register Transfer Instructions (20-Bit Immediate Value)

Instruction Types

MOVI20 #imm20,Rn
MOVI20S #imm20,Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, data transfer is performed via the ALU.

Instruction Issuance

These instructions do not cause resource contention.

Parallel Execution Capability

These are 32-bit instructions, and cannot be used in parallel execution. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction.)
(3) Register-Register Transfer Instructions  
(Excluding MOV Rm,Rn, MOV120, and MOV120S)

Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>#imm,Rn</td>
</tr>
<tr>
<td>MOVA</td>
<td>@(disp,PC),R0</td>
</tr>
<tr>
<td>MOVT</td>
<td>Rn</td>
</tr>
<tr>
<td>MOVRT</td>
<td>Rn</td>
</tr>
<tr>
<td>SWAP.B</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>SWAP.W</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>XTRCT</td>
<td>Rm,Rn</td>
</tr>
<tr>
<td>NOTT</td>
<td>Rn</td>
</tr>
</tbody>
</table>

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, data transfer is performed via the ALU.

Instruction Issuance

The SWAP.B, SWAP.W, and XTRCT instructions use the shifter. The other instructions do not cause resource contention.

Parallel Execution Capability

No particular comments
(4) Memory Load Instructions

Instruction Types

- MOV.W @(disp,PC),Rn
- MOV.L @(disp,PC),Rn
- MOV.B @(Rm,Rm),Rn
- MOV.W @(Rm,Rm),Rn
- MOV.L @(Rm,Rm),Rn
- MOV.B @(Rm,Rm),Rn
- MOV.W @(R0,Rm),Rn
- MOV.L @(R0,Rm),Rn
- MOV.B @(R0,Rm),Rn
- MOV.W @(R0,GPR),R0
- MOV.L @(R0,GPR),R0
- MOV.B @(R0,GPR),R0

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Operation

The pipeline has five stages: IF, ID, EX, MA, WB. Contention may occur if an instruction that uses the destination register of this instruction is among the three instructions following this instruction. (See section 8.5, Effect of Memory Load Instruction on Pipeline.)

Instruction Issuance

These instructions use the memory access pipeline.
Parallel Execution Capability

No particular comments
(5) Memory Load Instructions (12-Bit Displacement)

**Instruction Types**

- MOV.B @(disp12,Rm),Rn
- MOV.W @(disp12,Rm),Rn
- MOV.L @(disp12,Rm),Rn
- MOVU.B @(disp12,Rm),Rn
- MOVU.W @(disp12,Rm),Rn

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

The pipeline has five stages: IF, ID, EX, MA, WB. Contention may occur if an instruction that uses the destination register of this instruction is located within the 2 instructions following this instruction. (See section 8.5, Effect of Memory Load Instruction on Pipeline.)

**Instruction Issuance**

These instructions use the memory access pipeline.

**Parallel Execution Capability**

These are 32-bit instructions, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction.)
(6) Memory Load Instructions (MOVMU.L, MOVML.L)

Instruction Types

MOVMU.L @R15+,Rn
MOVML.L @R15+,Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>...</th>
<th>MA</th>
<th>MA</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>...</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>...</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Operation

These instructions perform restoration from the stack. The pipeline is in the form IF, ID, EX, MA, MA, MA, ... MA, WB, with MA repeated as often as necessary. Contention may occur if an instruction that uses the destination register of this instruction is located within the 3 instructions following this instruction. (See section 8.5, Effect of Memory Load Instruction on Pipeline.)

Instruction Issuance

If there is an uncompleted instruction in the pipeline when these instructions are decoded, execution of these instructions will be delayed. These instructions use the memory access pipeline.

Parallel Execution Capability

These are multi-cycle instructions, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(7) Memory Store Instructions

Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.B</td>
<td>Rm, @(Rn)</td>
</tr>
<tr>
<td>MOV.W</td>
<td>Rm, @(Rn)</td>
</tr>
<tr>
<td>MOV.L</td>
<td>Rm, @(Rn)</td>
</tr>
<tr>
<td>MOV.B</td>
<td>Rm, @(R^n)</td>
</tr>
<tr>
<td>MOV.W</td>
<td>Rm, @(R^n)</td>
</tr>
<tr>
<td>MOV.L</td>
<td>Rm, @(R^n)</td>
</tr>
<tr>
<td>MOV.B</td>
<td>R0, @(Rn+1)</td>
</tr>
<tr>
<td>MOV.W</td>
<td>R0, @(Rn+1)</td>
</tr>
<tr>
<td>MOV.L</td>
<td>R0, @(Rn+1)</td>
</tr>
<tr>
<td>MOV.B</td>
<td>R0, @(disp,Rn)</td>
</tr>
<tr>
<td>MOV.W</td>
<td>R0, @(disp,Rn)</td>
</tr>
<tr>
<td>MOV.L</td>
<td>R0, @(disp,Rn)</td>
</tr>
<tr>
<td>MOV.B</td>
<td>R0, @(R0,Rn)</td>
</tr>
<tr>
<td>MOV.W</td>
<td>R0, @(R0,Rn)</td>
</tr>
<tr>
<td>MOV.L</td>
<td>R0, @(R0,Rn)</td>
</tr>
<tr>
<td>MOV.B</td>
<td>R0, @(disp,GBR)</td>
</tr>
<tr>
<td>MOV.W</td>
<td>R0, @(disp,GBR)</td>
</tr>
<tr>
<td>MOV.L</td>
<td>R0, @(disp,GBR)</td>
</tr>
</tbody>
</table>

Pipeline

```
<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Operation

The pipeline ends after four stages: IF, ID, EX, MA. There is no WB stage as there is no return of data to the register.

Instruction Issuance

These instructions use the memory access pipeline.
Parallel Execution Capability

No particular comments
(8) Memory Store Instructions (12-Bit Displacement)

**Instruction Types**

- MOV.B  Rm, @(disp12, Rn)
- MOV.W  Rm, @(disp12, Rn)
- MOV.L  Rm, @(disp12, Rn)

**Pipeline**

![Pipeline Slots Diagram]

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>

**Operation**

The pipeline ends after four stages: IF, ID, EX, MA. There is no WB stage as there is no return of data to the register.

**Instruction Issuance**

These instructions use the memory access pipeline.

**Parallel Execution Capability**

These are 32-bit instructions, and cannot be used in parallel execution. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction.)
(9) Memory Store Instructions (MOVMU.L, MOVML.L)

Instruction Types

- MOVMU.L  Rm, @-R15
- MOVML.L  Rm, @-R15

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>...</th>
<th>MA</th>
<th>MA</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>...</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>...</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

These instructions perform saving to the stack. The pipeline is in the form IF, ID, EX, MA, MA, MA, ... MA, with MA repeated as often as necessary. There is no WB stage as there is no return of data to the register.

Instruction Issuance

If there is an uncompleted instruction in the pipeline when these instructions are decoded, execution of these instructions will be delayed.
These instructions use the memory access pipeline.

Parallel Execution Capability

These are multi-cycle instructions, and cannot be executed in parallel with a subsequent instruction.
(10) PREF Instruction

Instruction Type

PREF @Rm

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after four stages: IF, ID, EX, MA. There is no WB stage as there is no return of data to the register.

Instruction Issuance

This instruction uses the memory access pipeline.

Parallel Execution Capability

No particular comments
8.9.2 Arithmetic Operation Instructions

(1) Inter-Register Arithmetic Operation Instructions
(Excluding Multiply Instructions and DIVU or DIVS Instructions)

### Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>ADD #imm</td>
<td>#imm, Rn</td>
</tr>
<tr>
<td>ADDC</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>ADDV</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CMP/EQ #imm</td>
<td>#imm, R0</td>
</tr>
<tr>
<td>CMP/EQ</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CMP/HS</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CMP/GE</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CMP/HI</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CMP/GT</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CMP/PZ</td>
<td>Rn</td>
</tr>
<tr>
<td>CMP/PL</td>
<td>Rn</td>
</tr>
<tr>
<td>CMP/STR</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>DIV1</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>DIV0S</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>DIV0U</td>
<td>Rn</td>
</tr>
<tr>
<td>DT</td>
<td>Rn</td>
</tr>
<tr>
<td>EXTS.B</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>EXTS.W</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>EXTU.B</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>EXTU.W</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>NEG</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>NEG.C</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>SUB</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>SUBC</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>SUBV</td>
<td>Rn, Rn</td>
</tr>
<tr>
<td>CLIPU.B</td>
<td>Rn</td>
</tr>
<tr>
<td>CLIPU.W</td>
<td>Rn</td>
</tr>
</tbody>
</table>
Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, the data operation is completed via the ALU.

Instruction Issuance

The EXTS.B, EXTS.W, EXTU.B, and EXTU.W instructions use the shifter. The other instructions do not cause resource contention.

Parallel Execution Capability

With CLIP instructions, CS bit rewrite contention does not occur and parallel execution is possible.
(2) Multiply-and-Accumulate Instruction

**Instruction Type**

\[ \text{MAC.W } @Rm+,@Rn+ \]

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

The pipeline ends after seven stages: IF, ID, EX, MA, MA, mm, mm. mm indicates a state in which the multiplier is operating.

See section 8.7, Contention Due to Multiplier, for general pipeline details. This instruction has three execution slots, a latency of five, and four lock states. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When a MAC.W instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention.

<table>
<thead>
<tr>
<th>MAC.W @Rm+,@Rn+</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

(b) When a MAC.W instruction is immediately followed by a MULS.W, MULU.W, DMULS.W, DMULU.W, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction

As the MAC.W instruction locks the multiplier, stalling occurs a further 2-slot interval back.
(c) When a MAC.W instruction is immediately followed by an LDS.L (memory) instruction
Execution is delayed for a MAC execution state (3-slot) interval.

<table>
<thead>
<tr>
<th>MAC.W @Rm+,@Rn+</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS MACL,Rn</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

**Instruction Issuance**

This instruction uses the memory access pipeline.
This instruction uses the multiplier.
This instruction is executed even if the multiplier is locked.
This instruction locks the multiplier for a 4-slot interval.

**Parallel Execution Capability**

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(3) Double-Precision Multiply-and-Accumulate Instruction

Instruction Type

MAC.L @Rm+,@Rn+

Pipeline

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Instructions</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction A</td>
<td>IF</td>
<td>——</td>
<td>——</td>
<td>ID</td>
<td>EX</td>
<td>——</td>
<td>——</td>
<td>——</td>
<td>——</td>
</tr>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>——</td>
<td>——</td>
<td>ID</td>
<td>EX</td>
<td>——</td>
<td>——</td>
<td>——</td>
<td>——</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>——</td>
<td>——</td>
<td>ID</td>
<td>EX</td>
<td>——</td>
<td>——</td>
<td>——</td>
<td>——</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after eight stages: IF, ID, EX, MA, MA, mm, mm, mm. mm indicates a state in which the multiplier is operating.

See section 8.7, Contention Due to Multiplier, for general pipeline details. This instruction has four execution slots, a latency of six, and five lock states. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When a MAC.L instruction is immediately followed by a MAC.L or MAC.W instruction

There is no multiplier contention.

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Instructions</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC.L @Rm+,@Rn+</td>
<td>IF</td>
<td>——</td>
<td>——</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>MA</td>
<td>mm</td>
<td>mm</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>——</td>
<td>——</td>
<td>——</td>
<td>——</td>
<td>ID</td>
<td>EX</td>
<td>——</td>
<td>——</td>
</tr>
</tbody>
</table>
(b) When a MAC.L instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction
As the MAC.L instruction locks the multiplier, stalling occurs a further 2 states back.

| MAC.L @Rm+,@Rn+ | IF | ID | EX | MA | MA | mm | mm | mm | Slates |
|-----------------|----|----|----|----|----|----|----|----|
| STS MACH,Rn     | IF | —  | —  | —  | —  | —  | —  | —  | ID | EX | WB |
| Instruction after next | IF | —  | —  | —  | —  | —  | —  | —  | ID | EX | ... |

(c) When a MAC.L instruction is immediately followed by an LDS.L (memory) instruction
Execution is delayed for a MAC execution state (4-slot) interval.

| MAC.L @Rm+,@Rn+ | IF | ID | EX | MA | MA | mm | mm | mm | Slates |
|-----------------|----|----|----|----|----|----|----|----|
| LDS.L @Rn+,MACL | IF | —  | —  | —  | —  | —  | —  | —  | ID | EX | MA | WB |
| Instruction after next | IF | —  | —  | —  | —  | —  | —  | —  | ID | EX | ... |

**Instruction Issuance**

This instruction uses the memory access pipeline.
This instruction uses the multiplier.
This instruction is executed even if the multiplier is locked.
This instruction locks the multiplier for a 5-slot interval.

**Parallel Execution Capability**

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(4) Multiply Instructions

Instruction Types

- MULS.W  Rm, Rn
- MULU.W  Rm, Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>⋯</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>⋯</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after four stages: IF, ID, mm, mm. mm indicates a state in which the multiplier is operating.

See section 8.7, Contention Due to Multiplier, for general pipeline details. These instructions have one execution slot, a latency of two, and one lock state. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When a MULS.W instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention.

| MULS.W | IF | ID | mm | mm |
| MAC.W | IF | ID | EX | MA | MA | mm | mm |
| Instruction after next | IF | — | ID | EX | ⋯ |
(b) When a MULS.W instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction

As the MULS.W instruction locks the multiplier, parallel execution is not possible.

<table>
<thead>
<tr>
<th>MULS.W</th>
<th>Rm,Rn</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS</td>
<td>MACL,Rn</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

(c) When a MULS.W instruction is immediately followed by an LDS.L (memory) instruction

Parallel execution with the MULS.W instruction is not possible, as it locks the multiplier.

<table>
<thead>
<tr>
<th>MULS.W</th>
<th>Rm,Rn</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS.L</td>
<td>@Rn+,MACL</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Issuance**

These instructions use the multiplier.
These instructions lock the multiplier for a 1-slot interval.

**Parallel Execution Capability**

No particular comments
(5) Double-Precision Multiply Instructions

Instruction Types

- DMULS.L Rm, Rn
- DMULU.L Rm, Rn
- MUL.L Rm, Rn

Pipeline

```
               <--- <--- <--- <--- <--- Slots

  Instruction A  | IF | ID | mm | mm | mm
  Next instruction | IF | —  | ID | EX | ···
  Instruction after next | IF | —  | ID | EX | ···
```

Operation

The pipeline ends after five stages: IF, ID, mm, mm, mm. mm indicates a state in which the multiplier is operating.

See section 8.7, Contention Due to Multiplier, for general pipeline details. These instructions have two execution slots, a latency of three, and two lock states. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When a MUL.L instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention.

```
               <--- <--- <--- <--- <--- <--- <--- <--- Slots

  MUL.L Rm, Rn   | IF | ID | mm | mm | mm
  MAC.L @Rm+, @Rn+ | IF | —  | ID | EX | MA | MA | mm | mm | mm
  Instruction after next | IF | —  | —  | —  | —  | ID | EX | ···
```
(b) When a MUL.L instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction

As the MUL.L instruction locks the multiplier, stalling occurs a further 2-slot interval back.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.L Rm,Rn</td>
<td>IF</td>
<td>ID</td>
<td>mm</td>
<td>mm</td>
<td>mm</td>
</tr>
<tr>
<td>STS</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
</tbody>
</table>

(c) When a MUL.L instruction is immediately followed by an LDS.L (memory) instruction

Execution is delayed during execution of MUL.L (two cycles).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.L</td>
<td>IF</td>
<td>ID</td>
<td>mm</td>
<td>mm</td>
<td>mm</td>
</tr>
<tr>
<td>LDS.L @Rn+,MACL</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
</tbody>
</table>

**Instruction Issuance**

These instructions use the multiplier.

These instructions lock the multiplier for a 2-slot interval.

**Parallel Execution Capability**

These are multi-cycle instructions, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(6) Double-Precision Multiply Instruction (General Register Return)

Instruction Type

MULR  R0, Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after six stages: IF, ID, mm, mm, mm, WB. mm indicates a state in which the multiplier is operating.

See section 8.7, Contention Due to Multiplier, for general pipeline details. This instruction has two execution slots, a latency of four, and two lock states. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When a MULR instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention.

<table>
<thead>
<tr>
<th>MULR  R0,Rn</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC.L  @Rm+,@Rn+</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>MA</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>
(b) When a MULR instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction
As the MULR instruction locks the multiplier, stalling occurs a further 1-slot interval back.

↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slot

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULR R0,Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td></td>
<td></td>
<td></td>
<td>mm</td>
<td>mm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slot

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS MACL,Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) When a MULR instruction is immediately followed by an STS (register) or STS.L (memory) instruction
As the MULR instruction locks the multiplier, and multiplication result read path contention occurs, stalling occurs a further 2-slot interval back.

↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slot

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS MACL,Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) When a MULR instruction is immediately followed by an LDS.L (memory) instruction
Execution is delayed for a MULR instruction execution state (2-slot) interval.

↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slot

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS.L @Rn+,MACL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Issuance**

This instruction uses the multiplier.
This instruction locks the multiplier for a 2-slot interval.
This instruction uses the multiplication result read path.

**Parallel Execution Capability**

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(7) DIVU Instruction

Instruction Type

\[ \text{DIVU } R0, Rn \]

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>⋮</th>
<th>EX</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after 36 stages: IF, ID, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX. Data operations are completed using the ALU in the EX stages.

Instruction Issuance

This instruction uses the shift pipeline.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(8) DIVS Instruction

Instruction Type

DIVS  R0, Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>EX</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after 38 stages: IF, ID, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX, EX. Data operations are completed using the ALU in the EX stages.

Instruction Issuance

This instruction do not cause resource contention.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
8.9.3 Logical Operation Instructions

(1) Register-Register Logical Operation Instructions

Instruction Types

- **AND**  Rm, Rn
- **AND**  #imm, R0
- **NOT**  Rm, Rn
- **OR**  Rm, Rn
- **OR**  #imm, R0
- **TST**  Rm, Rn
- **TST**  #imm, R0
- **XOR**  Rm, Rn
- **XOR**  #imm, R0

Pipeline

```
↔ ↔ ↔ ↔ ↔ ↔ Slots
```

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, the data operation is completed via the ALU.

Instruction Issuance

These instructions do not cause resource contention.

Parallel Execution Capability

No particular comments
(2) Memory Logical Operation Instructions

Instruction Types

- **AND.B #imm, @(R0, GBR)**
- **OR.B #imm, @(R0, GBR)**
- **XOR.B #imm, @(R0, GBR)**

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

The pipeline ends after six stages: IF, ID, EX, MA, EX, MA.

**Instruction Issuance**

These instructions use the memory access pipeline.

**Parallel Execution Capability**

These are multi-cycle instructions, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(3) Memory Logical Operation Instructions

Instruction Type

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

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after five stages: IF, ID, EX, MA, EX.

Instruction Issuance

This instruction uses the memory access pipeline.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(4) TAS Instruction

Instruction Type
TAS.B @Rn

Pipeline

\[\begin{array}{cccccc}
\text{Instruction A} & \text{IF} & \text{ID} & \text{EX} & \text{MA} & \text{EX} & \text{MA} \\
\text{Next instruction} & \text{IF} & - & - & \text{ID} & \text{EX} & \cdots \\
\text{Instruction after next} & \text{IF} & - & - & \text{ID} & \text{EX} & \cdots \\
\end{array}\]

Operation
The pipeline ends after six stages: IF, ID, EX, MA, EX, MA.

Instruction Issuance
This instruction uses the memory access pipeline.

Parallel Execution Capability
This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(5) Register-Register Bit Operation Instructions

**Instruction Types**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLD</td>
<td>#imm3,Rn</td>
</tr>
<tr>
<td>BSET</td>
<td>#imm3,Rn</td>
</tr>
<tr>
<td>BCLR</td>
<td>#imm3,Rn</td>
</tr>
<tr>
<td>BST</td>
<td>#imm3,Rn</td>
</tr>
</tbody>
</table>

**Pipeline**

```
Instruction A | IF | ID | EX
Next instruction | IF | ID | EX | ...
Instruction after next | IF | ID | EX | ...
```

**Operation**

The pipeline ends after three stages: IF, ID, EX. In the EX stage, the data operation is completed via the ALU.

**Instruction Issuance**

These instructions do not cause resource contention.

**Parallel Execution Capability**

No particular comments
(6) Memory-Tbit Logical Operation Instructions

Instruction Types

BAND.B #imm3,@(disp12,Rn)
BANDNOT.B #imm3,@(disp12,Rn)
BLD.B #imm3,@(disp12,Rn)
BLDNOT.B #imm3,@(disp12,Rn)
BOR.B #imm3,@(disp12,Rn)
BORNOT.B #imm3,@(disp12,Rn)
BXOR.B #imm3,@(disp12,Rn)

Pipeline

<--- <--- <--- <--- <--- Slots

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after five stages: IF, ID, EX, MA, EX.

Instruction Issuance

These instructions use the memory access pipeline.

Parallel Execution Capability

These are 32-bit instructions, and cannot be used in parallel execution. If the instruction following this instruction is BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, or BXOR, the final step is executed in parallel with the instruction that follows. Parallel execution with the final step is not possible with any other instruction. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction).
### Pipeline Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAND.B #imm.@(disp12,Rn)</td>
<td>IF</td>
</tr>
<tr>
<td>BOR.B #imm.@(disp12,Rn)</td>
<td>IF</td>
</tr>
<tr>
<td>BANDNOT.B #imm.@(disp12,Rn)</td>
<td>IF</td>
</tr>
</tbody>
</table>

### Slots

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rm,Rn</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
</tr>
</tbody>
</table>

### Slots

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTCL</td>
<td>IF</td>
</tr>
<tr>
<td>BAND.B #imm.@(disp12,Rn)</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
</tr>
</tbody>
</table>
(7) Memory Bit Operation Instructions

Instruction Types

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCLR.B</td>
<td>#imm3,(disp12,Rn)</td>
</tr>
<tr>
<td>BSET.B</td>
<td>#imm3,(disp12,Rn)</td>
</tr>
<tr>
<td>BST.B</td>
<td>#imm3,(disp12,Rn)</td>
</tr>
</tbody>
</table>

Pipeline

```
IF ID EX MA EX MA
```

Table:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after six stages: IF, ID, EX, MA, EX, MA.

Instruction Issuance

These instructions use the memory access pipeline.

Parallel Execution Capability

These are 32-bit instructions, and cannot be used in parallel execution. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction.)
8.9.4 Shift Instructions

Instruction Types

ROTL Rn
ROTR Rn
ROTCL Rn
ROTCR Rn
SHAL Rn
SHAR Rn
SHLL Rn
SHLR Rn
SHLL2 Rn
SHLR2 Rn
SHLL8 Rn
SHLR8 Rn
SHLL16 Rn
SHLR16 Rn
SHAD Rm, Rn
SHLD Rm, Rn

Pipeline

Instruction A IF ID EX
Next instruction IF ID EX ... Innstript after next IF ID EX ...

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, the data operation is completed via the shifter.

Instruction Issuance

These instructions use the shift pipeline.
Parallel Execution Capability

No particular comments
8.9.5 Branch Instructions

(1) Conditional Branch Instructions

Instruction Types

<table>
<thead>
<tr>
<th>BF</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>label</td>
</tr>
</tbody>
</table>

Pipeline

(a) When condition is met

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>⋮</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>⋮</td>
<td>(Fetched but discarded)</td>
</tr>
<tr>
<td>Second instruction after next</td>
<td>IF</td>
<td>⋮</td>
<td>(Fetched but discarded)</td>
</tr>
<tr>
<td>Branch destination instruction</td>
<td>—</td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

(b) When condition is not met

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Second instruction after next</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after three stages: IF, ID, EX. Condition determination is performed in the ID stage. Conditional branch instructions are not delayed branch instructions.
(a) When condition is met
The branch destination address is calculated in the EX stage. All overrun-fetched instructions up to that point are discarded. The branch destination instruction fetch is started from the slot following the instruction A EX stage slot.

(b) When condition is not met
If it is determined in the ID stage that the condition is not met, processing proceeds with nothing done in the EX stage. The next instruction is fetched and executed.
A typical pipeline is shown below.
If the preceding instruction is a CMP instruction, execution is delayed by 1 cycle.

↔↔↔↔↔ Slots

<table>
<thead>
<tr>
<th>CMP</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>

Branch destination IF

If the preceding instruction is a single-precision FCMP instruction, execution is delayed by 2 cycles.

↔↔↔↔↔↔ Slots

<table>
<thead>
<tr>
<th>FCMP/single</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>

Branch destination IF

If the preceding instruction is a double-precision FCMP instruction, execution is delayed by 3 cycles.

↔↔↔↔↔↔↔ Slots

<table>
<thead>
<tr>
<th>FCMP/double</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>

Branch destination IF

Instruction Issuance
These instructions use the branch pipeline.

Parallel Execution Capability
No particular comments
(2) Delayed Conditional Branch Instructions

Instruction Types

BF/S label
BT/S label

Pipeline

(a) When condition is met

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay slot</td>
<td>IF</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>. . .</td>
<td>(Fetched but discarded)</td>
</tr>
<tr>
<td>Second instruction after next</td>
<td>IF</td>
<td>. . .</td>
<td>(Fetched but discarded)</td>
</tr>
<tr>
<td>Branch destination instruction</td>
<td>—</td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

(b) When condition is not met

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Second instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after three stages: IF, ID, EX. Condition determination is performed in the ID stage. Interrupts are not accepted in the delay slot.

(a) When condition is met

The branch destination address is calculated in the EX stage. All overrun-fetched instructions up to that point are discarded. The branch destination instruction fetch is started from the slot following the instruction A EX stage slot.
(b) When condition is not met

If it is determined in the ID stage that the condition is not met, processing proceeds with
nothing done in the EX stage. The next instruction is fetched and executed.

A typical pipeline is shown below.

If the preceding instruction is a CMP instruction, execution is delayed by 1 cycle.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF/S</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Delay slot</td>
<td>IF</td>
<td></td>
<td>ID</td>
</tr>
</tbody>
</table>

If the preceding instruction is a single-precision FCMP instruction, execution is delayed by 2 cycles.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMP/single</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF/S</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Delay slot</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
</tr>
</tbody>
</table>

If the preceding instruction is a double-precision FCMP instruction, execution is delayed by 3 cycles.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>IF</th>
<th>DF</th>
<th>E1</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMP/double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF/S</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>Delay slot</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Issuance**

These instructions use the branch pipeline.

If an instruction fetch has not yet been performed for the instruction (delay slot) immediately
following one of these instructions, execution of that instruction is delayed.

**Parallel Execution Capability**

No particular comments
(3) Unconditional Branch Instructions

Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA</td>
<td>label</td>
</tr>
<tr>
<td>BRAF</td>
<td>Rm</td>
</tr>
<tr>
<td>BSR</td>
<td>label</td>
</tr>
<tr>
<td>BSRF</td>
<td>Rm</td>
</tr>
<tr>
<td>JMP</td>
<td>@Rm</td>
</tr>
<tr>
<td>JSR</td>
<td>@Rm</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

Pipeline

```
  ⬛ ⬛ ⬛ ⬛ ⬛
Instruction A IF ID EX
Delay slot IF — — ID EX ⋯
Instruction after next IF ⋯ (Fetched but discarded)
Second instruction after next IF ⋯ (Fetched but discarded)
Branch destination instruction — IF ID EX ⋯
```

Operation

The pipeline ends after three stages: IF, ID, EX. Unconditional branch instructions are delayed branch instructions.

The branch destination address is calculated in the EX stage. The instruction after the unconditional branch instruction (instruction A) – that is, the delay slot instruction – is not discarded after being fetched, as with a conditional branch instruction, but is executed. However, the ID stage of this delay slot instruction is stalled for a 2-slot interval. The branch destination instruction fetch is started from the slot following the instruction A EX stage slot. Interrupts are not accepted in the delay slot.

Instruction Issuance

These instructions use the branch pipeline.

If an instruction fetch has not yet been performed for the instruction (delay slot) immediately following one of these instructions, execution of that instruction is delayed.
Parallel Execution Capability

No particular comments
(4) No Delay Unconditional Branch Instructions

Instruction Types

- **JSR/N** \( @Rm \)
- **RTS/N**
- **RTV/N** \( Rm \)

Pipeline

```
IF  ID  EX

Instruction A

Next instruction  IF  —  ⋄  ⋄  (Fetched but discarded)
Instruction after next  IF  ⋄  ⋄  (Fetched but discarded)
Second instruction after next  IF  ⋄  ⋄  (Fetched but discarded)
Branch destination instruction  —  IF  ID  EX  ⋄
```

Operation

The pipeline ends after three stages: IF, ID, EX. Condition determination is performed in the ID stage. Conditional branch instructions are not delayed branch instructions. The branch destination address is calculated in the EX stage. All overrun-fetched instructions up to that point are discarded. The branch destination instruction fetch is started from the slot following the instruction A EX stage slot.

Instruction Issuance

These instructions use the branch pipeline.

Parallel Execution Capability

No particular comments
(5) Unconditional Branch Instructions with No Delay (JSR/N @@(disp,TBR))

Instruction Types

\[ \text{JSR/N } @@(\text{disp},\text{TBR}) \]

Pipeline

\begin{tabular}{|c|c|c|c|c|}
\hline
Instruction A & IF & ID & EX & MA & EX \\
\hline
Next instruction & IF & - & - & - & - (Fetched but discarded) \\
Instruction after next & IF & - & - & - & - (Fetched but discarded) \\
Second instruction & IF & - & - & - & - (Fetched but discarded) \\
after next & Branch destination & - & - & - & - & IF & ID & EX & - \\
instruction & \\
\hline
\end{tabular}

Operation

The pipeline ends after five stages: IF, ID, EX, MA, EX. Condition determination is performed in the ID stage. This is not a delayed branch instruction. The branch destination address is calculated in the second EX stage. All overrun-fetched instructions up to that point are discarded. The branch destination instruction fetch is started from the slot following the slot with the second EX of instruction A.

Instruction Issuance

This instruction uses the branch pipeline.

This instruction uses the memory access pipeline.

Parallel Execution Capability

No particular comments
8.9.6 System Control Instructions

(1) System Control ALU Instructions

Instruction Types

- CLRT
- LDC Rm, GBR
- LDC Rm, TBR
- LDC Rm, VBR
- LDS Rm, PR
- NOP
- SETT
- STC GBR, Rn
- STC TBR, Rn
- STC VBR, Rn
- STS PR, Rn
- NOTT

Pipeline

```
Instruction A │ IF  ID  EX  
Next instruction │ IF  ID  EX  
Instruction after next │ IF  ID  EX  
```

Operation

The pipeline ends after three stages: IF, ID, EX. In the EX stage, the data operation is completed via the ALU.

Instruction Issuance

These instructions do not cause resource contention.

Parallel Execution Capability

No particular comments
(2) System Control ALU Instruction

**Instruction Type**

LDC  Rm, SR

**Pipeline**

```
  IF  ID  EX  EX  EX

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>⋯</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>⋯</td>
</tr>
</tbody>
</table>
```

**Operation**

The pipeline ends after five stages: IF, ID, EX, EX, EX. In the first EX stage, the data operation is completed via the ALU.

**Instruction Issuance**

This instruction does not cause resource contention.

**Parallel Execution Capability**

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(3) System Control ALU Instruction

Instruction Type

STC    SR, Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after four stages: IF, ID, EX, EX. In the second EX stage, the data operation is completed via the ALU.

Instruction Issuance

No particular comments

A typical pipeline when performing a CS bit read is shown below.

<table>
<thead>
<tr>
<th>CLIP</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>STC</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
</tr>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(4) LDC.L and LDS.L Instructions

Instruction Types

LDC.L @Rm+,GBR
LDC.L @Rm+,VBR
LDS.L @Rm+,PR

Pipeline

Instruction A IF ID EX MA WB
Next instruction IF ID EX ...
Instruction after next IF ID EX ...

Operation

The pipeline ends after five stages: IF, ID, EX, MA, WB.

Instruction Issuance

These instructions use the memory access pipeline.

Parallel Execution Capability

No particular comments
(5) LDC.L Instruction

Instruction Type

LDC.L @Rm+,SR

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after seven stages: IF, ID, EX, MA, EX, EX, EX.

Instruction Issue

This instruction uses the memory access pipeline.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(6) STC.L Instructions

Instruction Types

STC.L GBR, @-Rn
STC.L VBR, @-Rn
STS.L PR, @-Rn

Pipeline

```
Instruction A IF ID EX MA → → → → → → → → → → Slots
Next instruction IF ID EX ...
Instruction after next IF ID EX ...
```

Operation
The pipeline ends after four stages: IF, ID, EX, MA.

Instruction Issuance
These instructions use the memory access pipeline.

Parallel Execution Capability
No particular comments
(7) **STC.L Instruction**

**Instruction Type**

STC.L SR, @-Rn

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>

**Operation**

The pipeline ends after five stages: IF, ID, EX, EX, MA.

**Instruction Issuance**

This instruction uses the memory access pipeline.

**Parallel Execution Capability**

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.

Although this instruction uses the memory access pipeline, parallel execution is possible if the preceding instruction is a single-cycle memory access instruction.
(8) Register → MAC Transfer Instructions

Instruction Types

CLRMAC
LDS Rm, MACH
LDS Rm, MACL

Pipeline

\[
\begin{array}{c|cccc}
\text{Instruction A} & \text{IF} & \text{ID} & \text{mm} & \text{mm} \\
\hline
\text{Next instruction} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\end{array}
\]

Operation

The pipeline ends after four stages: IF, ID, mm, mm. mm indicates a state in which the multiplier is operating.

See section 8.7, Contention Due to Multiplier, for general pipeline details. These instructions have one execution slot, a latency of two, and one lock state. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When a CLRMAC instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention.

\[
\begin{array}{c|cccc|c}
\text{CLRMAC} & \text{IF} & \text{ID} & \text{mm} & \text{mm} & \text{mm} \\
\hline
\text{MAC.W } @Rm+, @Rn+ & \text{IF} & \text{ID} & \text{EX} & \text{MA} & \text{MA} & \text{mm} & \text{mm} \\
\text{Instruction after next} & \text{IF} & \text{—} & \text{—} & \text{ID} & \text{EX} & \ldots \\
\end{array}
\]
(b) When a CLRMAC instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction
Parallel execution with the CLRMAC instruction is not possible, as it locks the multiplier.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{CLRMAC} & \text{IF} & \text{ID} & \text{mm} & \text{mm} \\
\hline
\text{STS MACL},Rn & \text{IF} & \_ & \text{ID} & \text{EX} & \text{WB} \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \_ & \_ \\
\hline
\end{array}
\]

(c) When a CLRMAC instruction is immediately followed by an LDS.L (memory) instruction
Execution is delayed for a CLRMAC instruction execution state (1-slot) interval.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{CLRMAC} & \text{IF} & \text{ID} & \text{mm} & \text{mm} \\
\hline
\text{LDS.L @Rn+,MACL} & \text{IF} & \_ & \text{ID} & \text{EX} & \text{MA} & \text{WB} \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \_ & \_ \\
\hline
\end{array}
\]

**Instruction Issuance**

These instructions use the multiplier.

These instructions lock the multiplier for a 1-slot interval.

**Parallel Execution Capability**

No particular comments
(9) Memory ➔ MAC Transfer Instructions

Instruction Types

LDS.L @Rm+,MACH
LDS.L @Rm+,MACL

Pipeline

Operation

The pipeline ends after five stages: IF, ID, EX, MA, WB.

See section 8.7, Contention Due to Multiplier, for general pipeline details. This instruction has one execution slot, a latency of three, and two lock states. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When an LDS.L instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention, but there is memory access contention, with 1-cycle stalling.
(b) When an LDS.L instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction
As the LDS.L instruction locks the multiplier, stalling occurs a further 1-slot interval back.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS.L @Rm+,MACH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STS MACL,Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) When an LDS.L instruction is immediately followed by an LDS.L (memory) instruction
Execution is delayed for an LDS.L instruction execution state (1-slot) interval.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS.L @Rn+,MACL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDS.L @Rn+,MACL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Issuance**

These instructions use the memory access pipeline.
These instructions use the multiplier.
These instructions are executed if there is a remaining multiplication lock interval of 1.
These instructions lock the multiplier for a 2-slot interval.

**Parallel Execution Capability**

No particular comments
(10) MAC → Register Transfer Instructions

Instruction Types

STS  MACH, Rn
STS  MACL, Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after four stages: IF, ID, EX, WB.

See section 8.7, Contention Due to Multiplier, for general pipeline details. These instructions have one execution slot, a latency of two, and zero lock state. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When an STS instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention.

<table>
<thead>
<tr>
<th>STS  MACH, Rn</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC.W @Rm+, @Rn+,</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>
(b) When an STS instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction.

As the STS instruction does not lock the multiplier, parallel execution is performed.

<table>
<thead>
<tr>
<th></th>
<th>STS MACH,Rn</th>
<th>IF</th>
<th>ID</th>
<th>mm</th>
<th>mm</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MUL.L Rm,Rn</td>
<td>IF</td>
<td>ID</td>
<td>mm</td>
<td>mm</td>
<td>mm</td>
</tr>
<tr>
<td></td>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) When an STS instruction is immediately followed by a STS (register) or STS.L (memory) instruction.

Parallel execution is not possible, as contention occurs with the multiplication result read bus.

<table>
<thead>
<tr>
<th></th>
<th>STS MACH,Rn</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STS MACL,Rn</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td></td>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

(d) When an STS instruction is immediately followed by an LDS.L (memory) instruction.

Parallel execution is performed.

There is no multiplier contention.

<table>
<thead>
<tr>
<th></th>
<th>STS MACH,Rn</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDS.L @Rn+,MACL</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
</tr>
<tr>
<td></td>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Issuance

These instructions use the multiplier, but do not lock it.
These instructions use the multiplication result read path.

### Parallel Execution Capability

No particular comments
(11) MAC → Memory Transfer Instructions

Instruction Types

STS.L MACH,@-Rn
STS.L MACL,@-Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after four stages: IF, ID, EX, MA.

See section 8.7, Contention Due to Multiplier, for general pipeline details. These instructions have one execution slot, a latency of two, and zero lock state. Detailed examples where there are consecutive instructions relating to the pipeline of this instruction or the multiplier are given below.

(a) When an STS.L instruction is immediately followed by a MAC.W or MAC.L instruction

There is no multiplier contention, but there is memory access contention, with 1-cycle stalling.

<table>
<thead>
<tr>
<th>STS.L MACH, @-Rn</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC.W @Rm+, @Rn+</td>
<td>IF</td>
<td>—</td>
<td>EX</td>
<td>MA</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>
(b) When an STS.L instruction is immediately followed by a MULS.W, MULU.W, DMULS.L, DMULU.L, MUL.L, MULR, STS (register), STS.L (memory), or LDS (register) instruction. As the STS.L instruction does not lock the multiplier, parallel execution is performed.

\[ \begin{array}{c|cccc|c}
& \text{IF} & \text{ID} & \text{EX} & \text{MA} \\
\hline
\text{STS.L} & \text{MACL,}@-Rn & - & - & - \\
\text{MUL.L} & \text{Rm,Rn} & - & - & - \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\end{array} \]

(c) When an STS.L instruction is immediately followed by a STS (register) or STS.L (memory) instruction. Parallel execution is not possible, as contention occurs with the multiplication result read bus.

\[ \begin{array}{c|cccc|c}
& \text{IF} & \text{ID} & \text{EX} & \text{MA} \\
\hline
\text{STS.L} & \text{MACL,}@-Rn & - & - & - \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\end{array} \]

(d) When an STS.L instruction is immediately followed by an LDS.L (memory) instruction. Memory access pipeline contention occurs and parallel execution is not possible.

\[ \begin{array}{c|cccc|c}
& \text{IF} & \text{ID} & \text{EX} & \text{MA} \\
\hline
\text{LDS.L} & \text{@Rn+,MACL} & - & - & - \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \ldots \\
\end{array} \]

**Instruction Issuance**

These instructions use the memory access pipeline.

These instructions use the multiplier, but do not lock it.

These instructions use the multiplication result read path.

**Parallel Execution Capability**

No particular comments
(12) RTE Instruction

Instruction Type

RTE

Pipeline

\[\text{Instruction A} \quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MA} \quad \text{MA} \quad \text{EX} \quad \text{EX} \quad \text{EX}\]

\[\text{Delay slot} \quad \text{IF} \quad - \quad - \quad - \quad - \quad - \quad \text{ID} \quad \text{EX} \quad \cdots\]

\[\text{Branch destination} \quad \text{IF} \quad - \quad \text{ID} \quad \text{EX} \quad \cdots\]

Operation

The pipeline ends after eight stages: IF, ID, EX, MA, MA, EX, EX, EX. RTE is a delayed branch instruction. The ID stage of the delay slot instruction is stalled for a 5-slot interval. The IF stage of the branch destination instruction is started from the slot after the second MA stage of RTE.

Instruction Issuance

This instruction does not cause resource contention.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(13) RESBANK Instruction

Instruction Type

RESBANK

Pipeline

- When B0 == 0

\[
\text{Instruction A: } \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{EX} \quad \text{EX} \quad \text{EX} \quad \text{EX} \quad \text{EX} \quad \text{EX} \quad \text{EX} \quad \text{ID} \quad \text{EX} \quad \ldots
\]

- When B0 == 1

\[
\text{Instruction A: } \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MA} \quad \text{MA} \quad \text{MA} \quad \ldots \quad \text{MA} \quad \text{MA} \quad \text{MA} \quad \text{WB} \quad \text{ID} \quad \text{EX} \quad \ldots
\]

Operation

The operation is different when the BO bit is 0 and when the BO bit is 1.

When the BO bit is 0, restoration from a bank is performed. The pipeline comprises IF and ID followed by EX, EX, EX, EX, EX, EX (nine repetitions of EX), and ends after 11 stages. During this time, register restoration from the bank is performed.

When the BO bit is 1, restoration from the stack is performed. The pipeline comprises IF, ID, and EX, followed by MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, (19 repetitions of MA), followed by WB, and ends after 23 stages.

Instruction Issuance

When the BO bit is 0, this instruction does not cause resource contention.

When the BO bit is 1, this instruction uses the memory access pipeline.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(14) LDBANK Instruction

Instruction Type

LDBANK @Rm, R0

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>⋯</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after eight stages: IF, ID, EX, EX, EX, EX, EX, EX.

Instruction Issuance

This instruction does not cause resource contention.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(15) STBANK Instruction

Instruction Type

STBANK R0, @Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>ID</td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after nine stages: IF, ID, EX, EX, EX, EX, EX, EX, EX.

Instruction Issuance

This instruction does not cause resource contention.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(16) TRAP Instruction

Instruction Type

TRAPA #imm

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>⋯</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>⋯</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch destination</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation

The pipeline ends after eight stages: IF, ID, EX, EX, EX, MA, MA, MA. A TRAP instruction is not a delayed branch instruction. The IF stage of the branch destination instruction is started from the slot containing the third MA of the TRAP instruction.

Instruction Issuance

This instruction uses the memory access pipeline.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
(17) SLEEP Instruction

Instruction Type

SLEEP

Pipeline

<table>
<thead>
<tr>
<th>SLEEP</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>EX</th>
<th>EX</th>
</tr>
</thead>
</table>

Next instruction IF — ...  
Instruction after next IF — ...

Operation

The pipeline ends after seven stages: IF, ID, EX, MA, EX, EX, EX.

After a SLEEP instruction is executed, sleep mode or standby mode is entered.

Instruction Issuance

This instruction uses the memory access pipeline.

Parallel Execution Capability

This is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction.
8.9.7 Exception Handling

(1) Interrupt Exception Handling

Instruction Type
Interrupt exception handling

Pipeline

- No banking

\[
\begin{array}{cccccccc}
\text{Interrupt} & \text{IF} & \text{ID} & \text{EX} & \text{EX} & \text{MA} & \text{MA} & \text{MA} \\
\text{Next instruction} & \text{IF} & \ldots \\
\text{Instruction after next} & \text{IF} & \ldots \\
\text{Branch destination} & \text{IF} & \text{ID} \\
\end{array}
\]

- Banking, no overflow

\[
\begin{array}{cccccccc}
\text{Interrupt} & \text{IF} & \text{ID} & \text{EX} & \text{EX} & \text{MA} & \text{MA} & \text{MA} & \text{MA} \\
\text{Next instruction} & \text{IF} & \ldots \\
\text{Branch destination} & \text{IF} & \text{ID} \\
\end{array}
\]

- Banking and overflow

\[
\begin{array}{cccccccc}
\text{Interrupt} & \text{IF} & \text{ID} & \text{EX} & \text{EX} & \text{MA} & \text{MA} & \text{MA} & \ldots & \text{MA} \\
\text{Next instruction} & \text{IF} & \ldots \\
\text{Branch destination} & \text{IF} & \ldots & \text{ID} \\
\end{array}
\]

Operation

An interrupt is accepted in the ID stage of an instruction, and processing from that ID stage onward is replaced by an exception handling sequence.

Interrupt handling operations are different when there is no banking, when there is banking, and when there is banking and overflow.

When there is no banking, the pipeline ends after seven stages: IF, ID, EX, EX, MA, MA, MA.
When there is banking and no overflow, saving to the bank is performed automatically. The pipeline ends after eight stages: IF, ID, EX, EX, MA, MA, MA, EX.

When there is banking and overflow, registers saved to the bank are automatically restored, and the BO bit is set to 1. The pipeline ends after 27 stages: IF, ID, EX, EX, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA, MA.

After the first two stages there are two repetitions of EX, three repetitions of MA, one EX, and 19 repetitions of MA.

Interrupt exception handling is not a delayed branch. The IF stage of the branch destination instruction is started from the slot containing the third MA stage of the interrupt exception handling.

Interrupt sources comprise external interrupt request pins such as NMI, a user break, and interrupts by on-chip peripheral modules.

**Interrupt Acceptance**

Interrupt exception handling is not accepted in a delay slot.

If a multi-cycle instruction is currently being executed, interrupt exception handling is not accepted until after execution of that instruction is completed. However, a DIVU or DIVS instruction can be canceled during execution, allowing the interrupt to be accepted.
(2) Address Error Exception Handling

Instruction Type

Address error exception handling

Pipeline

<table>
<thead>
<tr>
<th>Address error exception handling</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch destination</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation

An address error is accepted in the ID stage of an instruction, and processing from that ID stage onward is replaced by the address error exception handling sequence.

The pipeline ends after seven stages: IF, ID, EX, EX, MA, MA, MA. Address error exception handling is not a delayed branch. The IF stage of the branch destination instruction is started from the slot containing the last MA stage of the address error exception handling.

Address error generation sources comprise those related to an instruction fetch, and those related to a data read or write. See the hardware manual for details of generation sources.

Address Error Exception Handling Acceptance

Address error exception handling is not accepted in a delay slot.

If a multi-cycle instruction is currently being executed, address error exception handling is not accepted until after execution of that instruction is completed. However, a DIVU or DIVS instruction can be canceled during execution, allowing address error exception handling to be accepted.
(3) Illegal Instruction Exception Handling

**Instruction Type**

Illegal instruction exception handling

**Pipeline**

```
[↔↔↔↔↔↔↔↔↔↔ Slots]
```

<table>
<thead>
<tr>
<th>Illegal instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>⋮</td>
<td>⋮</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>—</td>
<td>⋮</td>
<td>⋮</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch destination</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

An illegal instruction is accepted in the ID stage of an instruction, and processing from that ID stage onward is replaced by the illegal instruction exception handling sequence. The pipeline ends after seven stages: IF, ID, EX, EX, MA, MA, MA. Illegal instruction exception handling is not a delayed branch.

Address error generation sources comprise those related to general illegal instructions and those related to slot illegal instructions. When undefined code located other than in the slot immediately after a delayed branch instruction (called the delay slot) is decoded, general illegal instruction exception handling is performed. When undefined core located in the delay slot is decoded, or an instruction that modifies the program counter, and a 32-bit instruction, and a RESBANK instruction, and a DIVU or DIVS instruction are located in the delay slot and decoded, slot illegal instruction handling is performed.

General illegal instruction exception handling is also performed if an FPU instruction or FPU-related CPU instruction is executed while the FPU is in the module stopped state.

The IF stage of the branch destination instruction is started from the slot containing the last MA stage of the illegal instruction exception handling.
(4) FPU Exception Handling

Instruction Type

FPU exception handling

Pipeline

<table>
<thead>
<tr>
<th>FPU exception handling</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>MA</th>
</tr>
</thead>
</table>

Next instruction    | IF | ⋯  
Instruction after next | IF | ⋯  
Branch destination  | IF | ID  

Operation

An FPU execution is accepted in the ID stage of an instruction, and processing from that ID stage onward is replaced by the FPU exception handling sequence.

The pipeline ends after six stages: IF, ID, EX, MA, MA, MA. FPU exception handling is not a delayed branch. The IF stage of the branch destination instruction is started from the slot containing the last MA stage of the FPU exception handling.

Pipeline Processing of Instructions from Generation to Acceptance of FPU Exceptions

The FPU makes the instruction at which the execution occurred an NOP instruction, and also makes FPU instructions (excluding FCMP instructions) from occurrence of the execution to the instruction that accepts the exception NOP instructions. Consequently, FPU registers are not updated by instructions during this interval.

With FPU-related CPU instructions, as above, FPU registers are not updated (NOP operation is performed), but CPU registers are updated.

CPU instructions are not made NOP instructions, and operate as usual.
8.9.8 Floating-Point Instructions and FPU-Related CPU Instructions

(1) FPUL Load Instructions

Instruction Types

LDS  Rm, FPUL
LDS.L  @Rm+, FPUL

Pipeline

\[
\begin{array}{cccccc|c}
\text{Instruction A} & \text{IF} & \text{ID} & \text{EX} & \text{MA} & \text{Slots} \\
& \text{IF} & \text{DF} & \text{EX} & \text{NA} & \text{SF} & : \text{CPU pipeline} \\
\hline
\text{Next instruction} & \text{IF} & \text{ID} & \text{EX} & \ldots & : \text{CPU pipeline} \\
& \text{IF} & \text{DF} & \ldots & : \text{FPU pipeline} \\
\text{Instruction after next} & \text{IF} & \text{ID} & \text{EX} & \ldots & : \text{CPU pipeline} \\
& \text{IF} & \text{DF} & \ldots & : \text{FPU pipeline} \\
\end{array}
\]

Operation

The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after five stages – IF, DF, EX, NA, SF. Contention may occur if an instruction that reads FPUL is located within the 3 instructions following one of these instructions.

Instruction Issuance

These instructions use the FPU load/store pipeline and memory access pipeline. There is no contention between an LDS instruction and a CPU memory read instruction.

Parallel Execution Capability

No particular comments
(2) FPSCR Load Instructions

**Instruction Types**

LDS Rm, FPSCR

LDS.L @Rm+, FPSCR

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>CPU pipeline</th>
<th>IF</th>
<th>DF</th>
<th>EX</th>
<th>NA</th>
<th>SF</th>
<th>FPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next instruction</td>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>⋮</td>
<td>CPU pipeline</td>
<td>IF</td>
<td>—</td>
<td>DF</td>
<td>⋮</td>
<td>FPU pipeline</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>⋮</td>
<td>CPU pipeline</td>
<td>IF</td>
<td>DF</td>
<td>⋮</td>
<td>FPU pipeline</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after five stages – IF, DF, EX, NA, SF. A subsequent FPU-related instruction is stalled for the next 3 cycles.

**Instruction Issuance**

These instructions use the FPU load/store pipeline.

The LDS.L instruction also uses the memory access pipeline.

If an FPU arithmetic operation instruction is still performing calculation, these instructions are kept waiting until that instruction ends.

**Parallel Execution Capability**

These instructions cannot be executed in parallel with FPU instructions or FPU-related CPU instructions.
(3) FPUL Store Instruction (STS)

Instruction Type

STS   FPUL, Rn

Pipeline

```
                    ↔↔↔↔↔↔↔  Slots
Instruction A     IF  ID  EX  WB      : CPU pipeline
                   IF  DF  EX  NA      : FPU pipeline
Next instruction  IF  ID  EX  ...    : CPU pipeline
                   IF  DF  ...       : FPU pipeline
Instruction after next IF  ID  EX  ...  : CPU pipeline
                  IF  DF  ...      : FPU pipeline
```

Operation

The CPU pipeline ends after four stages – IF, ID, EX, WB – and the FPU pipeline after four stages – IF, DF, EX, NA. Contention may occur if an instruction that uses the destination of this instruction is located within the 3 instructions following this instruction.

Instruction Issuance

This instruction uses the multiplication result read path.

This instruction uses the FPU load/store pipeline and memory access pipeline.

There is no contention with a CPU memory write instruction.

If FPUL is waiting for the result of an FPU arithmetic operation, the latency of the previous instruction is reduced by 2. See section 8.6, Contention Due to FPU, for details.

Parallel Execution Capability

No particular comments
(4) FPUL Store Instruction (STS.L)

Instruction Type

STS.L FPUL,@-Rn

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>...</td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction after next</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>...</td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

Operation

The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after four stages – IF, DF, EX, NA.

Instruction Issuance

This instruction uses the FPU load/store pipeline and memory access pipeline.

If FPUL is waiting for the result of an FPU arithmetic operation, the latency of the previous instruction is reduced by 1. See section 8.6, Contention Due to FPU, for details.

Parallel Execution Capability

No particular comments
(5) FPSCR Store Instruction (STS)

**Instruction Type**

STS FPSCR, Rn

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>—</td>
<td>ID</td>
<td>EX</td>
<td>...</td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction after next</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DF</td>
<td>...</td>
<td></td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

**Operation**

The CPU pipeline ends after four stages – IF, ID, EX, MA, WB – and the FPU pipeline after four stages – IF, DF, EX, NA.

Contention may occur if an instruction that uses the destination of this instruction is located within the 3 instructions following this instruction.

**Instruction Issuance**

This instruction uses the multiplication result read path.

If an FPU arithmetic operation instruction is still performing calculation, this instruction is kept waiting until that instruction ends.

**Parallel Execution Capability**

This instruction cannot be executed in parallel with FPU instructions or FPU-related CPU instructions.
(6) FPSCR Store Instruction (STS.L)

**Instruction Type**

STS.L   FPSCR,@-Rn

**Pipeline**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next instruction</th>
<th>IF</th>
<th>—</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td>—</td>
<td>DF</td>
<td>...</td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction after next</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>...</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td>DF</td>
<td>...</td>
<td></td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

**Operation**

The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after four stages – IF, DF, EX, NA.

**Instruction Issuance**

This instruction uses the FPU load/store pipeline and memory access pipeline.

If an FPU arithmetic operation instruction is still performing calculation, this instruction is kept waiting until that instruction ends.

**Parallel Execution Capability**

This instruction cannot be executed in parallel with FPU instructions or FPU-related CPU instructions.
Some floating-point register-register transfer instructions, floating-point register-immediate instructions, and floating-point operation instructions

### Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLDS</td>
<td>FRm, FPUL</td>
</tr>
<tr>
<td>FMOV</td>
<td>FRm, FRn</td>
</tr>
<tr>
<td>FSTS</td>
<td>FPUL, FRn</td>
</tr>
<tr>
<td>FLDI0</td>
<td>FRn</td>
</tr>
<tr>
<td>FLDI1</td>
<td>FRn</td>
</tr>
<tr>
<td>FABS</td>
<td>FRn</td>
</tr>
<tr>
<td>FNEG</td>
<td>FRn</td>
</tr>
<tr>
<td>FABS</td>
<td>DRn</td>
</tr>
<tr>
<td>FNEG</td>
<td>DRn</td>
</tr>
</tbody>
</table>

### Pipeline

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CPU pipeline</th>
<th>FPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction A</td>
<td>IF ID EX</td>
<td>IF DF EX NA SF</td>
</tr>
<tr>
<td>Next instruction</td>
<td>IF ID EX ⋯</td>
<td>IF DF E1 E2 SF</td>
</tr>
<tr>
<td>Instruction after next</td>
<td>IF ID EX ⋯</td>
<td>IF DF E1 E2 E3</td>
</tr>
</tbody>
</table>

### Operation

The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after five stages – IF, DF, EX, NA, SF. Contention does not occur even if one of these instructions is immediately followed by an instruction that reads the destination of that instruction.

### Instruction Issuance

These instructions use the FPU load/store pipeline.

### Parallel Execution Capability

These are zero-latency instructions. Parallel execution is possible even if one of these instructions is executed as a preceding instruction and the succeeding instruction uses FRn, FPUL.
(8) Double-Precision Floating-Point Register to Register Data Transfer Instructions

**Instruction Types**

\[ \text{FMOV DRm, DRn} \]

**Pipeline**

![Pipeline Diagram]

**Operation**

The CPU pipeline ends after four stages – IF, ID, EX, EX – and the FPU pipeline after six stages – IF, DF, EX, EX, NA, SF. Contention does not occur even if one of these instructions is immediately followed by an instruction that reads the destination of that instruction.

**Instruction Issuance**

This instruction uses the FPU load/store pipeline.

**Parallel Execution Capability**

No particular comments
(9) FSCHG Instruction

Instruction Types

FSCHG

Pipeline

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
</tr>
</tbody>
</table>

| Next instruction | IF | ID | EX | ... | : CPU pipeline |
|-------------------|----|----|----|-----|------|------|------|
|                   | IF | DF | E1 | E2  | SF   | : FPU pipeline |

| Instruction after next | IF | ID | EX | ... | : CPU pipeline |
|------------------------|----|----|----|-----|------|------|------|
|                        | IF | DF | E1 | E2  | SF   | : FPU pipeline |

Operation

The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after five stages – IF, DF, EX, NA, SF. Contention does not occur even if one of these instructions is immediately followed by an instruction that reads the destination of that instruction.

Instruction Issuance

This instruction uses the FPU load/store pipeline.

Parallel Execution Capability

No particular comments
(10) Floating-Point Register Load Instructions

Instruction Types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOV.S @Rm, FRn</td>
<td>Single-Precision Single-Precision</td>
</tr>
<tr>
<td>FMOV.S @Rm+, FRn</td>
<td>Single-Precision Single-Precision</td>
</tr>
<tr>
<td>FMOV.S @(R0,Rm), FRn</td>
<td>Single-Precision Single-Precision</td>
</tr>
<tr>
<td>FMOV.D @Rm, DRn</td>
<td>Double-Precision Double-Precision</td>
</tr>
<tr>
<td>FMOV.D @Rm, DRn</td>
<td>Double-Precision Double-Precision</td>
</tr>
<tr>
<td>FMOV.D @(R0,Rm), DRn</td>
<td>Double-Precision Double-Precision</td>
</tr>
</tbody>
</table>

Pipeline

- Single-Precision

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF ID EX MA</td>
<td>CPU pipeline</td>
</tr>
<tr>
<td>IF DF EX NA SF</td>
<td>FPU pipeline</td>
</tr>
</tbody>
</table>

- Double-Precision

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF ID EX MA</td>
<td>CPU pipeline</td>
</tr>
<tr>
<td>IF DF EX NA SF</td>
<td>FPU pipeline</td>
</tr>
</tbody>
</table>
Operation

- Single-Precision
  The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after five stages – IF, DF, EX, NA, SF. Contention may occur if an instruction that reads the destination of one of these instructions is located within the 3 instructions following that instruction.

- Double-Precision
  The CPU pipeline ends after five stages – IF, ID, EX, MA, MA – and the FPU pipeline after six stages – IF, DF, EX, EX, NA, SF. Contention may occur if an instruction that reads the destination of one of these instructions is located within the 5 instructions following that instruction.

Instruction Issuance

These instructions use the FPU load/store pipeline and memory access pipeline.

Parallel Execution Capability

FMOV.D instruction is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(11) Floating-Point Register Load Instruction (12-Bit Displacement)

**Instruction Type**

- **FMOV.S** @(disp12,Rm),FRn
- **FMOV.D** @(disp12,Rm),DRn

**Pipeline**

- **Single-Precision**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>SF</td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

  Next instruction
  | IF | ID | EX | ... | : CPU pipeline |
  | IF | DF | EX | NA | SF | : FPU pipeline |

  Instruction after next
  | IF | ID | EX | ... | : CPU pipeline |
  | IF | DF | E1 | E2 | SF | : FPU pipeline |

- **Double-Precision**

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>SF</td>
<td>: FPU pipeline</td>
<td></td>
</tr>
</tbody>
</table>

  Next instruction
  | IF | — | ID | EX | ... | : CPU pipeline |
  | IF | — | DF | E1 | E2 | SF | : FPU pipeline |

  Instruction after next
  | IF | — | ID | EX | ... | : CPU pipeline |
  | IF | — | DF | E1 | E2 | SF | : FPU pipeline |

**Operation**

- **Single-Precision**

  The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after five stages – IF, DF, EX, NA, SF. Contention may occur if an instruction that reads the destination of this instruction is located within the 3 instructions following this instruction.

- **Double-Precision**

  The CPU pipeline ends after five stages – IF, ID, EX, MA, MA – and the FPU pipeline after six stages – IF, DF, EX, EX, NA, SF. Contention may occur if an instruction that reads the destination of this instruction is located within the 3 instructions following this instruction.
Instruction Issuance

These instructions use the FPU load/store pipeline and memory access pipeline.

Parallel Execution Capability

This is a 32-bit instruction, and cannot be used in parallel execution. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction.)
(12) Floating-Point Register Store Instructions

Instruction Types

- **FMOV.S** FRm,@Rn
- **FMOV.S** FRm,@-Rn
- **FMOV.S** FRm,@(R0,Rn)
- **FMOV.D** DRm,@Rn
- **FMOV.D** DRm,@-Rn
- **FMOV.D** DRm,@(R0,Rn)

Pipeline

- **Single-Precision**

  ```
  ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slots
  
  Instruction A
  IF  ID  EX  MA : CPU pipeline
  IF  DF  EX  NA : FPU pipeline
  
  Next instruction
  IF  EX  ... : CPU pipeline
  IF  E1  E2  SF : FPU pipeline
  
  Instruction after next
  IF  ID  EX  ... : CPU pipeline
  IF  DF  E1  E2  SF : FPU pipeline
  ```

- **Double-Precision**

  ```
  ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slots
  
  Instruction A
  IF  ID  EX  MA  MA : CPU pipeline
  IF  DF  EX  EX  NA : FPU pipeline
  
  Next instruction
  IF  --  ID  EX  ... : CPU pipeline
  IF  --  DF  E1  E2  SF : FPU pipeline
  
  Instruction after next
  IF  --  ID  EX  ... : CPU pipeline
  IF  --  DF  E1  E2  SF : FPU pipeline
  ```
Operation

- Single-Precision
  The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after four stages – IF, DF, EX, NA.

- Double-Precision
  The CPU pipeline ends after five stages – IF, ID, EX, MA, MA – and the FPU pipeline after five stages – IF, DF, EX, EX, NA.

Instruction Issuance

These instructions use the FPU load/store pipeline and memory access pipeline.

Parallel Execution Capability

FMOV.D instruction is a multi-cycle instruction, and cannot be executed in parallel with a subsequent instruction. (See section 8.3.4, Details of Contention Due to Multi-Cycle Instruction.)
(13) Floating-Point Register Store Instruction (12-Bit Displacement)

Instruction Type

FMOV.S  FRm,@(disp12,Rn)  
FMOV.D  DRm,@(disp12,Rn)

Pipeline

- Single-Precision

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>FPU pipeline</td>
</tr>
</tbody>
</table>

Next instruction

| IF | ID | EX | ... | CPU pipeline |
| IF | DF | E1 | E2 | SF | FPU pipeline |

Instruction after next

| IF | ID | EX | ... | CPU pipeline |
| IF | DF | EX | NA | SF | FPU pipeline |

- Double-Precision

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>MA</th>
<th>CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>EX</td>
<td>NA</td>
<td>FPU pipeline</td>
</tr>
</tbody>
</table>

Next instruction

| IF | —  | ID | EX | ... | CPU pipeline |
| IF | —  | DF | E1 | E2 | SF | FPU pipeline |

Instruction after next

| IF | —  | ID | EX | ... | CPU pipeline |
| IF | —  | DF | E1 | E2 | SF | FPU pipeline |

Operation

- Single-Precision

The CPU pipeline ends after four stages – IF, ID, EX, MA – and the FPU pipeline after four stages – IF, DF, EX, NA.

- Double-Precision

The CPU pipeline ends after five stages – IF, ID, EX, MA, MA – and the FPU pipeline after five stages – IF, DF, EX, EX, NA.
Instruction Issuance

These instructions use the FPU load/store pipeline and memory access pipeline.

Parallel Execution Capability

This is a 32-bit instruction, and cannot be used in parallel execution. (See section 8.3.5, Details of Contention Due to 32-Bit Instruction.)
(14) Floating-Point Operation Instructions (Excluding FDIV, FSQRT, FLOAT, and FTRC)

### Instruction Types

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD FRm, FRn</td>
<td></td>
</tr>
<tr>
<td>FMAC FR0, FRm, FRn</td>
<td></td>
</tr>
<tr>
<td>FMUL FRm, FRn</td>
<td></td>
</tr>
<tr>
<td>FSUB FRm, FRn</td>
<td></td>
</tr>
<tr>
<td>FADD DRm, DRn</td>
<td></td>
</tr>
<tr>
<td>FMUL DRm, DRn</td>
<td></td>
</tr>
<tr>
<td>FSUB DRm, DRn</td>
<td></td>
</tr>
</tbody>
</table>

### Pipeline

- **Single-Precision**

  ![Single-Precision Pipeline Diagram](image)

- **Double-Precision**

  ![Double-Precision Pipeline Diagram](image)
Operation

- Single-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after five stages – IF, DF, E1, E2, SF. Contention may occur if an instruction that reads the destination of one of these instructions is located within the 5 instructions following that instruction.

- Double-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after 10 stages – IF, DF, E1, E1, E1, E1, E1, E1, E2, SF. Contention may occur if an instruction that reads the destination of one of these instructions is located within the 15 instructions following that instruction.

Instruction Issuance

These instructions use the FPU arithmetic operation pipeline. See section 8.6, Contention Due to FPU, for details of contention.

Parallel Execution Capability

No particular comments
(15) Floating-Point Operation Instructions (FLOAT, FTRC) and FCNVSD, FCNVDS Instructions

**Instruction Types**

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT FPUL, FRn</td>
<td></td>
</tr>
<tr>
<td>FTRC DRm, FPUL</td>
<td></td>
</tr>
<tr>
<td>FLOAT FPUL, DRn</td>
<td></td>
</tr>
<tr>
<td>FTRC FRm, FPUL</td>
<td></td>
</tr>
<tr>
<td>FCNVSD</td>
<td></td>
</tr>
<tr>
<td>FCNVDS</td>
<td></td>
</tr>
</tbody>
</table>

**Pipeline**

- **Single-Precision**

  From left to right, there are 8 slots in the pipeline. The first 4 slots are for the CPU pipeline, and the remaining 4 slots are for the FPU pipeline.

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>E1</td>
<td>E2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction after next</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>E1</td>
<td>E2</td>
</tr>
</tbody>
</table>

- **Double-Precision**

  From left to right, there are 16 slots in the pipeline. The first 4 slots are for the CPU pipeline, and the remaining 12 slots are for the FPU pipeline.

<table>
<thead>
<tr>
<th>Instruction A</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>E1</td>
<td>E1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Next instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
<td>SF</td>
<td>: FPU pipeline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction after next</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>: CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>DF</td>
<td>EX</td>
<td>NA</td>
</tr>
</tbody>
</table>
Operation

- Single-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after five stages – IF, DF, E1, E2, SF. Contention may occur if an instruction that reads the destination of one of these instructions is located within the 5 instructions following that instruction.

- Double-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after six stages – IF, DF, E1, E1, E2, SF. Contention may occur if an instruction that reads the destination of one of these instructions is located within the 7 instructions following that instruction.

Instruction Issuance

These instructions use the FPU arithmetic operation pipeline. See section 8.6, Contention Due to FPU, for details of contention.

Parallel Execution Capability

No particular comments
(16) Floating-Point Operation Instructions (FDIV)

**Instruction Types**

- **FDIV** FRm, FRn
- **FDIV** DRm, DRn

**Pipeline**

- **Single-Precision**

  | Instruction A | IF ID EX | : CPU pipeline |
  | IF DF E1 ED ED ED ED ED ED ED E1 E2 SF | : FPU pipeline |

  Next instruction
  | IF ID EX ... | : CPU pipeline |
  | IF DF EX NA SF | : FPU pipeline |

  Instruction after next
  | IF ID EX ... | : CPU pipeline |
  | IF DF EX NA SF | : FPU pipeline |

- **Double-Precision**

  | Instruction A | IF ID EX | : CPU pipeline |
  | IF DF E1 E1 ED ... ED E1 E1 E1 E2 SF | : FPU pipeline |

  Next instruction
  | IF ID EX ... | : CPU pipeline |
  | IF DF EX NA SF | : FPU pipeline |

  Instruction after next
  | IF ID EX ... | : CPU pipeline |
  | IF DF EX NA SF | : FPU pipeline |
Operation

- Single-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after 14 stages – IF, DF, E1, ED, ED, ED, ED, ED, ED, E1, E2, SF. That is to say, after one E1 stage has been performed, the ED stage is repeated 8 times, followed by E1, E2, and SF.

- Double-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after 27 stages – IF, DF, E1, E1, ED, ED, ED, ED, ED, ED, ED, ED, ED, ED, ED, E1, E1, E1, E2, SF. That is to say, after the E1 stage has been performed twice, the ED stage is repeated 18 times, followed by E1, E1, E1, E2, and SF.

The contention described in section 8.6, Contention Due to FPU, occurs. If there is an overlapping instruction that accesses the FDIV result register in the FDIV pipeline, that instruction is kept waiting until execution of the FDIV instruction is finished. Stages from E1 onward are stalled until the end of FDIV execution, and subsequent instructions are also subject to stalling. Therefore, if a floating-point instruction that uses the FDIV result register, or an FPU-related CPU instruction, is not located within 21 instructions immediately after the FDIV instruction in the case of single-precision, or 49 instructions in the case of double-precision, a CPU instruction or another FPU instruction can be executed during that interval, enabling performance to be improved.

Instruction Issuance

These instructions use the FPU arithmetic operation pipeline. See section 8.6, Contention Due to FPU, for details of contention.

The ED stages of these instructions operate in states, without regard to slots.

Parallel Execution Capability

No particular comments
(17) Floating-Point Operation Instructions (FSQRT)

Instruction Types

FSQRT FRn
FSQRT DRn

Pipeline

• Single-Precision

        ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slots

Instruction A  IF  ID  EX  : CPU pipeline
                IF  DF  E1  ED  ED  ED  ED  ED  ED  E1  E2  SF  : FPU pipeline

Next instruction  IF  ID  EX  ...  : CPU pipeline
                IF  DF  EX  NA  SF  : FPU pipeline

Instruction after next  IF  ID  EX  ...  : CPU pipeline
                IF  DF  EX  NA  SF  : FPU pipeline

• Double-Precision

        ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ ↔ Slots

Instruction A  IF  ID  EX  : CPU pipeline
                IF  DF  E1  E1  ED  ...  ED  E1  E1  E1  E2  SF  : FPU pipeline

Next instruction  IF  ID  EX  ...  : CPU pipeline
                IF  DF  EX  NA  SF  : FPU pipeline

Instruction after next  IF  ID  EX  ...  : CPU pipeline
                IF  DF  EX  NA  SF  : FPU pipeline
Operation

• Single-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after 13 stages – IF, DF, E1, ED, ED, ED, ED, E1, E2, SF. That is to say, after one E1 stage has been performed, the ED stage is repeated 7 times, followed by E1, E2, and SF.

• Double-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after 26 stages – IF, DF, E1, E1, ED, ED, ED, ED, ED, ED, ED, ED, ED, ED, E1, E1, E1, E1, E1, SF. That is to say, after the E1 stage has been performed twice, the ED stage is repeated 17 times, followed by E1, E1, E1, E1, and SF.

The contention described in section 8.6, Contention Due to FPU, occurs. If there is an overlapping instruction that accesses the FSQRT result register in the FSQRT pipeline, that instruction is kept waiting until execution of the FSQRT instruction is finished. Stages from E1 onward are stalled until the end of FSQRT execution, and subsequent instructions are also subject to stalling. Therefore, if a floating-point instruction that uses the FSQRT result register, or an FPU-related CPU instruction, is not located within 19 instructions immediately after the FSQRT instruction in the case of single-precision, or 47 instructions in the case of double-precision, a CPU instruction or another FPU instruction can be executed during that interval, enabling performance to be improved.

Instruction Issuance

These instructions use the FPU arithmetic operation pipeline. See section 8.6, Contention Due to FPU, for details of contention.

The ED stages of these instructions operate in states, without regard to slots.

Parallel Execution Capability

No particular comments
(18) Floating-Point Compare Instructions

Instruction Types

| FCMP/EQ  | FRm, FRn |
| FCMP/GT  | FRm, FRn |
| FCMP/EQ  | DRm, DRn |
| FCMP/GT  | DRm, DRn |

Pipeline

• Single-Precision

| Instruction A | IF | ID | EX | : CPU pipeline |
| IF | DF | E1 | E2 | : FPU pipeline |

Next instruction: IF | ID | EX | ... | : CPU pipeline

| Instruction after next | IF | ID | EX | ... | : CPU pipeline |
| IF | DF | E1 | E2 | SF | : FPU pipeline |

• Double-Precision

| Instruction A | IF | ID | EX | EX | : CPU pipeline |
| IF | DF | E1 | E1 | E2 | : FPU pipeline |

Next instruction: IF | — | ID | EX | ... | : CPU pipeline

| Instruction after next | IF | — | ID | EX | ... | : CPU pipeline |
| IF | — | DF | EX | NA | SF | : FPU pipeline |
Operation

• Single-Precision
  The CPU pipeline ends after three stages – IF, ID, EX – and the FPU pipeline after four stages
  – IF, DF, E1, E2. As the T bit is checked in E2, an instruction that references the T bit
  immediately afterward is stalled for 2 cycles.

  FCMP  IF ID EX : CPU pipeline
         IF DF E1 E2 : FPU pipeline
  BT    IF — — ID EX : CPU pipeline
         IF — — DF ... : FPU pipeline

Operation

• Double-Precision
  The CPU pipeline ends after four stages – IF, ID, EX, EX – and the FPU pipeline after five
  stages – IF, DF, E1, E1, E2. As the T bit is checked in E2, an instruction that references the T
  bit immediately afterward is stalled for 3 cycles.

  FCMP  IF ID EX : CPU pipeline
         IF DF E1 E1 E2 : FPU pipeline
  BT    IF — — — ID EX : CPU pipeline
         IF — — — DF ... : FPU pipeline

Instruction Issuance

These instructions use the FPU arithmetic operation pipeline.

Parallel Execution Capability

Parallel execution of a double-precision FCMP instruction and the following instruction is not
possible.
8.10 Simple Method of Calculating Required Number of Clock Cycles

A simple method of calculating required number of clock cycles is described below. This method provides a rough approximation, but it allows the user to calculate the number of clock cycles needed to execute the target instruction string.

The calculation is based on the following rules.

(1) The instructions are assumed to already have been fetched, so fetch time is not taken into consideration.

(2) The 32-bit instructions operate in “execution state” cycles.

(3) If resource contention occurs, the previously issued instructions operate in “execution state” cycles. Parallel execution of subsequent instructions is not possible.

(4) If the result from the previously issued instruction is used by the instruction that immediately follows, the calculation assumes that the previously issued instruction will require “latency” cycles.

(5) If the result from the previously issued instruction is not used by the instruction that immediately follows, the calculation assumes that the previously issued instruction will require “execution state” cycles.

(6) Correction for parallel execution is performed in simplified form as a compensation item.

There are a large number of exceptional cases, so the calculation method introduced here cannot be 100% accurate. It does allow the user to obtain a rough idea of the number of clock cycles that will be required, however. Examples are provided below.

1. Counting Latency Cycles

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.L @ R1, R0</td>
<td></td>
<td>[EX]</td>
<td>MA</td>
<td>WB</td>
<td>2</td>
</tr>
<tr>
<td>ADD # imm, R0</td>
<td></td>
<td>—</td>
<td>—</td>
<td>[EX]</td>
<td>1</td>
</tr>
<tr>
<td>MOV.L R0, @ R1</td>
<td></td>
<td>—</td>
<td>[ID]</td>
<td>[EX]</td>
<td>MA</td>
</tr>
</tbody>
</table>

The result from MOV.L, which precedes ADD, will be used, so the calculation assumes that MOV.L will require “latency” cycles (two cycles) to execute. The next MOV.L instruction uses the result from ADD, so the calculation assumes that the ADD instruction will require “latency” execution (one cycle).
2. Counting Execution State Cycles

In this case, the result from the previously issued instruction is not used by the instructions that follow it, so the instructions execute in parallel provided no resource contention occurs. The number of cycles required by each instruction to execute are calculated in the “execution state.” When the preceding instruction uses one execution state cycle, the following instruction executes in parallel. When parallel execution takes place, the number of cycles required by the preceding instruction is calculated as “execution state” minus one. This serves as a simplified compensation. (This compensation appears as the final item in the equation introduced below.)

3. If Resource Contention Occurs

If resource contention occurs, parallel execution is not possible. The execution of each instruction requires “execution state” cycles.

4. Instructions Using More Than One Execution State

For instructions using more than one execution state, the calculation assumes that the number of remaining states is reduced one by one until only one remains, at which point parallel execution with the subsequent instructions is possible. In this case, the number of cycles required for execution is calculated as “execution state” minus one if parallel execution with subsequent instructions takes place, and as “execution state” if no parallel execution takes place. This serves
as a simplified compensation. (This compensation appears as the final item in the equation introduced below.)

Based on the above, the number of cycles necessary to execute the entire instruction string is as summarized below, in extremely simplified terms. If some portions of the string have dependencies and others do not, separate calculations should be made for each portion and the results added together.

- **If Dependencies Exist Between Instructions**
  Required number of cycles = sum total of “latency” cycles of all instructions

- **If No Dependencies Exist Between Instructions**
  Required number of cycles = sum total of “execution state” cycles of all instructions – (total number of instructions – number of instructions that cannot be executed in parallel) ÷ 2

In this case, “number of instructions that cannot be executed in parallel” is the total number of instructions that cannot be executed in parallel due to resource contention (in particular, memory access instructions that immediately follow another memory access instruction), instructions using more than one execution state, and 32-bit instructions.

The final item compensates for the effects of parallel execution by reducing the number of required cycles for the preceding instructions.

**Example: If Dependencies Exist Between Instructions**

```
BAND.B
ROTCL
BAND.B
ROTCL
```

The “latency” cycles for all instructions are added together, producing a total of eight cycles.

**Example: If No Dependencies Exist Between Instructions**

```
ADD # imm, R0
BAND.B # imm, @(disp12,R2)
MULR R4, R0
ROTCL R5
```

Required number of cycles = \(1 + 3 + 2 + 1 - (4 - 2) ÷ 2\)

\[= 7 - 1 = 6 \text{ cycles}\]
Appendix A  SH-2A/SH2A-FPU Parallel Execution

The table below can be used to determine whether or not parallel execution is supported, depending on the type of arithmetic unit used. In the case of instructions that belong to more than one category, parallel execution is supported if all of the applicable intersections are marked with a circle (o).

<table>
<thead>
<tr>
<th>First instruction</th>
<th>Second instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) BR</td>
<td>(2) MR</td>
</tr>
<tr>
<td>(1) BR</td>
<td>×</td>
</tr>
<tr>
<td>(2) MR</td>
<td>0</td>
</tr>
<tr>
<td>(3) MW</td>
<td>0</td>
</tr>
<tr>
<td>(4) MF</td>
<td>0</td>
</tr>
<tr>
<td>(5) ML</td>
<td>0</td>
</tr>
<tr>
<td>(6) MU</td>
<td>0</td>
</tr>
<tr>
<td>(7) SF</td>
<td>0</td>
</tr>
<tr>
<td>(8) FL</td>
<td>0</td>
</tr>
<tr>
<td>(9) FP</td>
<td>0</td>
</tr>
<tr>
<td>(10) FC</td>
<td>×</td>
</tr>
<tr>
<td>(11) EX</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classification of First Instruction</th>
<th>Classification of Second Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>BR</td>
<td>BF disp</td>
</tr>
<tr>
<td></td>
<td>BT/S disp</td>
<td>BSR disp</td>
</tr>
<tr>
<td></td>
<td>BRA disp</td>
<td>BRAF Rm</td>
</tr>
<tr>
<td></td>
<td>JSR @Rm</td>
<td>JSRN @Rm</td>
</tr>
<tr>
<td></td>
<td>RTS/N</td>
<td>RTV/N Rn</td>
</tr>
<tr>
<td>MR</td>
<td>MR</td>
<td>LDC.L @Rm+,GBR</td>
</tr>
<tr>
<td></td>
<td>MOV.B @((disp,GBR),R0)</td>
<td>MOV.B @((disp,Rm),R0)</td>
</tr>
<tr>
<td></td>
<td>MOV.B @Rm,Rn</td>
<td>MOV.B @Rm+,Rn</td>
</tr>
<tr>
<td></td>
<td>MOV.B @((disp12,Rm),Rn)</td>
<td>MOV.W @((disp,GBR),R0)</td>
</tr>
<tr>
<td></td>
<td>MOV.W @((R0,Rm),Rn)</td>
<td>MOV.W @Rm,Rn</td>
</tr>
<tr>
<td></td>
<td>MOV.W @Rm,R0</td>
<td>MOV.W @((disp12,Rm),Rn)</td>
</tr>
<tr>
<td></td>
<td>MOV.L @((disp,GBR),R0)</td>
<td>MOV.L @((disp,Rm),Rn)</td>
</tr>
<tr>
<td></td>
<td>MOV.L @Rm,Rn</td>
<td>MOV.L @Rm+,Rn</td>
</tr>
<tr>
<td></td>
<td>MOV.L @((disp12,Rm),Rn)</td>
<td>MOV.L @((disp,PC),Rn)</td>
</tr>
<tr>
<td></td>
<td>MOVU.W @((disp12,Rm),Rn)</td>
<td>MOVML.L @R15+,Rn</td>
</tr>
<tr>
<td></td>
<td>PREF @Rn</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix A  SH-2A/SH2A-FPU Parallel Execution

<table>
<thead>
<tr>
<th>Classifi-</th>
<th>Classifi-</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>cation of First Instruction</td>
<td>cation of Second Instruction</td>
<td></td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>AND.B #imm,@(R0,GBR)</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>BCLR.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>BSET.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>BST.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>OR.B #imm,@(R0,GBR)</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>STC.L SR,=@-Rn</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>TAS.B @Rn</td>
</tr>
<tr>
<td>MW</td>
<td>MR</td>
<td>XOR.B #imm,@(R0,GBR)</td>
</tr>
<tr>
<td>MW</td>
<td>MW</td>
<td>MOV.B R0,@(disp,GBR)</td>
</tr>
<tr>
<td>MW</td>
<td>MW</td>
<td>MOV.B R0,@(disp,Rn)</td>
</tr>
<tr>
<td>MW</td>
<td>MW</td>
<td>MOV.B Rm,=@Rn</td>
</tr>
<tr>
<td>MW</td>
<td>MU</td>
<td>CLR.MAC</td>
</tr>
<tr>
<td>ML</td>
<td>ML</td>
<td>DIV.U R0,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>EXTS.B Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>EXTS.W Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>EXTU.B Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>EXTU.W Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>ROTLC Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>ROTCR Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHAD Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLD Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLL Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLL2 Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLL8 Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLL16 Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLR Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLR16 Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLR2 Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SHLR8 Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SWAP.B Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>SWAP.W Rm,Rn</td>
</tr>
<tr>
<td>SF</td>
<td>SF</td>
<td>XTRCT Rm,Rn</td>
</tr>
<tr>
<td>ML</td>
<td>ML</td>
<td>MLS.MAC</td>
</tr>
<tr>
<td>ML</td>
<td>ML</td>
<td>MLS.MAC</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FABS DRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FABS FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FLD1 FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FLDS FRm,FPUL</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FMUL FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FMUL FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FNEG FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FNEG FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FMOV FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FMOV FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FADD DRm,DRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FADD FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FDIV DRm,DRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FDIV FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FLD1 FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FLDS FRm,FPUL</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FMUL FRm,FRn</td>
</tr>
<tr>
<td>FL</td>
<td>FL</td>
<td>FSQRT FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FADD DRm,DRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FCMP/GT DRm,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FCMPI/GT FRm,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FDIV DRm,DRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FDIV FRm,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FLOAT FPUL,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FLT5 FPUL,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FML FPm,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FSQRT FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FSUB DRm,DRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FSUB FRm,FRn</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FTRC DRm,FPUL</td>
</tr>
<tr>
<td>FP</td>
<td>FP</td>
<td>FTRC FRm,FPUL</td>
</tr>
</tbody>
</table>
### Classification of First Instruction | Classification of Second Instruction | Instruction
--- | --- | ---
FC | FC | FCMP/EQ DRm,DRn FCMP/GT DRm,DRn
ML.FC | ML.FC | STS FPSCR,Rn
EX | EX | ADD #imm,Rn ADD Rm,Rn ADDC Rm,Rn
 | | ADDV Rm,Rn AND #imm,Rn AND Rm,Rn
 | | BCLR #imm3,Rn BLD #imm3,Rn BSET #imm3,Rn
 | | BST #imm3,Rn CLRT CMP/EQ #imm,R0
 | | CMP/EQ Rm,Rn CMP/GE Rm,Rn CMP/GT Rm,Rn
 | | CMP/HI Rm,Rn CMP/HS Rm,Rn CMP/PL Rn
 | | CMP/PZ Rn CMP/STR Rm,Rn CLIPS.B Rn
 | | CLIPS.W Rn CLIPU.B Rn CLIPU.W Rn
 | | DIVS Rm,Rn DIVU DIVS R0,Rn
 | | DIV1 Rm,Rn DT Rn LDC Rm,GBR
 | | LDC Rm,SR LDC Rm,TBR LDC Rm,VBR
 | | LDS Rm,PR LDBANK @Rm,Rn MOV #imm,Rn
 | | MOV Rm,Rn MOVA @(disp,PC),R0 MOV120 #imm20,Rn
 | | MOV120S #imm20,Rn MOVT Rn MOVRT Rn
 | | NEG Rm,Rn NEG Rm,Rn NOP
 | | NOT Rm,Rn NOTT OR #imm,R0
 | | OR Rm,Rn SETT STC GBR,Rn
 | | STC SR,Rn STC TBR,Rn STC VBR,Rn
 | | STS PR,Rn STBANK R0,@Rn SUB Rm,Rn
 | | SUBC Rm,Rn SUBV Rm,Rn TST #imm,R0
 | | TST Rm,Rn XOR #imm,R0 XOR Rm,Rn
 | | RESBANK(BO==0)
MR.MU | MR.MU | LDS.L @Rm+,MACH LDS.L @Rm+,MACL
MW.ML | MW.ML | STS.L MACH,@-Rn STS.L MACL,@-Rn
MW.FL | MW.FL | FMOV.S @(R0,Rm),FRn FMOV.S @Rm,FRn FMOV.S @Rn+,FRn
 | | FMOV.S @(disp12,Rm),FRn FMOV.S FRm,(@R0,Rn) FMOV.S FRm,@-Rn
 | | FMOV.S FRm,FRn FMOV.S FRm,(disp12,Rn) FMOV.D @Rn+,DRn
 | | FMOV.D FRm,FRn FMOV.D FRm,(disp12,Rn) FMOV.D FRm,@-Rn
 | | FMOV.D @Rn,DRn FMOV.D Rm,@Rn FMOV.D Rm,(@disp12,Rn)
MF.FL | MF.FL | LDS Rn,FPU.L
MF.FC | MF.FC | LDS Rn,FPUL
MR.FC | MR.FC | LDS L @Rm+,FPSCR LDS L @Rm+,FPUL
MW.ML.FC | MW.ML.FC | STS.L FPSCR,@-Rn STS.L FPUL,@-Rn
BR | BR | JSR/N @@(disp8,TBR)
### Classification of First Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Classification of Second Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR, MU</td>
<td>RESBANK(BO==1)</td>
</tr>
<tr>
<td>EX</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>BAND.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>BANDNOT.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>BLD.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>BLDNOT.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>BOR.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>BORNOT.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>BXOR.B #imm3,@(disp12,Rn)</td>
</tr>
<tr>
<td></td>
<td>LDC.L @Rn+,SR</td>
</tr>
<tr>
<td></td>
<td>RTE</td>
</tr>
<tr>
<td></td>
<td>SLEEP</td>
</tr>
<tr>
<td></td>
<td>TST.B #imm,@(R0,GBR)</td>
</tr>
<tr>
<td>MU</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>MAC.W @Rn+,@Rn+</td>
</tr>
<tr>
<td></td>
<td>MAC.L @Rn+,@Rn+</td>
</tr>
</tbody>
</table>

- The first and last steps of multi-step instructions are executed in parallel.
- FPU instructions follow the SH4 classifications ((1) LS type, (2) FE type, (3) CO type). The new 32-bit FMOV instructions belong to the (1) LS type.
- As a rule, 32-bit instructions are executed in parallel if the preceding instruction is a multi-step instruction. They cannot be executed in parallel with the instructions that follow them. However, pairs of memory-Tbit bit-manipulation instructions are executed in parallel.
- The MOVML.L and MOVML.L instructions cannot be executed in parallel with the instructions that follow them.
- Parallel execution of delayed branch instructions and delayed slots is not supported.

#### Multi-step instructions:
- TRAPA, MOVMU.L, MOVML.L, AND.B, OR.B, TST.B, XOR.B, TAS.B, BCLR.B, BSET.B, BSET.B, BAND.B, BANDOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BXOR.B, MUL.L, DMULS.L, DMULU.L, MULR, DIVU, DIVS, FCMP/EG DRm,DRn, FCMP/GT DRm,DRn, LDC Rm,SR, STC SR,Rn, LDCL @Rm+,SR, STC.L SR,@-Rn, LDBANK, STBANK, RESBANK, FMOV.D, FMOV DRm,DRn, JSR/N @@(disp,TBR), SLEEP, RTE, MAC.W, MAC.L

#### 32-bit instructions:
- MOVI20, MOVI20S, MOV.B @(disp12,Rm),Rn, MOV.W @(disp12,Rm),Rn, MOV.L @(disp12,Rm),Rn, MOV.B Rm,@(disp12,Rn), MOV.W Rm,@(disp12,Rn), MOV.L Rm,@(disp12,Rn), MOV.B, MOV.W, MOV.S @(disp12,Rm),FRn, FMOV.D @(disp12,Rm),DRn, FMOV.S FRm,@(disp12,Rn), FMOV.D DRm,@(disp12,Rn), BCLR.B, BSET.B, BSET.B, BAND.B, BANDOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BXOR.B

#### 32-bit FMOV instructions:
- FMOV.S @(disp12,Rm),FRn, FMOV.D @(disp12,Rm),DRn, FMOV.S FRm,@(disp12,Rn), FMOV.D DRm,@(disp12,Rn),

#### Memory-Tbit bit-manipulation instructions:
- BAND.B, BANDNOT.B, BLD.B, BLDNOT.B, BOR.B, BORNOT.B, BXOR.B

#### Delayed branch instructions:
- BRA, BSR, BRAF, BSRF, JMP, JSR, RTS, RTE, BT/S, BF/S
Appendix B   Programming Guidelines
(Using MOVI20 and MOVI20S)

In the SH-2A/SH2A-FPU, the MOVI20 #imm20,Rn and MOVI20S #imm20,Rn instructions reduce literal access by PC-relative instructions and increase cycle performance. Use of a declaration of the sort shown below in the assembler is recommended in order to gain these benefits.

(1) Using MOVI20

MOVI20 performs sign extension. This instruction can be used to express the range H'00000000 to H'0007FFFF and H'FFF80000 to H'FFFFFFFF.

The following instruction string should be arranged continuously.
- MOVI20 #imm20, Rn
- Unconditional branch instruction*

Example:
- MOVI20 #imm20, Rn
- JMP @ Rm

(2) Using MOVI20S

MOVI20S performs sign extension. This instruction can be used with ADD #imm, Rn to express the range H'00000000 to H'07FFFF7F and H'F7FFFF80 to H'FFFFFFFF.

The following instruction string should be arranged continuously.
- MOVI20S #imm20, Rn
- ADD#imm, Rn
- Unconditional branch instruction*

Example:
- MOVI20S#imm20, Rn
- ADD#imm, Rn
- JMP @ Rm
Notes: To specify addresses in the range H'07FF FF80–H'07FF FFFF:
   MOVI20S #imm20, R0
   OR #imm, R0
   Unconditional branch instruction*
Alternately, use a 32-bit address read as follows:
   MOV.L @(disp, PC), Rn
   Unconditional branch instruction*

* Unconditional branch instruction: BRAF Rm, BSRF Rm, JMP @Rm, JSR @Rm,
   JSR/N @Rm
SH-2A, SH2A-FPU
Software Manual