

To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 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>

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (<http://www.renesas.com>)

Send any inquiries to <http://www.renesas.com/inquiry>.

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 depends 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-2E

Software Manual

Renesas 32-Bit RISC

Microcomputer

SuperH™ RISC engine Family/

SH7000 Series

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.

Introduction

The SH-2E is a new generation of RISC microcomputers that integrate a RISC-type CPU and the peripheral functions required for system configuration onto a single chip to achieve high-performance operation. It can operate in a power-down state, which is an essential feature for portable equipment.

This CPU has a RISC-type instruction set. Basic instructions can be executed in one clock cycle, improving instruction execution speed. In addition, the CPU has a 32-bit internal architecture for enhanced data-processing ability.

In addition, the SH-2E supports single-precision floating point calculations as well as entirely PCAPI compatible emulation of double-precision floating point calculations. The SH-2E instructions are a subset of the floating point calculations conforming to the IEEE754 standard.

This programming manual describes in detail the instructions for the SH-2E Series and is intended as a reference on instruction operation and architecture. It also covers the pipeline operation, which is a feature of the SH-2E Series.

For information on the hardware, please refer to the hardware manual for the product in question.

Main Revisions for This Edition

Item	Page	Revision (See Manual for Details)
All	—	• Notification of change in company name amended (Before) Hitachi, Ltd. → (After) Renesas Technology Corp.

Contents

Section 1	Features.....	1
1.1	SH-2E Features.....	1
Section 2	Register Configuration.....	3
2.1	General Registers.....	3
2.2	Control Registers.....	4
2.3	System Registers.....	5
2.4	Floating-Point Registers.....	6
2.5	Floating-Point System Registers.....	7
2.6	Initial Values of Registers.....	8
Section 3	Data Formats.....	9
3.1	Data Format in Registers.....	9
3.2	Data Format in Memory.....	9
3.3	Immediate Data Format.....	10
Section 4	Floating-Point Unit (FPU).....	11
4.1	Overview.....	11
4.2	Floating-Point Registers and Floating-Point System Registers.....	12
4.2.1	Floating-Point Register File.....	12
4.2.2	Floating-Point Communication Register (FPUL).....	12
4.2.3	Floating-Point Status/Control Register (FPSCR).....	12
4.3	Floating-Point Format.....	15
4.3.1	Floating-Point Format.....	15
4.3.2	Non-Numbers (NaN).....	16
4.3.3	Denormalized Number Values.....	16
4.3.4	Other Special Values.....	17
4.4	Floating-Point Exception Model.....	17
4.4.1	Enable State Exceptions.....	17
4.4.2	Disable State Exceptions.....	17
4.4.3	FPU Exception Event and Code.....	18
4.4.4	Floating-Point Data Arrangement in Memory.....	18
4.4.5	Arithmetic Operations Involving Special Operands.....	18
4.5	Synchronization with CPU.....	18
Section 5	Instruction Features.....	19
5.1	RISC-Type Instruction Set.....	19
5.2	Addressing Modes.....	22

5.3	Instruction Format.....	25
Section 6	Instruction Set.....	29
6.1	Instruction Set by Classification	29
6.2	Instruction Set in Alphabetical Order.....	44
Section 7	Instruction Descriptions.....	53
7.1	Sample Description (Name): Classification	53
7.2	CPU Instruction.....	57
7.2.1	ADD (ADD Binary): Arithmetic Instruction	57
7.2.2	ADDC (ADD with Carry): Arithmetic Instruction	58
7.2.3	ADDV (ADD with V Flag Overflow Check): Arithmetic Instruction.....	59
7.2.4	AND (AND Logical): Logic Operation Instruction.....	60
7.2.5	BF (Branch if False): Branch Instruction.....	62
7.2.6	BF/S (Branch if False with Delay Slot): Branch Instruction.....	63
7.2.7	BRA (Branch): Branch Instruction	65
7.2.8	BRAF (Branch Far): Branch Instruction	67
7.2.9	BSR (Branch to Subroutine): Branch Instruction.....	69
7.2.10	BSRF (Branch to Subroutine Far): Branch Instruction.....	71
7.2.11	BT (Branch if True): Branch Instruction.....	73
7.2.12	BT/S (Branch if True with Delay Slot): Branch Instruction	74
7.2.13	CLRMAC (Clear MAC Register): System Control Instruction.....	76
7.2.14	CLRT (Clear T Bit): System Control Instruction.....	77
7.2.15	CMP/cond (Compare Conditionally): Arithmetic Instruction.....	78
7.2.16	DIV0S (Divide Step 0 as Signed): Arithmetic Instruction.....	82
7.2.17	DIV0U (Divide Step 0 as Unsigned): Arithmetic Instruction	83
7.2.18	DIV1 (Divide 1 Step): Arithmetic Instruction	84
7.2.19	DMULS.L (Double-Length Multiply as Signed): Arithmetic Instruction	89
7.2.20	DMULU.L (Double-Length Multiply as Unsigned): Arithmetic Instruction.....	91
7.2.21	DT (Decrement and Test): Arithmetic Instruction.....	93
7.2.22	EXTS (Extend as Signed): Arithmetic Instruction.....	94
7.2.23	EXTU (Extend as Unsigned): Arithmetic Instruction.....	95
7.2.24	JMP (Jump): Branch Instruction	96
7.2.25	JSR (Jump to Subroutine): Branch Instruction (Class: Delayed Branch Instruction)	98
7.2.26	LDC (Load to Control Register): System Control Instruction (Class: Interrupt Disabled Instruction).....	100
7.2.27	LDS (Load to System Register): System Control Instruction	102
7.2.28	MAC.L (Multiply and Accumulate Calculation Long): Arithmetic Instruction.....	104

7.2.29	MAC.W (Multiply and Accumulate Calculation Word): Arithmetic Instruction	107
7.2.30	MOV (Move Data): Data Transfer Instruction	110
7.2.31	MOV (Move Immediate Data): Data Transfer Instruction.....	115
7.2.32	MOV (Move Peripheral Data): Data Transfer Instruction	117
7.2.33	MOV (Move Structure Data): Data Transfer Instruction	120
7.2.34	MOVA (Move Effective Address): Data Transfer Instruction.....	123
7.2.35	MOVT (Move T Bit): Data Transfer Instruction	124
7.2.36	MUL.L (Multiply Long): Arithmetic Instruction.....	125
7.2.37	MULS.W (Multiply as Signed Word): Arithmetic Instruction	126
7.2.38	MULU.W (Multiply as Unsigned Word): Arithmetic Instruction	127
7.2.39	NEG (Negate): Arithmetic Instruction	128
7.2.40	NEGC (Negate with Carry): Arithmetic Instruction	129
7.2.41	NOP (No Operation): System Control Instruction	130
7.2.42	NOT (NOT—Logical Complement): Logic Operation Instruction	131
7.2.43	OR (OR Logical) Logic Operation Instruction	132
7.2.44	ROTCL (Rotate with Carry Left): Shift Instruction.....	134
7.2.45	ROTCR (Rotate with Carry Right): Shift Instruction	135
7.2.46	ROTL (Rotate Left): Shift Instruction	136
7.2.47	ROTR (Rotate Right): Shift Instruction	137
7.2.48	RTE (Return from Exception): System Control Instruction.....	138
7.2.49	RTS (Return from Subroutine): Branch Instruction (Class: Delayed Branch Instruction).....	140
7.2.50	SETT (Set T Bit): System Control Instruction.....	142
7.2.51	SHAL (Shift Arithmetic Left): Shift Instruction	143
7.2.52	SHAR (Shift Arithmetic Right): Shift Instruction	144
7.2.53	SHLL (Shift Logical Left): Shift Instruction	145
7.2.54	SHLLn (Shift Logical Left n Bits): Shift Instruction	146
7.2.55	SHLR (Shift Logical Right): Shift Instruction	148
7.2.56	SHLRn (Shift Logical Right n Bits): Shift Instruction.....	149
7.2.57	SLEEP (Sleep): System Control Instruction	151
7.2.58	STC (Store Control Register): System Control Instruction (Interrupt Disabled Instruction).....	152
7.2.59	STS (Store System Register): System Control Instruction (Interrupt Disabled Instruction).....	154
7.2.60	SUB (Subtract Binary): Arithmetic Instruction	156
7.2.61	SUBC (Subtract with Carry): Arithmetic Instruction.....	157
7.2.62	SUBV (Subtract with V Flag Underflow Check): Arithmetic Instruction	158
7.2.63	SWAP (Swap Register Halves): Data Transfer Instruction	159
7.2.64	TAS (Test and Set): Logic Operation Instruction	161
7.2.65	TRAPA (Trap Always): System Control Instruction.....	162

7.2.66	TST (Test Logical): Logic Operation Instruction	163
7.2.67	XOR (Exclusive OR Logical): Logic Operation Instruction.....	165
7.2.68	XTRCT (Extract): Data Transfer Instruction	167
7.3	Floating Point Instructions and FPU Related CPU Instructions.....	168
7.3.1	FABS (Floating Point Absolute Value): Floating Point Instruction	170
7.3.2	FADD (Floating Point Add): Floating Point Instruction.....	172
7.3.3	FCMP (Floating Point Compare): Floating Point Instruction	175
7.3.4	FDIV (Floating Point Divide): Floating Point Instruction	179
7.3.5	FLDI0 (Floating Point Load Immediate 0): Floating Point Instruction	181
7.3.6	FLDI1 (Floating Point Load Immediate 1): Floating Point Instruction	182
7.3.7	FLDS (Floating Point Load to System Register): Floating Point Instruction	183
7.3.8	FLOAT (Floating Point Convert from Integer): Floating Point Instruction.....	184
7.3.9	FMAC (Floating Point Multiply Accumulate): Floating Point Instruction	185
7.3.10	FMOV (Floating Point Move): Floating Point Instruction	188
7.3.11	FMUL (Floating Point Multiply): Floating Point Instruction	192
7.3.12	FNEG (Floating Point Negate): Floating Point Instruction.....	194
7.3.13	FSTS (Floating Point Store From System Register): Floating Point Instruction	195
7.3.14	FSUB (Floating Point Subtract): Floating Point Instruction	196
7.3.15	FTRC (Floating Point Truncate And Convert To Integer): Floating Point Instruction	199
7.3.16	LDS (Load to System Register): FPU Related CPU Instruction.....	201
7.3.17	STS (Store from FPU System Register): FPU Related CPU Instruction	204
Section 8 Pipeline Operation.....		207
8.1	Basic Configuration of Pipelines.....	207
8.2	Slot and Pipeline Flow	209
8.3	Number of Instruction Execution Cycles	211
8.4	Contention between Instruction Fetch (IF) and Memory Access (MA).....	212
8.5	Effects of Memory Load Instructions on the Pipeline.....	215
8.6	FPU Contention.....	216
8.7	Programming Guide.....	217
8.8	Operation of Instruction Pipelines.....	218
8.8.1	Data Transfer Instructions.....	228
8.8.2	Arithmetic Instructions	231
8.8.3	Logic Operation Instructions	265
8.8.4	Shift Instructions	267
8.8.5	Branch Instructions	268
8.8.6	System Control Instructions	271
8.8.7	Exception Processing.....	277

8.8.8 Relationship between Floating-point Instructions and FPU-related CPU Instructions.....	279
--	-----

Appendix A Instruction Code	293
A.1 Instruction Set by Addressing Mode.....	293
A.1.1 No Operand.....	294
A.1.2 Direct Register Addressing	295
A.1.3 Indirect Register Addressing.....	299
A.1.4 Post-Increment Indirect Register Addressing	300
A.1.5 Pre-Decrement Indirect Register Addressing.....	301
A.1.6 Indirect Register Addressing with Displacement.....	301
A.1.7 Indirect Indexed Register Addressing.....	302
A.1.8 Indirect GBR Addressing with Displacement.....	302
A.1.9 Indirect Indexed GBR Addressing.....	303
A.1.10 PC Relative Addressing with Displacement	303
A.1.11 PC Relative Addressing	304
A.1.12 Immediate	305
A.2 Instruction Sets by Instruction Format	306
A.2.1 0 Format.....	307
A.2.2 n Format.....	308
A.2.3 m Format.....	310
A.2.4 nm Format.....	312
A.2.5 md Format.....	316
A.2.6 nd4 Format.....	316
A.2.7 nmd Format.....	316
A.2.8 d Format.....	317
A.2.9 d12 Format.....	318
A.2.10 nd8 Format.....	318
A.2.11 i Format.....	318
A.2.12 ni Format.....	319
A.3 Instruction Set by Instruction Code.....	320
A.4 Operation Code Map.....	329
Appendix B Pipeline Operation and Contention.....	332

Section 1 Features

1.1 SH-2E Features

The SH-2E CPU has RISC-type instruction sets. Basic instructions are executed in one clock cycle, which dramatically improves instruction execution speed. The CPU also has an internal 32-bit architecture for enhanced data processing ability. Table 1.1 lists the SH-2E CPU features.

Table 1.1 SH-2E CPU Features

Item	Feature
Architecture	<ul style="list-style-type: none"> • Original Renesas Technology architecture • 32-bit internal data bus
General-register machine	<ul style="list-style-type: none"> • Sixteen 32-bit general registers • Three 32-bit control registers • Four 32-bit system registers • Sixteen 32-bit floating-point registers • Two 32-bit floating point system registers
Instruction set	<ul style="list-style-type: none"> • Instruction length: 16-bit fixed length for improved code efficiency • Load-store architecture (basic arithmetic and logic operations are executed between registers) • Delayed branch system used for reduced pipeline disruption • Instruction set optimized for C language
Instruction execution time	<ul style="list-style-type: none"> • One instruction/cycle for basic instructions
Address space	<ul style="list-style-type: none"> • Architecture makes 4 Gbytes available
On-chip multiplier	<ul style="list-style-type: none"> • Multiplication operations executed in 1 to 2 cycles (16 bits \times 16 bits \rightarrow 32 bits) or 2 to 4 cycles (32 bits \times 32 bits \rightarrow 64 bits), and multiplication/accumulation operations executed in $3/(2)^*$ cycles (16 bits \times 16 bits + 64 bits \rightarrow 64 bits) or $3/(2 \text{ to } 4)^*$ cycles (32 bits \times 32 bits + 64 bits \rightarrow 64 bits)
Pipeline	<ul style="list-style-type: none"> • Five-stage pipeline
Processing states	<ul style="list-style-type: none"> • Reset state • Exception processing state • Program execution state • Power-down state • Bus release state

Item	Feature
Power-down states	<ul style="list-style-type: none">• Sleep mode• Standby mode
FPU	<ul style="list-style-type: none">• Single-precision floating point format• Subset of IEEE754 standard data types• Invalid calculation exception and divide-by-zero exception (in compliance with IEEE754 standard)• Rounding to zero (in compliance with IEEE754 standard)• General purpose register file, 16 32-bit floating point registers• Execution pitch for basic instructions: 1 cycle/latency or 2 cycles (FADD, FSUB, FMUL)• FMAC (floating point multiply accumulate) Execution pitch: 1 cycle/latency or 2 cycles• Support for FDIV• Support for FLDI0 and FLDI1 (load constant 0/1)

Note: * The normal minimum number of execution cycles. The number in parentheses in the number in contention with preceding/following instructions.

Section 2 Register Configuration

The register set consists of sixteen 32-bit general registers, three 32-bit control registers and four 32-bit system registers.

2.1 General Registers

There are 16 general registers (Rn) numbered R0–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.

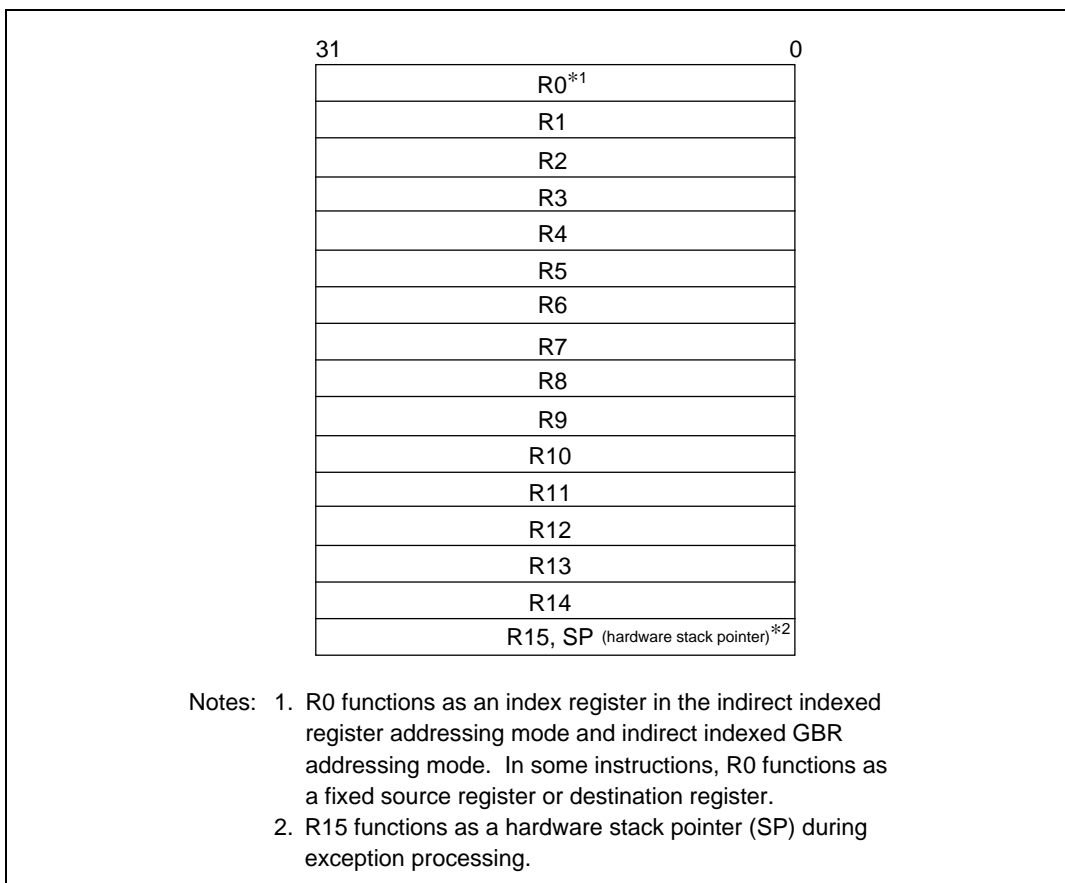


Figure 2.1 General Registers (SH-1 and SH-2)

2.2 Control Registers

The 32-bit control registers consist of the 32-bit status register (SR), global base register (GBR), and vector base register (VBR). The status register indicates processing states. The global base register functions as a base address for the indirect GBR addressing mode to transfer data to the registers of on-chip peripheral modules. The vector base register functions as the base address of the exception processing vector area (including interrupts).

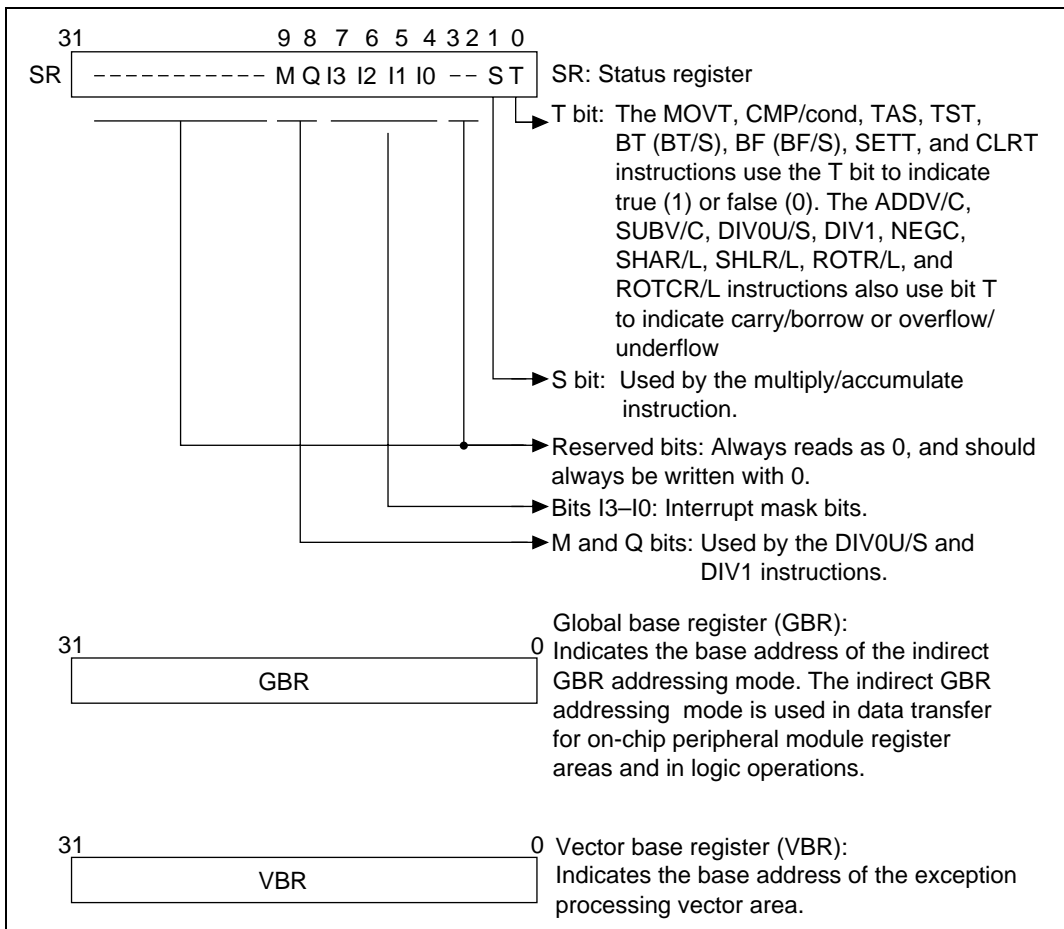


Figure 2.2 Control Registers

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.

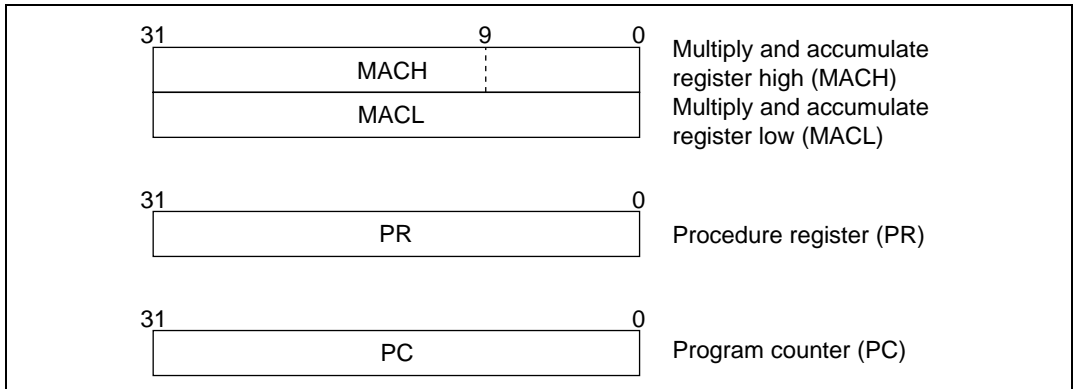


Figure 2.3 Organization of the System Registers

2.4 Floating-Point Registers

There are sixteen 32-bit floating-point registers, designated FR0 to FR15, which are used by floating-point instructions. FR0 functions as the index register for the FMAC instruction. These registers are incorporated into the floating-point unit (FPU). For details, see section 4, Floating-Point Unit (FPU).

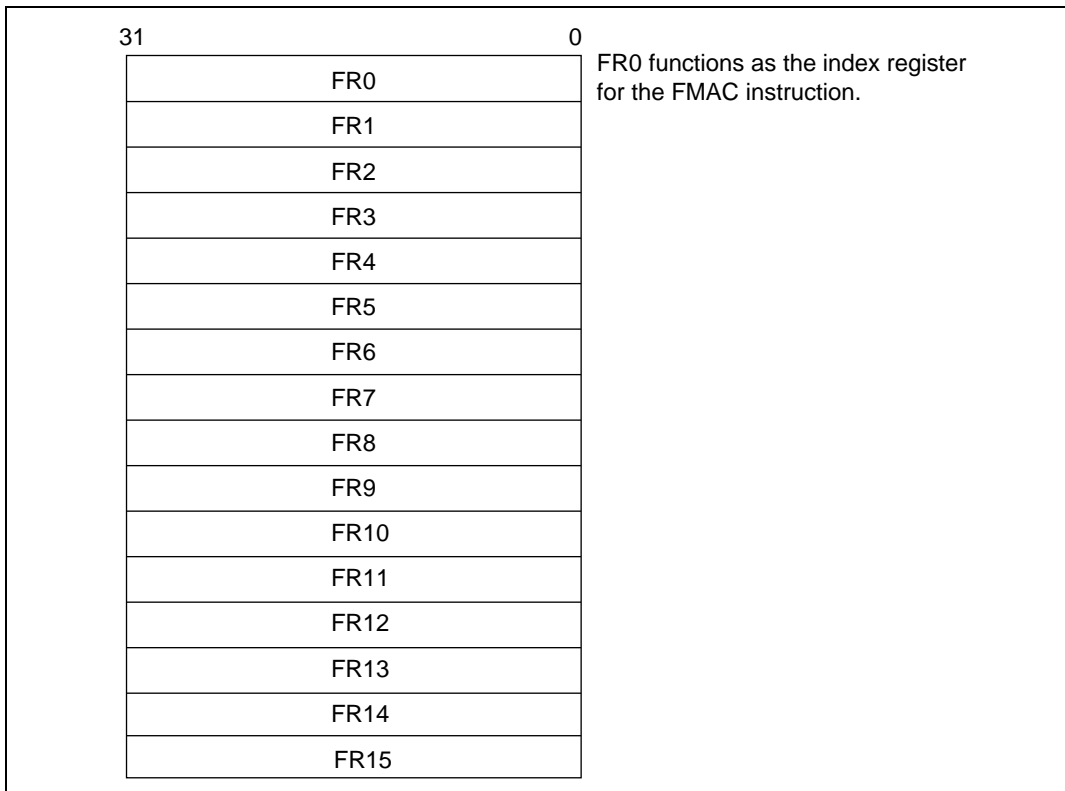


Figure 2.4 Floating-Point Registers

2.5 Floating-Point System Registers

There are two 32-bit floating-point system registers: the floating-point communication register (FPUL) and the floating-point status/control register (FPSCR). FPUL is used for communication between the CPU and the floating-point unit (FPU). FPSCR indicates and stores status/control information relating to FPU exceptions.

These registers are incorporated into the floating-point unit (FPU). For details, see section 4, Floating-Point Unit (FPU).

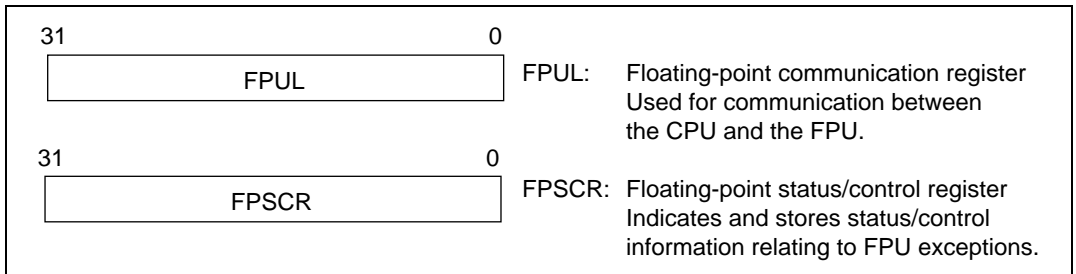


Figure 2.5 Floating-Point System Registers

2.6 Initial Values of Registers

Table 2.1 lists the values of the registers after reset.

Table 2.1 Initial Values of Registers

Classification	Register	Initial Value
General registers	R0–R14	Undefined
	R15 (SP)	Value of the stack pointer in the vector address table
Control registers	SR	Bits I3–I0 are 1111 (H'F), reserved bits are 0, and other bits are undefined
	GBR	Undefined
	VBR	H'00000000
System registers	MACH, MACL, PR	Undefined
	PC	Value of the program counter in the vector address table
Floating-point registers	FR0–FR15	Undefined
Floating-point system registers	FPUL	Undefined
	FPSCR	H'00040001

Section 3 Data Formats

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.

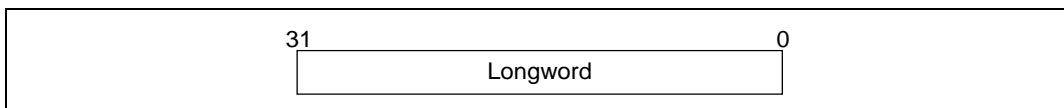


Figure 3.1 Data Format in Registers

3.2 Data Format in Memory

Memory data formats are classified into bytes, words, and longwords. Byte data can be accessed from any address, but an address error will occur if you try to access word data starting from an address other than $2n$ or longword data starting from an address other than $4n$. In such cases, the data accessed cannot be guaranteed. The hardware stack area, which is referred to by the hardware stack pointer (SP, R15), uses only longword data starting from address $4n$ because this area stores the program counter (PC) and status register (SR). See the hardware manual for more information on address errors.

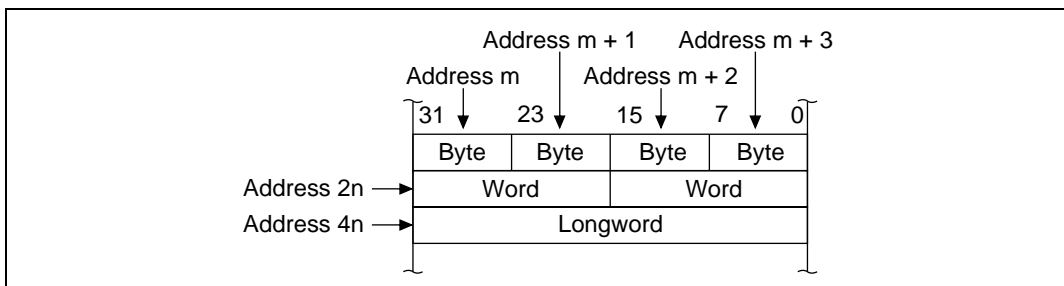


Figure 3.2 Data Format in Memory

3.3 Immediate Data Format

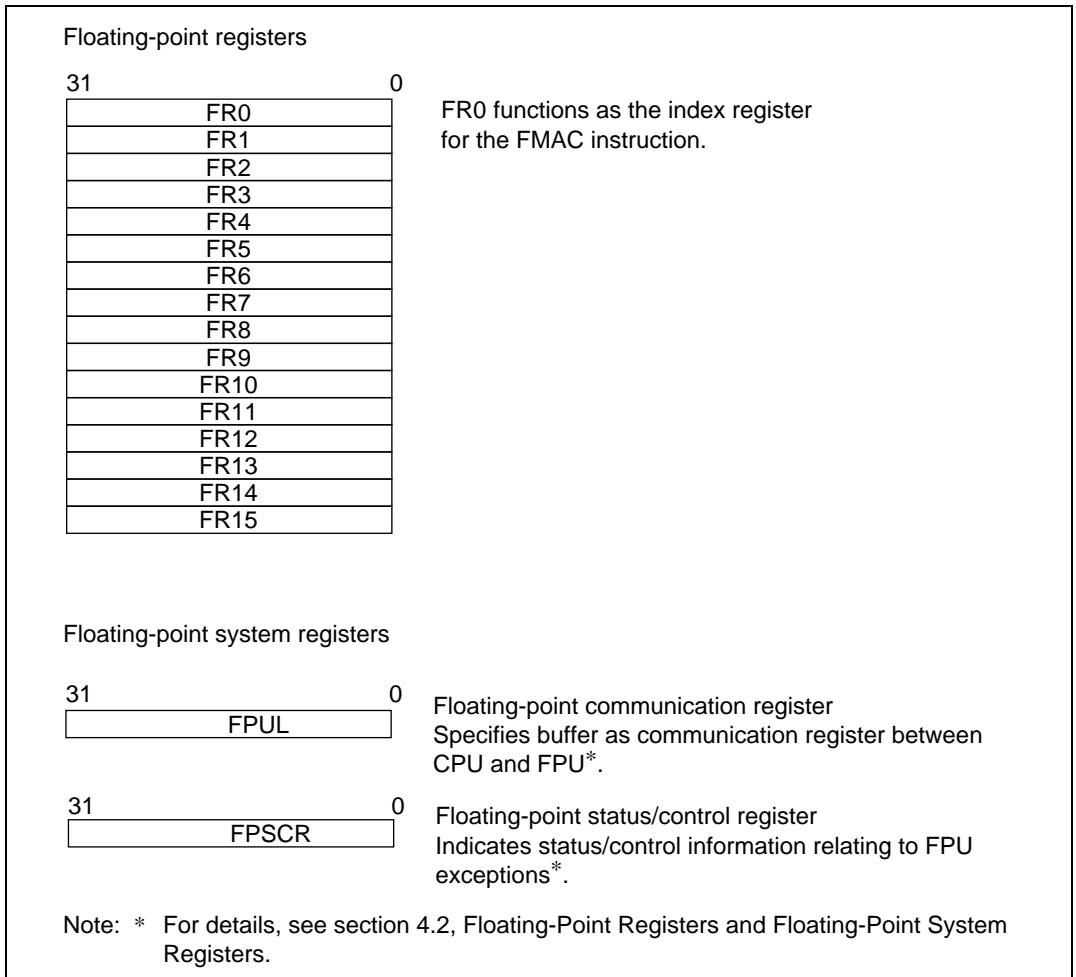
Byte immediate data is located in an instruction code. Immediate data accessed by the MOV, ADD, and CMP/EQ instructions is sign-extended and is handled in registers as longword data. Immediate data accessed by the TST, AND, OR, and XOR instructions is zero-extended and is handled as longword data. Consequently,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

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 5.1 Immediate Data in Section 5, Instruction Features.

Section 4 Floating-Point Unit (FPU)

4.1 Overview

The SH-2E has an on-chip floating-point unit (FPU). The FPU's register configuration is shown in figure 4.1.



**Figure 4.1 Overview of Register Configuration
(Floating-Point Registers and Floating-Point System Registers)**

4.2 Floating-Point Registers and Floating-Point System Registers

4.2.1 Floating-Point Register File

The SH-2E has sixteen 32-bit single-precision floating-point registers. Register specifications are always made as 4 bits. In assembly language, the floating-point registers are specified as FR0, FR1, FR2, and so on. FR0 functions as the index register for the FMAC instruction.

4.2.2 Floating-Point Communication Register (FPUL)

Information for transfer between the FPU and the CPU is transferred via the FPUL communication register, which resembles MACL and MACH in the integer unit. The SH-2E is provided with this communication register since the integer and floating-point formats are different. The 32-bit FPUL is a system register, and is accessed by the CPU by means of LDS and STS instructions.

4.2.3 Floating-Point Status/Control Register (FPSCR)

The SH-2E has a floating-point status/control register (FPSCR) that functions as a system register accessed by means of LDS and STS instructions (figure 4.2). FPSCR can be written to by a user program. This register is part of the process context, and must be saved when the context is switched. It may also be necessary to save this register when a procedure call is made.

FPSCR is a 32-bit register that controls the storage of detailed information relating to the rounding mode, asymptotic underflow (denormalized numbers), and FPU exceptions. The module stop bit that disables the FPU itself is provided in the module standby control register (MSTCR). For details, refer to hardware manual. After a reset start, the FPU is enabled.

Table 4.1 shows the flags corresponding the five kinds of FPU exception. A sixth flag is also provided as an FPU error flag that indicates an floating-point unit error state not covered by the other five flags.

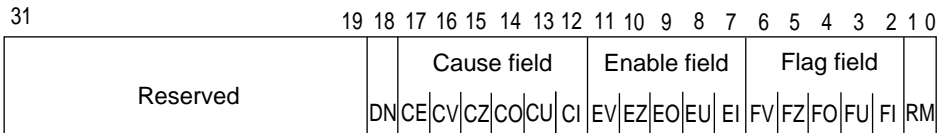
Table 4.1 Floating-Point Exception Flags

Flag	Meaning	Support in SH-2E
E	FPU error	—
V	Invalid operation	Yes
Z	Division by zero	Yes
O	Overflow (value not expressed)	—
U	Underflow (value not expressed)	—
I	Inexact (result not expressed)	—

The bits in the cause field indicate the exception cause for the instruction executing at the time. The cause bits are modified by a floating-point instruction. These bits are set to 1 or cleared to 0 according to whether or not an exception state occurred during execution of a single instruction.

The bits in the enable field specify the kinds of exception to be enabled, allowing the flow to be changed to exception processing. If the cause bit corresponding to an enable bit is set by the currently executing instruction, an exception occurs.

The bits in the flag field are used to keep a tally of all exceptions that occur during a series of instructions. Once one of these bits is set by an instruction, it is not reset by a subsequent instruction. The bits in this field can only be reset by the explicit execution of a store operation on FPSCR.



DN: Denormalized bit

In the SH-2E this bit is always set to 1, and the source or destination operand of a denormalized number is 0. This bit cannot be modified even by an LDS instruction.

CV: Invalid operation cause bit

When 1: Indicates that an invalid operation exception occurred during execution of the current instruction.

When 0: Indicates that an invalid operation exception has not occurred.

CZ: Division-by-zero cause bit

When 1: Indicates that a division-by-zero exception occurred during execution of the current instruction.

When 0: Indicates that a division-by-zero exception has not occurred.

EV: Invalid operation exception enable

When 1: Enables invalid operation exception generation.

When 0: An invalid operation exception is not generated, and a qNaN is returned as the result.

EZ: Division-by-zero exception enable

When 1: Enables exception generation due to division-by-zero during execution of the current instruction.

When 0: A division-by-zero exception is not generated, and infinity with the sign (+ or -) of the current expression is returned as the result.

FV: Invalid operation exception flag bit

When 1: Indicates that an invalid operation exception occurred during instruction execution.

When 0: Indicates that an invalid operation exception has not occurred.

FZ: Division-by-zero exception flag bit

When 1: Indicates that a division-by-zero exception occurred during instruction execution.

When 0: Indicates that a division-by-zero exception has not occurred.

RM: Rounding bits. In the SH-2E, the value of these bits is always 01, meaning that rounding to zero (RZ mode) is being used. These bits cannot be modified even by an LDS instruction.

In the SH-2E, the cause field EOUI bits (CE, CO, CU, and CI), enable field OUI bits (EO, EU, and EI), and flag field OUI bits (FO, FU, and FI), and the reserved area, are preset to 0, and cannot be modified even by using an LDS instruction.

Figure 4.2 Floating-Point Status/Control Register

4.3 Floating-Point Format

4.3.1 Floating-Point Format

The SH-2E supports single-precision floating-point operations, and fully complies with the IEEE754 floating-point standard.

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

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

The exponent is expressed in biased form, as follows:

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

The range of unbiased exponent E is $E_{\min} - 1$ to $E_{\max} + 1$. The two values $E_{\min} - 1$ and $E_{\max} + 1$ are distinguished as follows. $E_{\min} - 1$ indicates zero (both positive and negative sign) and a denormalized number, and $E_{\max} + 1$ indicates positive or negative infinity or a non-number (NaN). In a single-precision operation, the bias value is 127, E_{\min} is -126 , and E_{\max} is 127.

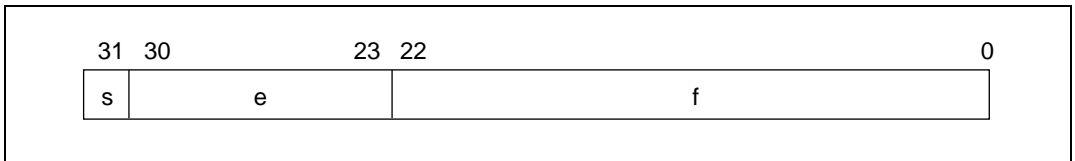


Figure 4.3 Floating-Point Number Format

Floating-point number value v is determined as follows:

If $E = E_{\max} + 1$ and $f! = 0$, v is a non-number (NaN) irrespective of sign s

If $E = E_{\max} + 1$ and $f = 0$, $v = (-1)^s$ (infinity) [positive or negative infinity]

If $E_{\min} \leq E \leq E_{\max}$, $v = (-1)^s 2^E (1.f)$ [normalized number]

If $E = E_{\min} - 1$ and $f! = 0$, $v = (-1)^s 2^{E_{\min}}$ (0.f) [denormalized number]

If $E = E_{\min} - 1$ and $f = 0$, $v = (-1)^s 0$ [positive or negative zero]

4.3.2 Non-Numbers (NaN)

With non-number (NaN) representation in a single-precision operation value, at least one of bits 22 to 0 is set. If bit 22 is set, this indicates a signaling NaN (sNaN). If bit 22 is reset, the value is a quiet NaN (qNaN).

The bit pattern of a non-number (NaN) is shown in the figure below. Bit N in the figure is set for a signaling NaN and reset for a quiet NaN. x indicates a don't care bit (with the proviso that at least one of bits 22 to 0 is set). In a non-number (NaN), the sign bit is a don't care bit.

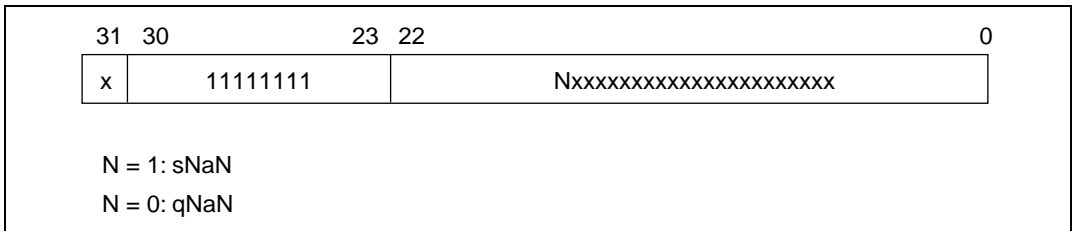


Figure 4.4 NaN Bit Pattern

If a non-number (sNaN) is input in an operation that generates a floating-point value:

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

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

Refer to section 7, Instruction Descriptions for details of floating-point operations when a non-number (NaN) is input.

4.3.3 Denormalized Number Values

For a denormalized number floating-point value, the biased exponent is expressed as 0, the fraction as a non-zero value, and the hidden bit as 0. In the SH-2E's floating-point unit, a denormalized number (operand source or operation result) is always flushed to 0 in a floating-point operation that generates a value (an operation other than copy).

4.3.4 Other Special Values

Floating-point value representations include the seven different kinds of special values shown in table 4.2.

Table 4.2 Representation of Special Values in Single-Precision Floating-Point Operations Specified by IEEE754 Standard

Value	Representation
+0.0	0x00000000
-0.0	0x80000000
Denormalized number	As described in 4.3.3, Denormalized Number Values
+INF	0x7F800000
-INF	0xFF800000
qNaN (quiet NaN)	As described in 4.3.2, Non-Numbers (NaN)
sNaN (signaling NaN)	As described in 4.3.2, Non-Numbers (NaN)

4.4 Floating-Point Exception Model

4.4.1 Enable State Exceptions

Invalid operation and division-by-zero exceptions are both placed in the enable state by setting the enable bit. All exceptions generated by the FPU are mapped as the same exception event. The meaning of a particular exception is determined by software by reading system register FPSCR and analyzing the information held there.

4.4.2 Disable State Exceptions

If the EV enable bit is not set, a qNaN will be generated as the result of an invalid operation (except for FCMP and FTRC). If the EZ enable bit is not set, division-by-zero will return infinity with the sign (+ or -) of the current expression. Overflow will generate a finite number which is the largest value that can be expressed by an absolute value in the format, with the correct sign. Underflow will generate zero with the correct sign. If the operation result is inexact, the destination register will store that inexact result.

4.4.3 FPU Exception Event and Code

All FPU exceptions have a vector table address offset in address H'00000034 as the same general exception event; that is, an FPU exception.

4.4.4 Floating-Point Data Arrangement in Memory

Single-precision floating-point data is located in memory at a 4-byte boundary; that is, it is arranged in the same form as an SH-2E long integer.

4.4.5 Arithmetic Operations Involving Special Operands

All arithmetic operations involving special operands (qNaN, sNaN, +INF, -INF, +0, -0) comply with the specifications of the IEEE754 standard. Refer to section 7, Instruction Descriptions for details.

4.5 Synchronization with CPU

Synchronization with CPU: Floating-point instructions and CPU instructions are executed in turn, according to their order in the program, but in some cases operations may not be completed in the program order due to a difference in execution cycles. When a floating-point instruction accesses only FPU resources, there is no need for synchronization with the CPU, and a CPU instruction following an FPU instruction can finish its operation before completion of the FPU operation. Consequently, in an optimized program, it is possible to effectively conceal the execution cycle of a floating-point instruction that requires a long execution cycle, such as a divide instruction. On the other hand, a floating-point instruction that accesses CPU resources, such as a compare instruction, must be synchronized to ensure that the program order is observed.

Floating-Point Instructions That Require Synchronization: Load, store, and compare instructions, and instructions that access the FPUL or FPSCR register, must be synchronized because they access CPU resources. Load and store instructions access a general register. Post-increment load and pre-decrement store instructions change the contents of a general register. A compare instruction modifies the T bit. An FPUL or FPSCR access instruction references or changes the contents of the FPUL or FPSCR register. These references and changes must all be synchronized with the CPU.

Section 5 Instruction Features

5.1 RISC-Type Instruction Set

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

16-Bit Fixed Length: All instructions are 16 bits long, increasing program coding efficiency.

One Instruction/Cycle: Basic instructions can be executed in one cycle using the pipeline system. Instructions are executed in 50 ns at 40 MHz.

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 5.1 Sign Extension of Word Data

SH-2E CPU	Description	Example for Other CPU
MOV.W @(disp,PC),R1	Data is sign-extended to 32 bits, and R1 becomes H'0001234. It is next operated upon by an ADD instruction.	ADD.W #H'1234,R0
ADD R1,R0		
..... .DATA.W H'1234		

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

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.

Delayed Branch Instructions: Unconditional branch instructions are delayed. Pipeline disruption during branching is reduced by first executing the instruction that follows the branch instruction, and then branching (table 5.2). With delayed branching, branching occurs after execution of the slot instruction. However, instructions such as register changes etc. are executed in the order of delayed branch instruction, then delay slot instruction. For example, even if the register in which the branch destination address has been loaded is changed by the delay slot instruction, the branch will still be made using the value of the register prior to the change as the branch destination address.

Table 5.2 Delayed Branch Instructions

SH-2E CPU		Description	Example for Other CPU	
BRA	TRGET	Executes an ADD before branching to TRGET.	ADD.W	R1, R0
ADD	R1, R0		BRA	TRGET

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.

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 5.3 T Bit

SH-2E CPU		Description	Example for Other CPU	
CMP/GE	R1, R0	T bit is set when $R0 \geq R1$.	CMP.W	R1, R0
BT	TRGET0	The program branches to TRGET0 when $R0 \geq R1$ and to TRGET1 when $R0 < R1$.	BGE	TRGET0
BF	TRGET1		BLT	TRGET1
ADD	#-1, R0	T bit is not changed by ADD.	SUB.W	#1, R0
CMP/EQ	#0, R0	T bit is set when $R0 = 0$.	BEQ	TRGET
BT	TRGET	The program branches if $R0 = 0$.		

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.

Table 5.4 Immediate Data Accessing

Classification	SH-2E CPU		Example for Other CPU
8-bit immediate	MOV	#H'12,R0	MOV.B #H'12,R0
16-bit immediate	MOV.W	@(disp,PC),R0	MOV.W #H'1234,R0
		
	.DATA.W	H'1234	
32-bit immediate	MOV.L	@(disp,PC),R0	MOV.L #H'12345678,R0
		
	.DATA.L	H'12345678	

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

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.

Table 5.5 Absolute Address

Classification	SH-2E CPU		Example for Other CPU
Absolute address	MOV.L	@(disp,PC),R1	MOV.B @H'12345678,R0
	MOV.B	@R1,R0	
		
	.DATA.L	H'12345678	

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.

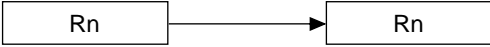
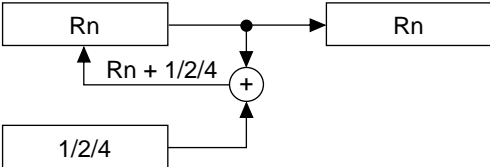
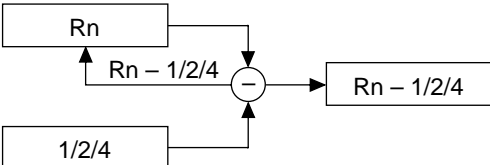
Table 5.6 Displacement Accessing

Classification	SH-2E CPU		Example for Other CPU
16-bit displacement	MOV.W	@(disp,PC),R0	MOV.W @(H'1234,R1),R2
	MOV.W	@(R0,R1),R2	
		
	.DATA.W	H'1234	

5.2 Addressing Modes

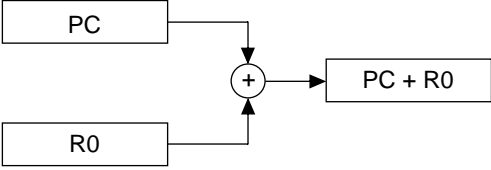
Addressing modes effective address calculation by the CPU core are described below.

Table 5.7 Addressing Modes and Effective Addresses

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
Direct register addressing	Rn	The effective address is register Rn. (The operand is the contents of register Rn.)	—
Indirect register addressing	@Rn	The effective address is the content of register Rn. 	Rn
Post-increment indirect register addressing	@Rn+	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. 	Rn (After the instruction is executed) Byte: Rn + 1 → Rn Word: Rn + 2 → Rn Longword: Rn + 4 → Rn
Pre-decrement indirect register addressing	@-Rn	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. 	Byte: Rn - 1 → Rn Word: Rn - 2 → Rn Longword: Rn - 4 → Rn (Instruction executed with Rn after calculation)

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
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: $Rn + disp \times 2$ Longword: $Rn + disp \times 4$
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: $GBR + disp \times 2$ Longword: $GBR + disp \times 4$
Indirect indexed GBR addressing	@(R0, GBR)	The effective address is the GBR value plus R0.	$GBR + R0$

Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
PC relative addressing with displacement	@(disp:8, PC)	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.	Word: $PC + disp \times 2$ Longword: $PC \& H'FFFFFFFC + disp \times 4$
PC relative addressing	disp:8	The effective address is the PC value sign-extended with an 8-bit displacement (disp), doubled, and added to the PC.	$PC + disp \times 2$
	disp:12	The effective address is the PC value sign-extended with a 12-bit displacement (disp), doubled, and added to the PC.	$PC + disp \times 2$

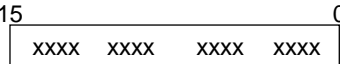
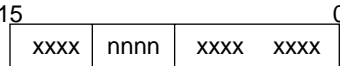
Addressing Mode	Instruction Format	Effective Addresses Calculation	Formula
PC relative addressing (cont)	Rn	The effective address is the register PC plus Rn. 	PC + Rn
Immediate addressing	#imm:8	The 8-bit immediate data (imm) for the TST, AND, OR, and XOR instructions are zero-extended.	—
	#imm:8	The 8-bit immediate data (imm) for the MOV, ADD, and CMP/EQ instructions are sign-extended.	—
	#imm:8	Immediate data (imm) for the TRAPA instruction is zero-extended and is quadrupled.	—

5.3 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
- iiiii: Immediate data
- dddd: Displacement

Table 5.8 Instruction Formats

Instruction Formats	Source Operand	Destination Operand	Example
0 format 	—	—	NOP
n format 	—	nnnn: Direct register	MOVT Rn
	Control register or system register	nnnn: Direct register	STS MACH, Rn

Instruction Formats	Source Operand	Destination Operand	Example				
n format (cont)	Control register or system register	nnnn: Indirect pre-decrement register	STC.L SR, @-Rn				
m format	mmmm: Direct register	Control register or system register	LDC Rm, SR				
<div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 25%;">xxxx</td> <td style="width: 25%;">mmmm</td> <td style="width: 25%;">xxxx</td> <td style="width: 25%;">xxxx</td> </tr> </table>	xxxx	mmmm	xxxx	xxxx	mmmm: Indirect post-increment register	Control register or system register	LDC.L @Rm+, SR
xxxx	mmmm	xxxx	xxxx				
	mmmm: Direct register	—	JMP @Rm				
	mmmm: PC relative using Rm*	—	BRAF Rm				
nm format	mmmm: Direct register	nnnn: Direct register	ADD Rm, Rn				
<div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 25%;">xxxx</td> <td style="width: 25%;">nnnn</td> <td style="width: 25%;">mmmm</td> <td style="width: 25%;">xxxx</td> </tr> </table>	xxxx	nnnn	mmmm	xxxx	mmmm: Direct register	nnnn: Indirect register	MOV.L Rm, @Rn
xxxx	nnnn	mmmm	xxxx				
	mmmm: Indirect post-increment register (multiply/accumulate)	MACH, MACL	MAC.W @Rm+, @Rn+				
	nnnn*: Indirect post-increment register (multiply/accumulate)						
	mmmm: Indirect post-increment register	nnnn: Direct register	MOV.L @Rm+, Rn				
	mmmm: Direct register	nnnn: Indirect pre-decrement register	MOV.L Rm, @-Rn				
	mmmm: Direct register	nnnn: Indirect indexed register	MOV.L Rm, @(R0, Rn)				
md format	mmmmdddd: indirect register with displacement	R0 (Direct register)	MOV.B @(disp, Rm), R0				
<div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 25%;">xxxx</td> <td style="width: 25%;">xxxx</td> <td style="width: 25%;">mmmm</td> <td style="width: 25%;">dddd</td> </tr> </table>	xxxx	xxxx	mmmm	dddd			
xxxx	xxxx	mmmm	dddd				

Instruction Formats	Source Operand	Destination Operand	Example
nd4 format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx xxxx nnnn dddd </div>	R0 (Direct register)	nnnnddd: Indirect register with displacement	MOV.B R0,@(disp,Rn)
nmd format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx nnnn mmmm dddd </div>	mmmm: Direct register	nnnnddd: Indirect register with displacement	MOV.L Rm,@(disp,Rn)
	mmmmddd: Indirect register with displacement	nnnn: Direct register	MOV.L @(disp,Rm),Rn
d format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx xxxx dddd dddd </div>	ddddddd: Indirect GBR with displacement	R0 (Direct register)	MOV.L @(disp,GBR),R0
	R0(Direct register)	ddddddd: Indirect GBR with displacement	MOV.L R0,@(disp,GBR)
	ddddddd: PC relative with displacement	R0 (Direct register)	MOVA @(disp,PC),R0
	ddddddd: PC relative	—	BF label
d12 format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx dddd dddd dddd </div>	ddddddddddd: PC relative	—	BRA label (label = disp + PC)
nd8 format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx nnnn dddd dddd </div>	ddddddd: PC relative with displacement	nnnn: Direct register	MOV.L @(disp,PC),Rn
i format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx xxxx iiii iiii </div>	iiiiiii: Immediate	Indirect indexed GBR	AND.B #imm,@(R0,GBR)
	iiiiiii: Immediate	R0 (Direct register)	AND #imm,R0
	iiiiiii: Immediate	—	TRAPA #imm
ni format <div style="display: flex; align-items: center; justify-content: space-between;"> 15 0 </div> <div style="border: 1px solid black; padding: 5px; display: flex; justify-content: space-around; width: 100%;"> xxxx nnnn iiii iiii </div>	iiiiiii: Immediate	nnnn: Direct register	ADD #imm,Rn

Note: * In multiply/accumulate instructions, nnnn is the source register.

Section 6 Instruction Set

6.1 Instruction Set by Classification

Table 6.1 shows instruction by classification

Table 6.1 Classification of Instructions

Classification	Types	Operation Code	Function	No. of Instructions
Data transfer	5	MOV	Data transfer, immediate data transfer, peripheral module data transfer, structure data transfer	39
		MOVA	Effective address transfer	
		MOVT	T bit transfer	
		SWAP	Swap of upper and lower bytes	
		XTRCT	Extraction of the middle of registers connected	
Arithmetic operations	21	ADD	Binary addition	33
		ADDC	Binary addition with carry	
		ADDV	Binary addition with overflow check	
		CMP/cond	Comparison	
		DIV1	Division	
		DIV0S	Initialization of signed division	
		DIV0U	Initialization of unsigned division	
		DMULS	Signed double-length multiplication	
		DMULU	Unsigned double-length multiplication	
		DT	Decrement and test	
		EXTS	Sign extension	
		EXTU	Zero extension	
		MAC	Multiply-and-accumulate, double-length multiply-and-accumulate operation	
		MUL	Double-length multiply operation	
		MULS	Signed multiplication	
MULU	Unsigned multiplication			
NEG	Negation			

Classification	Types	Operation Code	Function	No. of Instructions
Arithmetic operations (cont)	21	NEGC	Negation with borrow	33
		SUB	Binary subtraction	
		SUBC	Binary subtraction with borrow	
		SUBV	Binary subtraction with underflow	
Logic operations	6	AND	Logical AND	14
		NOT	Bit inversion	
		OR	Logical OR	
		TAS	Memory test and bit set	
		TST	Logical AND and T bit set	
		XOR	Exclusive OR	
Shift	10	ROTL	One-bit left rotation	14
		ROTR	One-bit right rotation	
		ROTCL	One-bit left rotation with T bit	
		ROTCR	One-bit right rotation with T bit	
		SHAL	One-bit arithmetic left shift	
		SHAR	One-bit arithmetic right shift	
		SHLL	One-bit logical left shift	
		SHLLn	n-bit logical left shift	
		SHLR	One-bit logical right shift	
		SHLRn	n-bit logical right shift	
Branch	9	BF	Conditional branch, conditional branch with delay (Branch when T = 0)	11
		BT	Conditional branch, conditional branch with delay (Branch when T = 1)	
		BRA	Unconditional branch	
		BRAF	Unconditional branch	
		BSR	Branch to subroutine procedure	
		BSRF	Branch to subroutine procedure	
		JMP	Unconditional branch	
		JSR	Branch to subroutine procedure	
		RTS	Return from subroutine procedure	

Classification	Types	Operation Code	Function	No. of Instructions
System control	11	CLRT	T bit clear	31
		CLRMAC	MAC register clear	
		LDC	Load to control register	
		LDS	Load to system register	
		NOP	No operation	
		RTE	Return from exception processing	
		SETT	T bit set	
		SLEEP	Transition to power-down mode	
		STC	Store control register data	
		STS	Store system register data	
		TRAPA	Trap exception handling	
Floating-point instructions	15	FABS	Floating-point absolute value	22
		FADD	Floating-point addition	
		FCMP	Floating-point comparison	
		FDIV	Floating-point division	
		FLDI0	Floating-point load immediate 0	
		FLDI1	Floating-point load immediate 1	
		FLDS	Floating-point load into system register FPUL	
		FLOAT	Integer-to-floating-point conversion	
		FMAC	Floating-point multiply-and-accumulate operation	
		FMOV	Floating-point data transfer	
		FMUL	Floating-point multiplication	
		FNEG	Floating-point sign inversion	
		FSTS	Floating-point store from system register FPUL	
		FSUB	Floating-point subtraction	
		FTRC	Floating-point conversion with rounding to integer	
FPU-related CPU instructions	2	LDS	Load into floating-point system register	8
		STS	Store from floating-point system register	
Total:	79			172

Table 6.2 shows the format used in tables 6.3 to 6.8, which list instruction codes, operation, and execution states in order by classification.

Table 6.2 Instruction Code Format

Item	Format	Explanation
Instruction	OP.Sz SRC,DEST	OP: Operation code Sz: Size (B: byte, W: word, or L: longword) SRC: Source DEST: Destination Rm: Source register Rn: Destination register imm: Immediate data disp: Displacement* ¹
Instruction code	MSB ↔ LSB	mmmm: Source register nnnn: Destination register 0000: R0 0001: R1 ⋮ 1111: R15 iiii: Immediate data dddd: Displacement
Operation	→, ← (xx) M/Q/T & ^ ~ << n >> n	Direction of transfer Memory operand 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-bit left shift 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 $\times 1$, $\times 2$, or $\times 4$. For details, see section 7, 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.

Table 6.3 Data Transfer Instructions

Instruction	Instruction Code	Operation	Execution Cycles	T Bit
MOV #imm,Rn	1110nnnniiiiiiii	imm → Sign extension → Rn	1	—
MOV.W @(disp,PC),Rn	1001nnnnddddddd	(disp × 2 + PC) → Sign extension → Rn	1	—
MOV.L @(disp,PC),Rn	1101nnnnddddddd	(disp × 4 + PC) → Rn	1	—
MOV Rm,Rn	0110nnnnmmmm0011	Rm → Rn	1	—
MOV.B Rm,@Rn	0010nnnnmmmm0000	Rm → (Rn)	1	—
MOV.W Rm,@Rn	0010nnnnmmmm0001	Rm → (Rn)	1	—
MOV.L Rm,@Rn	0010nnnnmmmm0010	Rm → (Rn)	1	—
MOV.B @Rm,Rn	0110nnnnmmmm0000	(Rm) → Sign extension → Rn	1	—
MOV.W @Rm,Rn	0110nnnnmmmm0001	(Rm) → Sign extension → Rn	1	—
MOV.L @Rm,Rn	0110nnnnmmmm0010	(Rm) → Rn	1	—
MOV.B Rm,@-Rn	0010nnnnmmmm0100	Rn-1 → Rn, Rm → (Rn)	1	—
MOV.W Rm,@-Rn	0010nnnnmmmm0101	Rn-2 → Rn, Rm → (Rn)	1	—
MOV.L Rm,@-Rn	0010nnnnmmmm0110	Rn-4 → Rn, Rm → (Rn)	1	—
MOV.B @Rm+,Rn	0110nnnnmmmm0100	(Rm) → Sign extension → Rn, Rm + 1 → Rm	1	—
MOV.W @Rm+,Rn	0110nnnnmmmm0101	(Rm) → Sign extension → Rn, Rm + 2 → Rm	1	—
MOV.L @Rm+,Rn	0110nnnnmmmm0110	(Rm) → Rn, Rm + 4 → Rm	1	—
MOV.B R0,@(disp,Rn)	10000000nnnnddd	R0 → (disp + Rn)	1	—
MOV.W R0,@(disp,Rn)	10000001nnnnddd	R0 → (disp × 2 + Rn)	1	—
MOV.L Rm,@(disp,Rn)	0001nnnnmmmmddd	Rm → (disp × 4 + Rn)	1	—
MOV.B @(disp,Rm),R0	10000100mmmmddd	(disp + Rm) → Sign extension → R0	1	—
MOV.W @(disp,Rm),R0	10000101mmmmddd	(disp × 2 + Rm) → Sign extension → R0	1	—
MOV.L @(disp,Rm),Rn	0101nnnnmmmmddd	(disp × 4 + Rm) → Rn	1	—
MOV.B Rm,@(R0,Rn)	0000nnnnmmmm0100	Rm → (R0 + Rn)	1	—

Instruction	Instruction Code	Operation	Execution	
			Cycles	T Bit
MOV.W Rm,@(R0,Rn)	0000nnnnmmmm0101	Rm → (R0 + Rn)	1	—
MOV.L Rm,@(R0,Rn)	0000nnnnmmmm0110	Rm → (R0 + Rn)	1	—
MOV.B @(R0,Rm),Rn	0000nnnnmmmm1100	(R0 + Rm) → Sign extension → Rn	1	—
MOV.W @(R0,Rm),Rn	0000nnnnmmmm1101	(R0 + Rm) → Sign extension → Rn	1	—
MOV.L @(R0,Rm),Rn	0000nnnnmmmm1110	(R0 + Rm) → Rn	1	—
MOV.B R0,@(disp,GBR)	11000000dddddddd	R0 → (disp + GBR)	1	—
MOV.W R0,@(disp,GBR)	11000001dddddddd	R0 → (disp × 2 + GBR)	1	—
MOV.L R0,@(disp,GBR)	11000010dddddddd	R0 → (disp × 4 + GBR)	1	—
MOV.B @(disp,GBR),R0	11000100dddddddd	(disp + GBR) → Sign extension → R0	1	—
MOV.W @(disp,GBR),R0	11000101dddddddd	(disp × 2 + GBR) → Sign extension → R0	1	—
MOV.L @(disp,GBR),R0	11000110dddddddd	(disp × 4 + GBR) → R0	1	—
MOVA @(disp,PC),R0	11000111dddddddd	disp × 4 + PC → R0	1	—
MOVT Rn	0000nnnn00101001	T → Rn	1	—
SWAP.B Rm,Rn	0110nnnnmmmm1000	Rm → Swap bottom two bytes → Rn	1	—
SWAP.W Rm,Rn	0110nnnnmmmm1001	Rm → Swap two consecutive words → Rn	1	—
XTRCT Rm,Rn	0010nnnnmmmm1101	Rm: Middle 32 bits of Rn → Rn	1	—

Table 6.4 Arithmetic Operation Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
ADD	Rm, Rn	0011nnnnmmmm1100	$Rn + Rm \rightarrow Rn$	1	—
ADD	#imm, Rn	0111nnnniiiiiii	$Rn + imm \rightarrow Rn$	1	—
ADDC	Rm, Rn	0011nnnnmmmm1110	$Rn + Rm + T \rightarrow Rn$, Carry $\rightarrow T$	1	Carry
ADDV	Rm, Rn	0011nnnnmmmm1111	$Rn + Rm \rightarrow Rn$, Overflow $\rightarrow T$	1	Overflow
CMP/EQ	#imm, R0	10001000iiiiiii	If $R0 = imm$, $1 \rightarrow T$	1	Comparison result
CMP/EQ	Rm, Rn	0011nnnnmmmm0000	If $Rn = Rm$, $1 \rightarrow T$	1	Comparison result
CMP/HS	Rm, Rn	0011nnnnmmmm0010	If $Rn \geq Rm$ with unsigned data, $1 \rightarrow T$	1	Comparison result
CMP/GE	Rm, Rn	0011nnnnmmmm0011	If $Rn \geq Rm$ with signed data, $1 \rightarrow T$	1	Comparison result
CMP/HI	Rm, Rn	0011nnnnmmmm0110	If $Rn > Rm$ with unsigned data, $1 \rightarrow T$	1	Comparison result
CMP/GT	Rm, Rn	0011nnnnmmmm0111	If $Rn > Rm$ with signed data, $1 \rightarrow T$	1	Comparison result
CMP/PL	Rn	0100nnnn00010101	If $Rn > 0$, $1 \rightarrow T$	1	Comparison result
CMP/PZ	Rn	0100nnnn00010001	If $Rn \geq 0$, $1 \rightarrow T$	1	Comparison result
CMP/STR	Rm, Rn	0010nnnnmmmm1100	If Rn and Rm have an equivalent byte, $1 \rightarrow T$	1	Comparison result
DIV1	Rm, Rn	0011nnnnmmmm0100	Single-step division ($Rn \div Rm$)	1	Calculation result
DIV0S	Rm, Rn	0010nnnnmmmm0111	MSB of Rn $\rightarrow Q$, MSB of Rm $\rightarrow M$, $M \wedge Q \rightarrow T$	1	Calculation result
DIV0U		000000000011001	$0 \rightarrow M/Q/T$	1	0
DMULS.L	Rm, Rn	0011nnnnmmmm1101	Signed operation of Rn $\times Rm \rightarrow MACH, MACL$ $32 \times 32 \rightarrow 64$ bits	2 to 4*	—

Instruction	Instruction Code	Operation	Execution Cycles	T Bit
DMULU.L Rm, Rn	0011nnnnmmmm0101	Unsigned operation of $Rn \times Rm \rightarrow MACH$, $MACL 32 \times 32 \rightarrow 64$ bits	2 to 4*	—
DT Rn	0100nnnn00010000	$Rn - 1 \rightarrow Rn$, when Rn is 0, $1 \rightarrow T$. When Rn is nonzero, $0 \rightarrow T$	1	Comparison result
EXTS.B Rm, Rn	0110nnnnmmmm1110	Byte in Rm is sign-extended $\rightarrow Rn$	1	—
EXTS.W Rm, Rn	0110nnnnmmmm1111	Word in Rm is sign-extended $\rightarrow Rn$	1	—
EXTU.B Rm, Rn	0110nnnnmmmm1100	Byte in Rm is zero-extended $\rightarrow Rn$	1	—
EXTU.W Rm, Rn	0110nnnnmmmm1101	Word in Rm is zero-extended $\rightarrow Rn$	1	—
MAC.L @Rm+, @Rn+	0000nnnnmmmm1111	Signed operation of $(Rn) \times (Rm) + MAC \rightarrow MAC 32 \times 32 + 64 \rightarrow 64$ bits	3/(2 to 4)*	—
MAC.W @Rm+, @Rn+	0100nnnnmmmm1111	Signed operation of $(Rn) \times (Rm) + MAC \rightarrow MAC 16 \times 16 + 64 \rightarrow 64$ bits	3/(2)*	—
MUL.L Rm, Rn	0000nnnnmmmm0111	$Rn \times Rm \rightarrow MACL$, $32 \times 32 \rightarrow 32$ bits	2 to 4*	—
MULS.W Rm, Rn	0010nnnnmmmm1111	Signed operation of $Rn \times Rm \rightarrow MAC 16 \times 16 \rightarrow 32$ bits	1 to 3*	—
MULU.W Rm, Rn	0010nnnnmmmm1110	Unsigned operation of $Rn \times Rm \rightarrow MAC 16 \times 16 \rightarrow 32$ bits	1 to 3*	—
NEG Rm, Rn	0110nnnnmmmm1011	$0 - Rm \rightarrow Rn$	1	—
NEGC Rm, Rn	0110nnnnmmmm1010	$0 - Rm - T \rightarrow Rn$, Borrow $\rightarrow T$	1	Borrow
SUB Rm, Rn	0011nnnnmmmm1000	$Rn - Rm \rightarrow Rn$	1	—

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
SUBC	Rm, Rn	0011nnnnmmmm1010	Rn – Rm – T → Rn, Borrow → T	1	Borrow
SUBV	Rm, Rn	0011nnnnmmmm1011	Rn – Rm → Rn, Underflow → T	1	Overflow

Note: * The normal minimum number of execution cycles. (The number in parentheses is the number of cycles when there is contention with following instructions.)

Table 6.5 Logic Operation Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
AND	Rm, Rn	0010nnnnmmmm1001	Rn & Rm → Rn	1	—
AND	#imm, R0	11001001iiiiiiii	R0 & imm → R0	1	—
AND.B	#imm, @(R0, GBR)	11001101iiiiiiii	(R0 + GBR) & imm → (R0 + GBR)	3	—
NOT	Rm, Rn	0110nnnnmmmm0111	~ Rm → Rn	1	—
OR	Rm, Rn	0010nnnnmmmm1011	Rn Rm → Rn	1	—
OR	#imm, R0	11001011iiiiiiii	R0 imm → R0	1	—
OR.B	#imm, @(R0, GBR)	11001111iiiiiiii	(R0 + GBR) imm → (R0 + GBR)	3	—
TAS.B	@Rn	0100nnnn00011011	If (Rn) is 0, 1 → T; 1 → MSB of (Rn)	4	Test result
TST	Rm, Rn	0010nnnnmmmm1000	Rn & Rm; if the result is 0, 1 → T	1	Test result
TST	#imm, R0	11001000iiiiiiii	R0 & imm; if the result is 0, 1 → T	1	Test result
TST.B	#imm, @(R0, GBR)	11001100iiiiiiii	(R0 + GBR) & imm; if the result is 0, 1 → T	3	Test result
XOR	Rm, Rn	0010nnnnmmmm1010	Rn ^ Rm → Rn	1	—
XOR	#imm, R0	11001010iiiiiiii	R0 ^ imm → R0	1	—
XOR.B	#imm, @(R0, GBR)	11001110iiiiiiii	(R0 + GBR) ^ imm → (R0 + GBR)	3	—

Table 6.6 Shift Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
ROTL	Rn	0100nnnn00000100	$T \leftarrow Rn \leftarrow MSB$	1	MSB
ROTR	Rn	0100nnnn00000101	$LSB \rightarrow Rn \rightarrow T$	1	LSB
ROTCL	Rn	0100nnnn00100100	$T \leftarrow Rn \leftarrow T$	1	MSB
ROTCR	Rn	0100nnnn00100101	$T \rightarrow Rn \rightarrow T$	1	LSB
SHAL	Rn	0100nnnn00100000	$T \leftarrow Rn \leftarrow 0$	1	MSB
SHAR	Rn	0100nnnn00100001	$MSB \rightarrow Rn \rightarrow T$	1	LSB
SHLL	Rn	0100nnnn00000000	$T \leftarrow Rn \leftarrow 0$	1	MSB
SHLR	Rn	0100nnnn00000001	$0 \rightarrow Rn \rightarrow T$	1	LSB
SHLL2	Rn	0100nnnn00001000	$Rn \ll 2 \rightarrow Rn$	1	—
SHLR2	Rn	0100nnnn00001001	$Rn \gg 2 \rightarrow Rn$	1	—
SHLL8	Rn	0100nnnn00011000	$Rn \ll 8 \rightarrow Rn$	1	—
SHLR8	Rn	0100nnnn00011001	$Rn \gg 8 \rightarrow Rn$	1	—
SHLL16	Rn	0100nnnn00101000	$Rn \ll 16 \rightarrow Rn$	1	—
SHLR16	Rn	0100nnnn00101001	$Rn \gg 16 \rightarrow Rn$	1	—

Table 6.7 Branch Instructions

Instruction	Instruction Code	Operation	Execution Cycles	T Bit
BF label	100010111ddddddd	If T = 0, disp × 2 + PC → PC; if T = 1, nop	3/1*	—
BF/S label	100011111ddddddd	Delayed branch, if T = 0, disp × 2 + PC → PC; if T = 1, nop	3/1*	—
BT label	100010011ddddddd	If T = 1, disp × 2 + PC → PC; if T = 0, nop	3/1*	—
BT/S label	100011011ddddddd	Delayed branch, if T = 1, disp × 2 + PC → PC; if T = 0, nop	2/1*	—
BRA label	1010100000000000	Delayed branch, disp × 2 + PC → PC	2	—
BRAF Rm	0000mmmm00100011	Delayed branch, Rm + PC → PC	2	—
BSR label	1011000000000000	Delayed branch, PC → PR, disp × 2 + PC → PC	2	—
BSRF Rm	0000mmmm00000011	Delayed branch, PC → PR, Rm + PC → PC	2	—
JMP @Rm	0100mmmm00101011	Delayed branch, Rm → PC	2	—
JSR @Rm	0100mmmm00001011	Delayed branch, PC → PR, Rm → PC	2	—
RTS	0000000000001011	Delayed branch, PR → PC	2	—

Note: * One state when the program does not branch.

Table 6.8 System Control Instructions

Instruction	Instruction Code	Operation	Execution Cycles	T Bit
CLRT	0000000000001000	0 → T	1	0
CLRMAC	000000000101000	0 → MACH, MACL	1	—
LDC Rm, SR	0100mmmm00001110	Rm → SR	1	LSB
LDC Rm, GBR	0100mmmm00011110	Rm → GBR	1	—
LDC Rm, VBR	0100mmmm00101110	Rm → VBR	1	—
LDC.L @Rm+, SR	0100mmmm00000111	(Rm) → SR, Rm + 4 → Rm	3	LSB
LDC.L @Rm+, GBR	0100mmmm00010111	(Rm) → GBR, Rm + 4 → Rm	3	—
LDC.L @Rm+, VBR	0100mmmm00100111	(Rm) → VBR, Rm + 4 → Rm	3	—
LDS Rm, MACH	0100mmmm00001010	Rm → MACH	1	—
LDS Rm, MACL	0100mmmm00011010	Rm → MACL	1	—
LDS Rm, PR	0100mmmm00101010	Rm → PR	1	—
LDS.L @Rm+, MACH	0100mmmm00000110	(Rm) → MACH, Rm + 4 → Rm	1	—
LDS.L @Rm+, MACL	0100mmmm00010110	(Rm) → MACL, Rm + 4 → Rm	1	—
LDS.L @Rm+, PR	0100mmmm00100110	(Rm) → PR, Rm + 4 → Rm	1	—
NOP	0000000000001001	No operation	1	—
RTE	000000000101011	Delayed branch, stack area → PC/SR	4	—
SETT	000000000011000	1 → T	1	1
SLEEP	000000000011011	Sleep	3*	—
STC SR, Rn	0000nnnn00000010	SR → Rn	1	—
STC GBR, Rn	0000nnnn00010010	GBR → Rn	1	—
STC VBR, Rn	0000nnnn00100010	VBR → Rn	1	—
STC.L SR, @-Rn	0100nnnn00000011	Rn - 4 → Rn, SR → (Rn)	2	—
STC.L GBR, @-Rn	0100nnnn00010011	Rn - 4 → Rn, GBR → (Rn)	2	—
STC.L VBR, @-Rn	0100nnnn00100011	Rn - 4 → Rn, BR → (Rn)	2	—
STS MACH, Rn	0000nnnn00001010	MACH → Rn	1	—
STS MACL, Rn	0000nnnn00011010	MACL → Rn	1	—
STS PR, Rn	0000nnnn00101010	PR → Rn	1	—

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
STS.L	MACH,@-Rn	0100nnnn00000010	Rn - 4 → Rn, MACH → (Rn)	1	—
STS.L	MACL,@-Rn	0100nnnn00010010	Rn - 4 → Rn, MACL → (Rn)	1	—
STS.L	PR,@-Rn	0100nnnn00100010	Rn - 4 → Rn, PR → (Rn)	1	—
TRAPA	#imm	11000011iiiiiiii	PC/SR → stack area, imm × 4 + VBR → PC	8	—

Note: * The number of execution cycles before the chip enters sleep mode: 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.

Table 6.9 Floating-Point Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
FABS	FRn	1111nnnn01011101	FRn → FRn	1	—
FADD	FRm, FRn	1111nnnnmmmm0000	FRn + FRm → FRn	1	—
FCMP/EQ	FRm, FRn	1111nnnnmmmm0100	(FRn = FRm)? 1:0 → T	1	Comparison result
FCMP/GT	FRm, FRn	1111nnnnmmmm0101	(FRn > FRm)? 1:0 → T	1	Comparison result
FDIV	FRm, FRn	1111nnnnmmmm0011	FRn/FRm → FRn	13	—
FLDI0	FRn	1111nnnn10001101	0x00000000 → FRn	1	—
FLDI1	FRn	1111nnnn10011101	0x3F800000 → FRn	1	—
FLDS	FRm, FPUL	1111mmmm00011101	FRm → FPUL	1	—
FLOAT	FPUL, FRn	1111nnnn00101101	(float) FPUL → FRn	1	—
FMAC	FR0, FRm, FRn	1111nnnnmmmm1110	FR0 × FRm + FRn → FRn	1	—
FMOV	FRm, FRn	1111nnnnmmmm1100	FRm → FRn	1	—
FMOV.S	@(R0, Rm), FRn	1111nnnnmmmm0110	(R0 + Rm) → FRn	1	—
FMOV.S	@Rm+, FRn	1111nnnnmmmm1001	(Rm) → FRn, Rm+ = 4	1	—
FMOV.S	@Rm, FRn	1111nnnnmmmm1000	(Rm) → FRn	1	—
FMOV.S	FRm, @(R0, Rn)	1111nnnnmmmm0111	FRm → (R0 + Rn)	1	—
FMOV.S	FRm, @-Rn	1111nnnnmmmm1011	Rn- = 4, FRm → (Rn)	1	—
FMOV.S	FRm, @Rn	1111nnnnmmmm1010	FRm → (Rn)	1	—
FMUL	FRm, FRn	1111nnnnmmmm0010	FRn × FRm → FRn	1	—
FNEG	FRn	1111nnnn01001101	-FRn → FRn	1	—
FSTS	FPUL, FRn	1111nnnn00001101	FRn → FPUL	1	—
FSUB	FRm, FRn	1111nnnnmmmm0001	FRn - FRm → FRn	1	—
FTRC	FRm, FPUL	1111nnnn00111101	(long) FRm → FPUL	1	—

Table 6.10 FPU-Related CPU Instructions

Instruction		Instruction Code	Operation	Execution Cycles	T Bit
LDS	Rm, FPSCR	0100mmmm01101010	Rm → FPSCR	1	—
LDS	Rm, FPUL	0100mmmm01011010	Rm → FPUL	1	—
LDS.L	@Rm+, FPSCR	0100mmmm01100110	@Rm → FPSCR, Rm+ = 4	1	—
LDS.L	@Rm+, FPUL	0100mmmm01010110	@Rm → FPUL, Rm+ = 4	1	—
STS	FPSCR, Rn	0000nnnn01101010	FPSCR → Rn	1	—
STS	FPUL, Rn	0000nnnn01011010	FPUL → Rn	1	—
STS.L	FPSCR, @-Rn	0100nnnn01100010	Rn- = 4, FPSCR → @Rn	1	—
STS.L	FPUL, @-Rn	0100nnnn01010010	Rn- = 4, FPUL → @Rn	1	—

6.2 Instruction Set in Alphabetical Order

Table 6.11 alphabetically lists the instruction codes and number of execution cycles for each instruction.

Table 6.11 Instruction Set Listed Alphabetically

Instruction	Operation	Code	Cycles	T Bit
ADD #imm, Rn	Rn + imm → Rn	0111nnnniiaiiiiiii	1	—
ADD Rm, Rn	Rn + Rm → Rn	0011nnnnmmmm1100	1	—
ADDC Rm, Rn	Rn + Rm + T → Rn, Carry → T	0011nnnnmmmm1110	1	Carry
ADDV Rm, Rn	Rn + Rm → Rn, Overflow → T	0011nnnnmmmm1111	1	Over-flow
AND #imm, R0	R0 & imm → R0	11001001iiiiiiii	1	—
AND Rm, Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
AND.B #imm, @(R0, GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	3	—
BF label	If T = 0, disp + PC → PC; if T = 1, nop	10001011ddddddd	3/1* ¹	—
BF/S label	If T = 0, disp + PC → PC; if T = 1, nop	10001111ddddddd	2/1* ¹	—
BRA label	Delayed branch, disp + PC → PC	1010ddddddd	2	—
BRAF Rn	Delayed branch, Rn + PC → PC	0000nnnn00100011	2	—
BSR label	Delayed branch, PC → PR, disp + PC → PC	1011ddddddd	2	—
BSRF Rn	Delayed branch, PC → PR, Rn + PC → PC	0000nnnn00000011	2	—
BT label	If T = 1, disp + PC → PC; if T = 0, nop	10001001ddddddd	3/1* ¹	—
BT/S label	If T = 1, disp + PC → PC; if T = 0, nop	10001101ddddddd	2/1* ¹	—
CLRMAC	0 → MACH, MACL	000000000101000	1	—
CLRT	0 → T	000000000001000	1	0

Instruction		Operation	Code	Cycles	T Bit
CMP/EQ	#imm, R0	If R0 = imm, 1 → T	10001000iiiiiii	1	Comparison result
CMP/EQ	Rm, Rn	If Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/GE	Rm, Rn	If Rn ≥ Rm with signed data, 1 → T	0011nnnnmmmm0011	1	Comparison result
CMP/GT	Rm, Rn	If Rn > Rm with signed data, 1 → T	0011nnnnmmmm0111	1	Comparison result
CMP/HI	Rm, Rn	If Rn > Rm with unsigned data,	0011nnnnmmmm0110	1	Comparison result
CMP/HS	Rm, Rn	If Rn ≥ Rm with unsigned data, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/PL	Rn	If Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	If Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
CMP/STR	Rm, Rn	If Rn and Rm have an equivalent byte, 1 → T	0010nnnnmmmm1100	1	Comparison result
DIV0S	Rm, Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q → T	0010nnnnmmmm0111	1	Calculation result
DIV0U		0 → M/Q/T	000000000011001	1	0
DIV1	Rm, Rn	Single-step division (Rn/Rm)	0011nnnnmmmm0100	1	Calculation result
DMULS.L	Rm, Rn	Signed operation of Rn × Rm → MACH, MACL	0011nnnnmmmm1101	2 to 4*2	—
DMULU.L	Rm, Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 to 4*2	—
DT	Rn	Rn – 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result

Instruction	Operation	Code	Cycles	T Bit
EXTS.B Rm, Rn	A byte in Rm is sign-extended → Rn	0110nnnnmmmm1110	1	—
EXTS.W Rm, Rn	A word in Rm is sign-extended → Rn	0110nnnnmmmm1111	1	—
EXTU.B Rm, Rn	A byte in Rm is zero-extended → Rn	0110nnnnmmmm1100	1	—
EXTU.W Rm, Rn	A word in Rm is zero-extended → Rn	0110nnnnmmmm1101	1	—
FABS FRn	FRn → FRn	1111nnnn01011101	1	—
FADD FRm, FRn	FRn + FRm → FRn	1111nnnnmmmm0000	1	—
FCMP/EQ FRm, FRn	(FRn == FRm)? 1:0 → T	1111nnnnmmmm0100	1	Comparison result
FCMP/GT FRm, FRn	(FRn > FRm)? 1:0 → T	1111nnnnmmmm0101	1	Comparison result
FDIV FRm, FRn	FRn/FRm → FRn	1111nnnnmmmm0011	13	—
FLDI0 FRn	H'00000000 → FRn	1111nnnn10001101	1	—
FLDI1 FRn	H'3F800000 → FRn	1111nnnn10011101	1	—
FLDS FRm, FPUL	FRm → FPUL	1111mmmm00011101	1	—
FLOAT FPUL, FRn	(float) FPUL → FRn	1111nnnn00101101	1	—
FMAC FR0, FRm, FRn	FR0 × FRm + FRn → FRn	1111nnnnmmmm1110	1	—
FMOV FRm, FRn	FRm → FRn	1111nnnnmmmm1100	1	—
FMOV.S @(R0, Rm), FRn	(R0 + Rm) → FRn	1111nnnnmmmm0110	1	—
FMOV.S @Rm+, FRn	(Rm) → FRn, Rm + 4 = Rm	1111nnnnmmmm1001	1	—
FMOV.S @Rm, FRn	(Rm) → FRn	1111nnnnmmmm1000	1	—
FMOV.S FRm, @(R0, Rn)	(FRm) → (R0 + Rn)	1111nnnnmmmm0111	1	—
FMOV.S FRm, @-Rn	Rn - 4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	1	—
FMOV.S FRm, @Rn	FRm → (Rn)	1111nnnnmmmm1010	1	—
FMOV.S FRm, FRn	FRn × FRm → FRn	1111nnnnmmmm0010	1	—

Instruction		Operation	Code	Cycles	T Bit
FMUL	FRm, FRn	FRn × FRm → FRn	1111nnnnmmmm0010	1	—
FNEG	FRn	−FRn → FRn	1111nnnn01001101	1	—
FSTS	FPUL, FRn	FPUL → FRn	1111nnnn00001101	1	—
FSUB	FRm, FRn	FRn − FRm → FRn	1111nnnnmmmm0001	1	—
FTRC	FRm, FPUL	(long) FRm → FPUL	1111mmmm00111101	1	—
JMP	@Rm	Delayed branch, Rm → PC	0100nnnn00101011	2	—
JSR	@Rm	Delayed branch, PC → PR, Rm → PC	0100nnnn00001011	2	—
LDC	Rm, GBR	Rm → GBR	0100mmmm00011110	1	—
LDC	Rm, SR	Rm → SR	0100mmmm00001110	1	LSB
LDC	Rm, VBR	Rm → VBR	0100mmmm00101110	1	—
LDC.L	@Rm+, GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	3	—
LDC.L	@Rm+, SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	3	LSB
LDC.L	@Rm+, VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	3	—
LDS	Rm, FPSCR	Rm → FPSCR	0100mmmm01101010	1	—
LDS	Rm, FPUL	Rm → FPUL	0100mmmm01011010	1	—
LDS	Rm, MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm, MACL	Rm → MACL	0100mmmm00011010	1	—
LDS	Rm, PR	Rm → PR	0100mmmm00101010	1	—
LDS.L	@Rm+, FPSCR	@Rm → FPSCR, Rm + 4	0100mmmm01100110	1	—
LDS.L	@Rm+, FPUL	@Rm → FPUL, Rm + 4	0100mmmm01010110	1	—
LDS.L	@Rm+, MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L	@Rm+, MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L	@Rm+, PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—

Instruction	Operation	Code	Cycles	T Bit
MAC.L @Rm+, @Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	3/(2 to 4)*2	—
MAC.W @Rm+, @Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	3/ (2)*2	—
MOV #imm, Rn	imm → Sign extension → Rn	1110nnnniiiiiii	1	—
MOV Rm, Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MOV.B @(disp, GBR), R0	(disp + GBR) → Sign extension → R0	11000100ddddddd	1	—
MOV.B @(disp, Rm), R0	(disp + Rm) → Sign extension → R0	10000100mmmmddd	1	—
MOV.B @(R0, Rm), Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnmmmm1100	1	—
MOV.B @Rm+, Rn	(Rm) → Sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
MOV.B @Rm, Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0000	1	—
MOV.B R0, @(disp, GBR)	R0 → (disp + GBR)	11000000ddddddd	1	—
MOV.B R0, @(disp, Rn)	R0 → (disp + Rn)	10000000nnnndddd	1	—
MOV.B Rm, @(R0, Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.B Rm, @-Rn	Rn - 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
MOV.B Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.L @(disp, GBR), R0	(disp × 4 + GBR) → R0	11000110ddddddd	1	—
MOV.L @(disp, PC), Rn	(disp × 4 + PC) → Rn	1101nnnndddddddd	1	—
MOV.L @(disp, Rm), Rn	(disp × 4 + Rm) → Rn	0101nnnnmmmmddd	1	—
MOV.L @(R0, Rm), Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—
MOV.L @Rm+, Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—

Instruction	Operation	Code	Cycles	T Bit
MOV.L @Rm, Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—
MOV.L R0, @(disp, GBR)	R0 → (disp × 4 + GBR)	11000010ddddddd	1	—
MOV.L Rm, @(disp, Rn)	Rm → (disp × 4 + Rn)	0001nnnnmmmmddd	1	—
MOV.L Rm, @(R0, Rn)	Rm → (R0 × 4 + Rn)	0000nnnnmmmm0110	1	—
MOV.L Rm, @-Rn	Rn - 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—
MOV.L Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0010	1	—
MOV.W @(disp, GBR), R0	(disp × 2 + GBR) → Sign extension → R0	11000101ddddddd	1	—
MOV.W @(disp, PC), Rn	(disp × 2 + PC) → Sign extension → Rn	1001nnnnddddddd	1	—
MOV.W @(disp, Rm), R0	(disp × 2 + Rm) → Sign extension → R0	10000101mmmmddd	1	—
MOV.W @(R0, Rm), Rn	(R0 + Rm) → Sign extension → Rn	0000nnnnmmmm1101	1	—
MOV.W @Rm+, Rn	(Rm) → Sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
MOV.W @Rm, Rn	(Rm) → Sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.W R0, @(disp, GBR)	R0 → (disp × 2 + GBR)	11000001ddddddd	1	—
MOV.W R0, @(disp, Rn)	R0 → (disp × 2 + Rn)	10000001nnnndddd	1	—
MOV.W Rm, @(R0, Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.W Rm, @-Rn	Rn - 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
MOV.W Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—
MOVA @(disp, PC), R0	disp × 4 + PC → R0	11000111ddddddd	1	—

Instruction		Operation	Code	Cycles	T Bit
MOVT	Rn	$T \rightarrow Rn$	0000nnnn00101001	1	—
MUL.L	Rm, Rn	$Rn \times Rm \rightarrow MAC$	0000nnnnmmmm0111	2 to 4 ^{*2}	—
MULS.W	Rm, Rn	Signed operation of $Rn \times Rm \rightarrow MACL$	0010nnnnmmmm1111	1 to 3 ^{*2}	—
MULU.W	Rm, Rn	Unsigned operation of $Rn \times Rm \rightarrow$ MACL	0010nnnnmmmm1110	1 to 3 ^{*2}	—
NEG	Rm, Rn	$0 - Rm \rightarrow Rn$	0110nnnnmmmm1011	1	—
NEGC	Rm, Rn	$0 - Rm - T \rightarrow Rn$, Borrow $\rightarrow T$	0110nnnnmmmm1010	1	Borrow
NOP		No operation	0000000000001001	1	—
NOT	Rm, Rn	$\sim Rm \rightarrow Rn$	0110nnnnmmmm0111	1	—
OR	#imm, R0	$R0 imm \rightarrow R0$	11001011iiiiiii	1	—
OR	Rm, Rn	$Rn Rm \rightarrow Rn$	0010nnnnmmmm1011	1	—
OR.B	#imm, @(R0, GBR)	$(R0 + GBR) imm$ $\rightarrow (R0 + GBR)$	11001111iiiiiii	3	—
ROTCL	Rn	$T \leftarrow Rn \leftarrow T$	0100nnnn00100100	1	MSB
ROTCR	Rn	$T \rightarrow Rn \rightarrow T$	0100nnnn00100101	1	LSB
ROTL	Rn	$T \leftarrow Rn \leftarrow MSB$	0100nnnn00000100	1	MSB
ROTR	Rn	$LSB \rightarrow Rn \rightarrow T$	0100nnnn00000101	1	LSB
RTE		Delayed branch, SSR/SPC \rightarrow SR/PC	000000000101011	4	LSB
RTS		Delayed branch, PR \rightarrow PC	000000000001011	2	—
SETT		$1 \rightarrow T$	0000000000011000	1	1
SHAL	Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00100000	1	MSB
SHAR	Rn	$MSB \rightarrow Rn \rightarrow T$	0100nnnn00100001	1	LSB
SHLL	Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00000000	1	MSB
SHLL2	Rn	$Rn \ll 2 \rightarrow Rn$	0100nnnn00001000	1	—
SHLL8	Rn	$Rn \ll 8 \rightarrow Rn$	0100nnnn00011000	1	—
SHLL16	Rn	$Rn \ll 16 \rightarrow Rn$	0100nnnn00101000	1	—
SHLR	Rn	$0 \rightarrow Rn \rightarrow T$	0100nnnn00000001	1	LSB
SHLR2	Rn	$Rn \gg 2 \rightarrow Rn$	0100nnnn00001001	1	—

Instruction		Operation	Code	Cycles	T Bit
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—
SLEEP		Sleep	000000000011011	3	—
STC	GBR, Rn	GBR → Rn	0000nnnn00010010	1	—
STC	SR, Rn	SR → Rn	0000nnnn00000010	1	—
STC	VBR, Rn	VBR → Rn	0000nnnn00100010	1	—
STC.L	GBR, @-Rn	Rn - 4 → Rn, GBR → (Rn)	0100nnnn00010011	2	—
STC.L	SR, @-Rn	Rn - 4 → Rn, SR → (Rn)	0100nnnn00000011	2	—
STC.L	VBR, @-Rn	Rn - 4 → Rn, VBR → (Rn)	0100nnnn00100011	2	—
STS	FPSCR, Rn	FPSCR → Rn	0000nnnn01101010	1	—
STS	FPUL, Rn	FPUL → Rn	0000nnnn01011010	1	—
STS	MACH, Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL, Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR, Rn	PR → Rn	0000nnnn00101010	1	—
STS.L	FPSCR, @-Rn	Rn - 4 → Rn, FPSCR → @Rn	0100nnnn01100010	1	—
STS.L	FPUL, @-Rn	Rn - 4 → Rn, FPUL → @Rn	0100nnnn01010010	1	—
STS.L	MACH, @-Rn	Rn - 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L	MACL, @-Rn	Rn - 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L	PR, @-Rn	Rn - 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—
SUB	Rm, Rn	Rn - Rm → Rn	0011nnnnmmmm1000	1	—
SUBC	Rm, Rn	Rn - Rm - T → Rn, Borrow → T	0011nnnnmmmm1010	1	Borrow
SUBV	Rm, Rn	Rn - Rm → Rn, Underflow → T	0011nnnnmmmm1011	1	Underflow
SWAP.B	Rm, Rn	Rm → Swap the two lowest-order bytes → Rn	0110nnnnmmmm1000	1	—

Instruction	Operation	Code	Cycles	T Bit
SWAP.W Rm, Rn	Rm → Swap two consecutive words → Rn	0110nnnnnmmmm1001	1	—
TAS.B @Rn	If (Rn) is 0, 1 → T; 1 → MSB of (Rn)	0100nnnn00011011	4	Test result
TST #imm, R0	R0 & imm; if the result is 0, 1 → T	11001000iiiiiii	1	Test result
TST Rm, Rn	Rn & Rm; if the result is 0, 1 → T	0010nnnnmmmm1000	1	Test result
TST.B #imm, @(R0, GBR)	(R0 + GBR) & imm; if the result is 0, 1 → T	11001100iiiiiii	3	Test result
XOR #imm, R0	R0 ^ imm → R0	11001010iiiiiii	1	—
XOR Rm, Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XOR.B #imm, @(R0, GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiii	3	—
XTRCT Rm, Rn	Rm: Middle 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

Notes: 1. The normal minimum number of execution cycles.
2. One state when it does not branch.

Section 7 Instruction Descriptions

7.1 Sample Description (Name): Classification

This section describes instructions in alphabetical order using the format shown below in section 7.1.1. The actual descriptions begin at section 7.2.2.

Class: Indicates if the instruction is a delayed branch instruction or interrupt disabled instruction

Format	Abstract	Code	Cycle	T Bit
Assembler input format; imm and disp are numbers, expressions, or symbols	A brief description of operation	Displayed in order MSB ↔ LSB	Number of cycles when there is no wait state	The value of T bit after the instruction is executed

Description: Description of operation

Notes: Notes on using the instruction

Operation: Operation written in C language. The following resources should be used.

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

```
unsigned char    Read_Byte(unsigned long Addr);
unsigned short  Read_Word(unsigned long Addr);
unsigned long    Read_Long(unsigned long Addr);
```

- Writes data of each length to address Addr. An address error will occur if word data is written to an address other than 2n or if longword data is written to an address other than 4n:

```
unsigned char    Write_Byte(unsigned long Addr, unsigned long Data);
unsigned short   Write_Word(unsigned long Addr, unsigned long Data);
unsigned long    Write_Long(unsigned long Addr, unsigned long Data);
```

- Starts execution from the slot instruction located at an address (Addr – 4). For Delay_Slot (4), execution starts from an instruction at address 0 rather than address 4. When execution moves from this function to one of the following instructions and one of the listed instructions precedes it, it will be considered an illegal slot instruction (the listed instructions become illegal slot instructions when used as delay slot instructions):

BF, BT, BRA, BSR, JMP, JSR, RTS, RTE, TRAPA, BF/S, BT/S, BRAF, BSRF

```
Delay_Slot(unsigned long Addr);
```

If the address (Addr_4) instruction is 32-bit, 2 is returned; 0 is returned if it is 16-bit.

- List registers:

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

- Definition of SR structures:

```
struct SR0 {  
    unsigned long dummy0:4;  
    unsigned long RC0:12;  
    unsigned long dummy1:4;  
    unsigned long DMY0:1;  
    unsigned long DMX0:1;  
    unsigned long M0:1;  
    unsigned long Q0:1;  
    unsigned long I0:4;  
    unsigned long RF10:1;  
    unsigned long RF00:1;  
    unsigned long S0:1;  
    unsigned long T0:1;  
};
```

- Definition of bits in SR:

```
#define M ((*(struct SRO *)(&SR)).M0)
#define Q ((*(struct SRO *)(&SR)).Q0)
#define S ((*(struct SRO *)(&SR)).S0)
#define T ((*(struct SRO *)(&SR)).T0)
#define RF1 ((*struct SRO *)(&SR)).RF10)
#define RF0 ((*struct SRO *)(&SR)).RF00)
```

- Error display function:

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

The PC should point to the location four bytes after the current instruction. Therefore, `PC = 4;` means the instruction starts execution from address 0, not address 4.

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

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

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

Notes: 1. In addressing modes that use the displacements listed below (disp), the assembler statements in this manual show the value prior to scaling ($\times 1$, $\times 2$, and $\times 4$) according to the operand size. This is done to clarify the LSI operation. Actual assembler statements should follow the rules of the assembler in question.

`@(disp:4, Rn)`; Indirect register addressing with displacement

`@(disp:8, GBR)`; Indirect GBR addressing with displacement

`@(disp:8, PC)`; Indirect PC addressing with displacement

`disp:8, disp:12;`; PC relative addressing

2. 16-bit instruction code that is not assigned as instructions is handled as an ordinary illegal instruction and produces illegal instruction exception processing.
Also, if the FPU is put into stop status by the module stop bit, floating-point instructions and FPU-related CPU instructions are handled as illegal instructions.
3. An ordinary illegal instruction or branched instruction (i.e., an illegal slot instruction) that follows a BRA, BT/S or another delayed branch instruction will cause illegal instruction exception processing.

Example 1:

```
....  
BRA      LABEL  
.data.w  H'FFFF ← Illegal slot instruction  
....      [H'FFFF is an ordinary illegal instruction from the start]
```

Example 2:

```
RTE  
BT/S     LABEL ← Illegal slot instruction
```

7.2 CPU Instruction

7.2.1 ADD (ADD Binary): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
ADD Rm,Rn	$Rm + Rn \rightarrow Rn$	0011nnnnmmmm1100	1	—
ADD #imm,Rn	$Rn + imm \rightarrow Rn$	0111nnnniiiiiii	1	—

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]+=(0xFFFFFFFF0 | (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'FFFFFFF

```

7.2.2 ADDC (ADD with Carry): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
ADDC Rm,Rn	$Rn + Rm + T \rightarrow Rn, \text{carry} \rightarrow T$	0011nnnnmmmm1110	1	Carry

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:

```
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'FFFFFFF
              ;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
```


7.2.3 ADDV (ADD with V Flag Overflow Check): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
ADDV Rm,Rn	$Rn + Rm \rightarrow Rn$, overflow $\rightarrow T$	0011nnnnnmmmm1111	1	Overflow

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

```

7.2.4 AND (AND Logical): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
AND Rm,Rn	$Rn \ \& \ Rm \rightarrow Rn$	0010nnnnmmmm1001	1	—
AND #imm,R0	$R0 \ \& \ imm \rightarrow R0$	11001001iiiiiiii	1	—
AND.B #imm,@(R0,GBR)	$(R0 + GBR) \ \& \ imm \rightarrow (R0 + GBR)$	11001101iiiiiiii	3	—

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
```

7.2.5 BF (Branch if False): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BF label	When T = 0, $\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$; When T = 1, nop	10001011ddddddd	3/1	—

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:

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

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFFFF00 | (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
```

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

Format	Abstract	Code	Cycle	T Bit
BF/S label	When T = 0, $\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$; When T = 1, nop	10001111ddddddd	2/1	—

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.

Operation:

```

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

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFFFF00 | (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.

7.2.7 BRA (Branch): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BRA label	$\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$	1010ddddddddddd	2	—

Description: Branches unconditionally after executing the instruction following this BRA instruction. The branch destination is an address specified by $\text{PC} + \text{displacement}$. 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:

```
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.

7.2.8 BRAF (Branch Far): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BRAF Rm	Rm + PC → PC	0000mmmm00100011	2	—

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:

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

    temp=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.

7.2.9 BSR (Branch to Subroutine): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BSR label	PC → PR, disp × 2+ PC → PC	1011ddddddddddd	2	—

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:

```
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.

7.2.10 BSRF (Branch to Subroutine Far): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BSRF Rm	PC → PR, Rm + PC → PC	0000mmmm00000011	2	—

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+Is_32bit_Inst(PR+2);
    PC+=R[m];
    Delay_Slot(PR+2);
}
```

Example:

```

    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
    . . . . .
    . . . . .
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 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.

7.2.11 BT (Branch if True): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BT label	When T = 1, $\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$; When T = 0, nop	10001001ddddddd	3/1	—

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:

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

    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFFFF00 | (long)d);
    if (T==1) PC=PC+(disp<<1);
    else PC+=2;
}
```

Example:

```
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
.....             address of the BT instruction
TRGET_T:           ;← Branch destination of the BT instruction
```

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

Format	Abstract	Code	Cycle	T Bit
BT/S label	When T = 1, $\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$; When T = 0, nop	10001101ddddddd	2/1	—

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=(0xFFFFFFFF00 | (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
. . . . .          ; address of the BT/S instruction
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.

7.2.13 CLRMAC (Clear MAC Register): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRMAC	0 → MACH, MACL	0000000000101000	1	—

Description: Clear the MACH and MACL Register.

Operation:

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

Example:

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

7.2.14 CLRT (Clear T Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRT	$0 \rightarrow T$	0000000000001000	1	0

Description: Clears the T bit.

Operation:

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

Example:

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

7.2.15 CMP/cond (Compare Conditionally): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit	
CMP/EQ	Rm,Rn	When $Rn = Rm$, $1 \rightarrow T$	0011nnnnmmmm0000	1	Comparison result
CMP/GE	Rm,Rn	When signed and $Rn \geq Rm$, $1 \rightarrow T$	0011nnnnmmmm0011	1	Comparison result
CMP/GT	Rm,Rn	When signed and $Rn > Rm$, $1 \rightarrow T$	0011nnnnmmmm0111	1	Comparison result
CMP/HI	Rm,Rn	When unsigned and $Rn > Rm$, $1 \rightarrow T$	0011nnnnmmmm0110	1	Comparison result
CMP/HS	Rm,Rn	When unsigned and $Rn \geq Rm$, $1 \rightarrow T$	0011nnnnmmmm0010	1	Comparison result
CMP/PL	Rn	When $Rn > 0$, $1 \rightarrow T$	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	When $Rn \geq 0$, $1 \rightarrow T$	0100nnnn00010001	1	Comparison result
CMP/STR	Rm,Rn	When a byte in Rn equals a byte in Rm, $1 \rightarrow T$	0010nnnnmmmm1100	1	Comparison result
CMP/EQ	#imm,R0	When $R0 = imm$, $1 \rightarrow T$	10001000iiiiiii	1	Comparison result

Description: Compares general register Rn data with Rm data, and sets the T bit to 1 if a specified condition (cond) is satisfied. The T bit is cleared to 0 if the condition is not satisfied. 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 7.2 shows the mnemonics for the conditions.

Table 7.2 CMP Mnemonics

Mnemonics	Condition
CMP/EQ Rm,Rn	If $Rn = Rm$, $T = 1$
CMP/GE Rm,Rn	If $Rn \geq Rm$ with signed data, $T = 1$
CMP/GT Rm,Rn	If $Rn > Rm$ with signed data, $T = 1$
CMP/HI Rm,Rn	If $Rn > Rm$ with unsigned data, $T = 1$
CMP/HS Rm,Rn	If $Rn \geq Rm$ with unsigned data, $T = 1$
CMP/PL Rn	If $Rn > 0$, $T = 1$
CMP/PZ Rn	If $Rn \geq 0$, $T = 1$
CMP/STR Rm,Rn	If a byte in Rn equals a byte in Rm, $T = 1$
CMP/EQ #imm,R0	If $R0 = imm$, $T = 1$

Operation:

```

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

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

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

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

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

```

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

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

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

    temp=R[n]^R[m];
    HH=(temp>>12)&0x000000FF;
    HL=(temp>>8)&0x000000FF;
    LH=(temp>>4)&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=(0xFFFFFFFF | (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 = "XYZZ"
BT        TRGET_T      ;Branches because T = 1

```

7.2.16 DIV0S (Divide Step 0 as Signed): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DIV0S Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M^Q → T	0010nnnnmmmm0111	1	Calculation result

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

Operation:

```

DIV0S(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.

7.2.17 DIV0U (Divide Step 0 as Unsigned): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DIV0U	0 → M/Q/T	0000000000011001	1	0

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

Operation:

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

Example: See DIV1.

7.2.18 DIV1 (Divide 1 Step): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DIV1 Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnnnmmmm0100	1	Calculation result

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;
}
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;
}
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            ;
ROTCL     R3                ;
SUBC      R2, R1            ; Decrements if the dividend is negative
DIV0S     R0, R1            ; Flag initialization
.repeat   16                ;
DIV1      R0, R1            ; Repeat 16 times
.aendr
EXTS.W    R1, R1            ;
ROTCL     R1                ; R1 = quotient (one's complement)
ADDC      R2, R1            ; Increments and takes the two's complement if the MSB of the quotient
is 1
EXTS.W    R1, R1            ; R1 = quotient (two's complement)

```

Example 4:

```

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

```

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

Format	Abstract	Code	Cycle	T Bit
DMULS.L Rm, Rn	With sign, $R_n \times R_m \rightarrow \text{MACH}, \text{MACL}$	0011nnnnmmmm1101	2 to 4	—

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:

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

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

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

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

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

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

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

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

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

Example:

```
DMULS.L R0,R1      ;Before execution: R0 = H'FFFFFFFE, R1 = H'00005555
                   ;After execution:  MACH = H'FFFFFFF, MACL = H'FFFF5556

STS      MACH,R0    ;Operation result (top)
STS      MACL,R0    ;Operation result (bottom)
```


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

Format	Abstract	Code	Cycle	T Bit
DMULU.L Rm, Rn	Without sign, $R_n \times R_m \rightarrow MACH, MACL$	0011nnnnnnmmmm0101	2 to 4	—

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:

```
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'FFFFFFF, MACL = H'FFFF5556  
STS      MACH,R0    ; Operation result (top)  
STS      MACL,R0    ; Operation result (bottom)
```

7.2.21 DT (Decrement and Test): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
DT Rn	$Rn - 1 \rightarrow Rn$; When Rn is 0, $1 \rightarrow T$, when Rn is nonzero, $0 \rightarrow T$	0100nnnn00010000	1	Comparison result

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:

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

Example:

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

7.2.22 EXTS (Extend as Signed): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
EXTS.B Rm, Rn	Sign-extend Rm from byte → Rn	0110nnnnmmmm1110	1	—
EXTS.W Rm, Rn	Sign-extend Rm from word → Rn	0110nnnnmmmm1111	1	—

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:

```
EXTSB(long m, long n)    /* EXTS.B Rm,Rn */
{
    R[n]=R[m];
    if ((R[m]&0x00000080)==0) R[n]&=0x000000FF;
    else R[n]|=0xFFFFF00;
    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'00000080
                        ; After execution:   R1 = H'FFFFFF80

EXTS.W R0, R1           ; Before execution: R0 = H'00008000
                        ; After execution:   R1 = H'FFFF8000
```

7.2.23 EXTU (Extend as Unsigned): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
EXTU.B Rm, Rn	Zero-extend Rm from byte → Rn	0110nnnnmmmm1100	1	—
EXTU.W Rm, Rn	Zero-extend Rm from word → Rn	0110nnnnmmmm1101	1	—

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:

```
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
```

7.2.24 JMP (Jump): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
JMP @Rm	Rm → PC	0100mmmm00101011	2	—

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:

```
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
.....
TRGET:    ADD      #1, R1    ; ← 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.

7.2.25 JSR (Jump to Subroutine): Branch Instruction (Class: Delayed Branch Instruction)

Format	Abstract	Code	Cycle	T Bit
JSR @Rm	PC → PR, Rm → PC	0100mmmm00001011	2	—

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:

```
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.

7.2.26 LDC (Load to Control Register): System Control Instruction (Class: Interrupt Disabled Instruction)

Format	Abstract	Code	Cycle	T Bit
LDC Rm,SR	Rm → SR	0100mmmm00001110	1	LSB
LDC Rm,GBR	Rm → GBR	0100mmmm00011110	1	—
LDC Rm,VBR	Rm → VBR	0100mmmm00101110	1	—
LDC.L @Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	3	LSB
LDC.L @Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	3	—
LDC.L @Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	3	—

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

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

Operation:

```
LDCSR(long m) /* LDC Rm,SR */
```

```
{
    SR=R[m]&0x0FFF0FFF;
    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])&0xFFFF0FFF;
    R[m]+=4;
    PC+=2;
}

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'0FFF0FFF

LDC.L  @R15+,GBR  ;Before execution: R15 = H'10000000
                ;After execution:   R15 = H'10000004, GBR = @H'10000000

```

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

Class: Interrupt disabled instruction

Format	Abstract	Code	Cycle	T Bit
LDS Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS Rm,MACL	Rm → MACL	0100mmmm00011010	1	—
LDS Rm,PR	Rm → PR	0100mmmm00101010	1	—
LDS.L @Rm+, MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L @Rm+, MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L @Rm+, PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—

Description: Store the source operand into the system register MACH, MACL, or PR.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

Operation:

```

LDSMACH(long m)          /* LDS Rm,MACH */
{
    MACH=R[m];
    PC+=2;
}
LDSMACL(long m)         /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}
LDSRPR(long m)          /* LDS Rm,PR */
{
    PR=R[m];
    PC+=2;
}
LDSMMACH(long m)        /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);

```

```

    R[m] += 4;
    PC += 2;
}
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

```

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

Format	Abstract	Code	Cycle	T Bit
MAC.L @Rm+, @Rn+	Signed operation, (Rn) × (Rm) + MAC → MAC	0000nnnnnmmmm1111	3/(2 to 4)	—

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 (fnLm<0){  
    Res2=~Res2;  
    if (Res0==0) Res2++;  
    else Res0=(~Res0)+1;  
}
```

```
if (S==1){  
    Res0=MACL+Res0;  
    if (MACL>Res0) Res2++;  
    Res2+=(MACH&0x0000FFFF);
```

```
if (((long)Res2<0)&&(Res2<0xFFFF8000)){  
    Res2=0x00008000;  
    Res0=0x00000000;  
}
```

```

    if ( ((long)Res2>0)&&(Res2>0x00007FFF) ) {
        Res2=0x00007FFF;
        Res0=0xFFFFFFFF;
    };

    MACH={Res2;
    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.1  H'1234ABCD ;
      .data.1  H'5678EF01 ;
TBLN  .data.1  H'0123ABCD ;
      .data.1  H'4567DEF0 ;

```


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

Format	Abstract	Code	Cycle	T Bit
MAC.W @Rm+, @Rn+	With sign, $(Rn) \times (Rm) + MAC \rightarrow MAC$	0100nnnnmmmm1111	3/(2)	—
MAC @Rm+, @Rn+				

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 LSB of the MACH register is set to 1. 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:

```
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+=tempn;
if ((long)MACL>=0) ans=0;
else ans=1;
ans+=dest;
if (S==1) {
    if (ans==1) {
        if (src==0) MACL=0x7FFFFFFF;
        if (src==2) MACL=0x80000000;
    }
}
else {
    MACH+=tempn;
    if (templ>MACL) MACH+=1;
}
PC+=2;
}
```

Example:

```
MOVA      TBLM,R0      ; 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      ;
```

7.2.30 MOV (Move Data): Data Transfer Instruction

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

Description: 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]|=0xFFFFFFFF;
    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]|=0FFFFFF0;
    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]|=0xFFFFFFFF0;
    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'FFFFF80
MOV.W R0,@-R1	; Before execution: R0 = H'AAAAAAAA, R1 = H'FFFF7F80
	; After execution: R1 = H'FFFF7F7E, @R1 = H'AAAA
MOV.L @R0+,R1	; Before execution: R0 = H'12345670
	; After execution: R0 = H'12345674, R1 = @H'12345670
MOV.B R1,@(R0,R2)	; Before execution: R2 = H'00000004, R0 = H'10000000
	; After execution: R1 = @H'10000004
MOV.W @(R0,R2),R1	; Before execution: R2 = H'00000004, R0 = H'10000000
	; After execution: R1 = @H'10000004

7.2.31 MOV (Move Immediate Data): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
MOV #imm,Rn	imm → sign extension → Rn	1110nnnniiiiiii	1	—
MOV.W @(disp, PC),Rn	(disp × 2 + PC) → sign extension → Rn	1001nnnnddddddd	1	—
MOV.L @(disp, PC),Rn	(disp × 4 + PC) → Rn	1101nnnnddddddd	1	—

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 second instruction 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 second instruction 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".

Operation:

```

MOVI(long i,long n)    /* MOV #imm,Rn */
{
    if ((i&0x80)==0) R[n]=(0x000000FF & (long)i);
    else R[n]=(0xFFFFF00 | (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&0xFFFFFFF0)+(disp<<2));
    PC+=2;
}

```

Example:

Address			
1000	MOV	#H'80,R1	;R1 = H'FFFFFF80
1002	MOV.W	IMM,R2	;R2 = H'FFF9ABC, 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
		.align 4	;
1018		.data.l H'12345678	;

7.2.32 MOV (Move Peripheral Data): Data Transfer Instruction

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

Description: 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 7.1 will give better results.

MOV.B @(12, GBR), R0		MOV.B @(12, GBR), R0
AND #80, R0	↘	ADD #20, R1
ADD #20, R1	↗	AND #80, R0

Figure 7.1 Using R0 after MOV

Operation:

```
MOVB LG(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]|=0xFFFFFFFF0;
    PC+=2;
}

MOVW LG(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;
}

MOVL LG(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

```

7.2.33 MOV (Move Structure Data): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
MOV.B R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnnndddd	1	—
MOV.W R0,@(disp,Rn)	R0 → (disp × 2 + Rn)	10000001nnnnndddd	1	—
MOV.L Rm,@(disp,Rn)	Rm → (disp × 4 + Rn)	0001nnnnmmmmndddd	1	—
MOV.B @(disp,Rm),R0	(disp + Rm) → sign extension → R0	10000100mmmmndddd	1	—
MOV.W @(disp,Rm),R0	(disp × 2 + Rm) → sign extension → R0	10000101mmmmndddd	1	—
MOV.L @(disp,Rm),Rn	disp × 4 + Rm → Rn	0101nnnnmmmmndddd	1	—

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 7.2 will give better results.

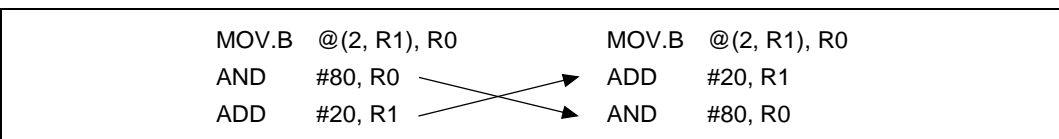


Figure 7.2 Using R0 after MOV

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]|=0xFFFFF00;
}
```

```
    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:

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

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


7.2.34 MOVA (Move Effective Address): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
MOVA @(disp,PC),R0	disp × 4 + PC → R0	11000111111111111111111111111111	1	—

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.

Operation:

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

    disp=(0x000000FF & (long)d);
    R[0]=(PC&0xFFFFFFFF)+(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                    ;
```

7.2.35 MOVT (Move T Bit): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
MOVT Rn	T → Rn	0000nnnn00101001	1	—

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

Operation:

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

Example:

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

7.2.36 MULL (Multiply Long): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
MULL Rm,Rn	$Rn \times Rm \rightarrow MACL$	0000nnnnmmmm0111	2 to 4	—

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:

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

Example:

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

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

Format	Abstract	Code	Cycle	T Bit
MULS.W Rm,Rn	Signed operation, $Rn \times Rm \rightarrow MACL$	0010nnnnnmmmm1111	1 to 3	—
MULS Rm,Rn				

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

Operation:

```
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
```

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

Format	Abstract	Code	Cycle	T Bit
MULU.W Rm,Rn	Unsigned, $Rn \times Rm \rightarrow MACL$	0010nnnnnnmmmm1110	1 to 3	—
MULU Rm,Rn				

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

Operation:

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

Example:

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

7.2.39 NEG (Negate): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
NEG Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—

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

Operation:

```
NEG(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'FFFFFFF
```

7.2.40 NEGC (Negate with Carry): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
NEGC Rm,Rn	$0 - Rm - T \rightarrow Rn$, Borrow $\rightarrow T$	0110nnnnmmmm1010	1	Borrow

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

Operation:

```
NEGC(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'FFFFFFF, T = 1
NEGC  R0,R0   ; Before execution: R0 = H'00000000, T = 1
           ; After execution:   R0 = H'FFFFFFF, T = 1
```

7.2.41 NOP (No Operation): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
NOP	No operation	0000000000001001	1	—

Description: Increments the PC to execute the next instruction.

Operation:

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

Example:

```
    NOP    ;Executes in one cycle
```


7.2.42 NOT (NOT—Logical Complement): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
NOT Rm,Rn	$\sim Rm \rightarrow Rn$	0110nnnnmmmm0111	1	—

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

Operation:

```
NOT(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
```

7.2.43 OR (OR Logical) Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
OR Rm,Rn	$Rn \mid Rm \rightarrow Rn$	0010nnnnmmmm1011	1	—
OR #imm,R0	$R0 \mid imm \rightarrow R0$	11001011iiiiiii	1	—
OR.B #imm,@(R0,GBR)	$(R0 + GBR) \mid imm \rightarrow (R0 + GBR)$	11001111iiiiiii	3	—

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:

```

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

7.2.44 ROTCL (Rotate with Carry Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTCL Rn	$T \leftarrow Rn \leftarrow T$	0100nnnn00100100	1	MSB

Description: Rotates the contents of general register Rn and the T bit to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 7.3).

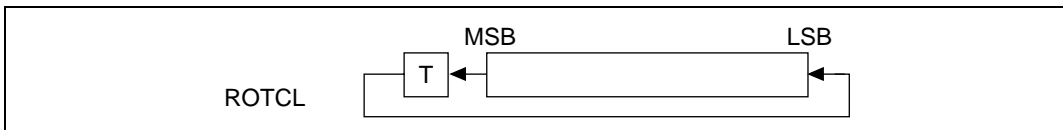


Figure 7.3 Rotate with Carry Left

Operation:

```

ROTCL(long n) /* ROTCL Rn */
{
    long temp;

    if ((R[n]&0x80000000)==0) temp=0;
    else temp=1;
    R[n]<<=1;
    if (T==1) R[n]|=0x00000001;
    else R[n]&=0xFFFFFFFE;
    if (temp==1) T=1;
    else T=0;
    PC+=2;
}

```

Example:

```

ROTCL R0      ; Before execution: R0 = H'80000000, T = 0
              ; After execution:  R0 = H'00000000, T = 1

```

7.2.45 ROTCR (Rotate with Carry Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTCR Rn	$T \rightarrow Rn \rightarrow T$	0100nnnn00100101	1	LSB

Description: Rotates the contents of general register Rn and the T bit to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 7.4).

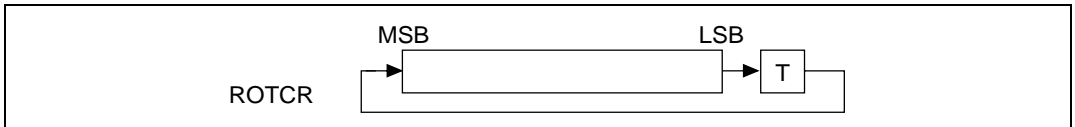


Figure 7.4 Rotate with Carry Right

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
```

7.2.46 ROTL (Rotate Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTL Rn	$T \leftarrow Rn \leftarrow \text{MSB}$	0100nnnn00000100	1	MSB

Description: Rotates the contents of general register Rn to the left by one bit, and stores the result in Rn (figure 7.5). The bit that is shifted out of the operand is transferred to the T bit.

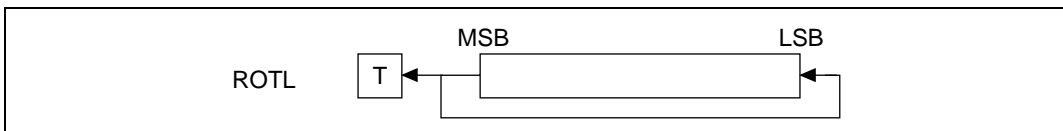


Figure 7.5 Rotate Left

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
```

7.2.47 ROTR (Rotate Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTR Rn	LSB → Rn → T	0100nnnn00000101	1	LSB

Description: Rotates the contents of general register Rn to the right by one bit, and stores the result in Rn (figure 7.6). The bit that is shifted out of the operand is transferred to the T bit.

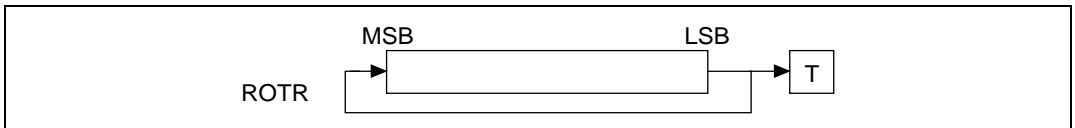


Figure 7.6 Rotate Right

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
```

7.2.48 RTE (Return from Exception): System Control Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
RTE	Delayed branch, Stack area → PC/SR	0000000000101011	4	LSB

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:

```
RTE() /* RTE */
{
    unsigned long temp;

    temp=PC;
    PC=Read_Long(R[15])+4;
    R[15]+=4;
    SR=Read_Long(R[15])&0xFFF0FFF;
    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.

7.2.49 RTS (Return from Subroutine): Branch Instruction (Class: Delayed Branch Instruction)

Format	Abstract	Code	Cycle	T Bit
RTS	Delayed branch, PR → PC	0000000000001011	2	—

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:

```
RTS() /* RTS */
{
    unsigned long temp;

    temp=PC;
    PC=PR+4;
    Delay_Slot(temp+2);
}
```

Example:

```

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   ; Jump table
.....
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.

7.2.50 SETT (Set T Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
SETT	1 → T	0000000000011000	1	1

Description: Sets the T bit to 1.

Operation:

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

Example:

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

7.2.51 SHAL (Shift Arithmetic Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHAL Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00100000	1	MSB

Description: Arithmetically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 7.7).

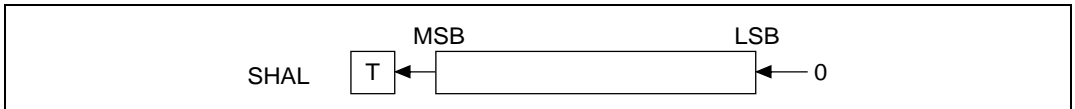


Figure 7.7 Shift Arithmetic Left

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
```

7.2.52 SHAR (Shift Arithmetic Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHAR Rn	MSB → Rn → T	0100nnnn00100001	1	LSB

Description: Arithmetically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 7.8).

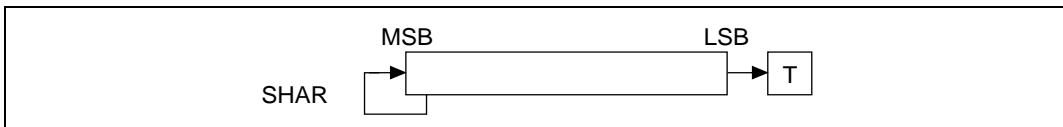


Figure 7.8 Shift Arithmetic Right

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
```

7.2.53 SHLL (Shift Logical Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHLL Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00000000	1	MSB

Description: Logically shifts the contents of general register Rn to the left by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 7.9).

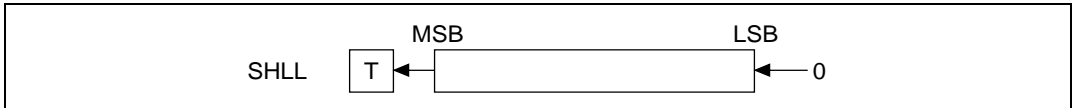


Figure 7.9 Shift Logical Left

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
```

7.2.54 SHLLn (Shift Logical Left n Bits): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHLL2 Rn	$Rn \ll 2 \rightarrow Rn$	0100nnnn00001000	1	—
SHLL8 Rn	$Rn \ll 8 \rightarrow Rn$	0100nnnn00011000	1	—
SHLL16 Rn	$Rn \ll 16 \rightarrow Rn$	0100nnnn00101000	1	—

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 7.10).

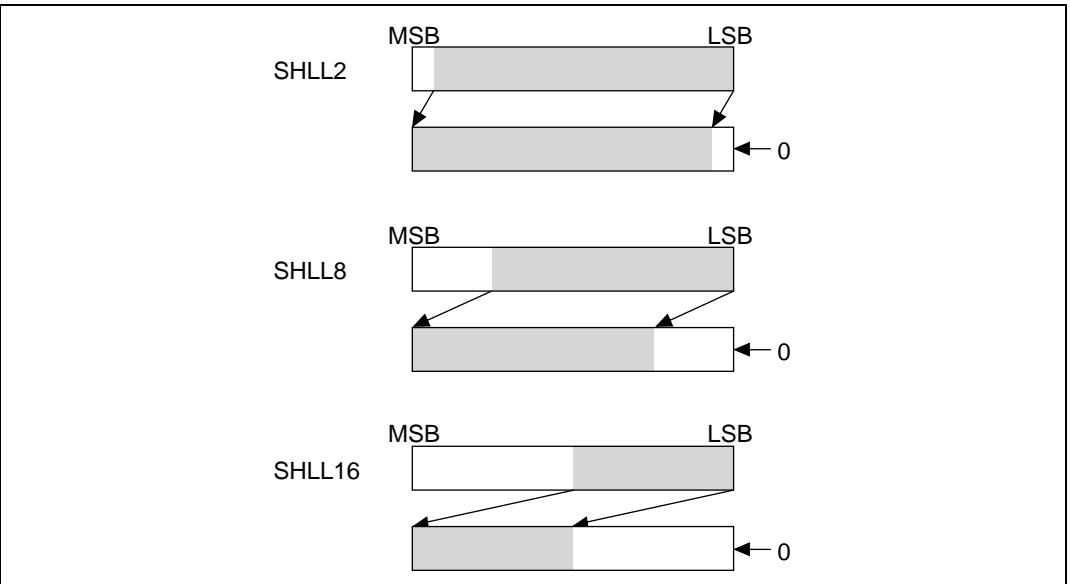


Figure 7.10 Shift Logical Left n Bits

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:

```
SHLL2 R0 ; Before execution: R0 = H'12345678
          ; After execution: R0 = H'48D159E0
```

```
SHLL8 R0 ; Before execution: R0 = H'12345678
          ; After execution: R0 = H'34567800
```

```
SHLL16 R0 ; Before execution: R0 = H'12345678
           ; After execution: R0 = H'56780000
```

7.2.55 SHLR (Shift Logical Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHLR Rn	0 → Rn → T	0100nnnn00000001	1	LSB

Description: Logically shifts the contents of general register Rn to the right by one bit, and stores the result in Rn. The bit that is shifted out of the operand is transferred to the T bit (figure 7.11).

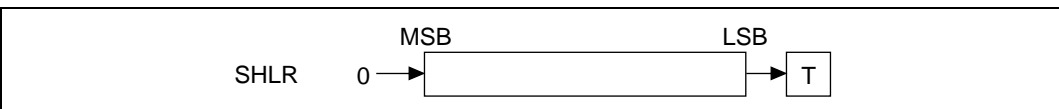


Figure 7.11 Shift Logical Right

Operation:

```
SHLR(long n) /* SHLR Rn */
{
    if ((R[n]&0x00000001)==0) T=0;
    else T=1;
    R[n]>>=1;
    R[n]&=0x7FFFFFFF;
    PC+=2;
}
```

Examples:

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

7.2.56 SHLRn (Shift Logical Right n Bits): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
SHLR2 Rn	$Rn \gg 2 \rightarrow Rn$	0100nnnn00001001	1	—
SHLR8 Rn	$Rn \gg 8 \rightarrow Rn$	0100nnnn00011001	1	—
SHLR16 Rn	$Rn \gg 16 \rightarrow Rn$	0100nnnn00101001	1	—

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 7.12).

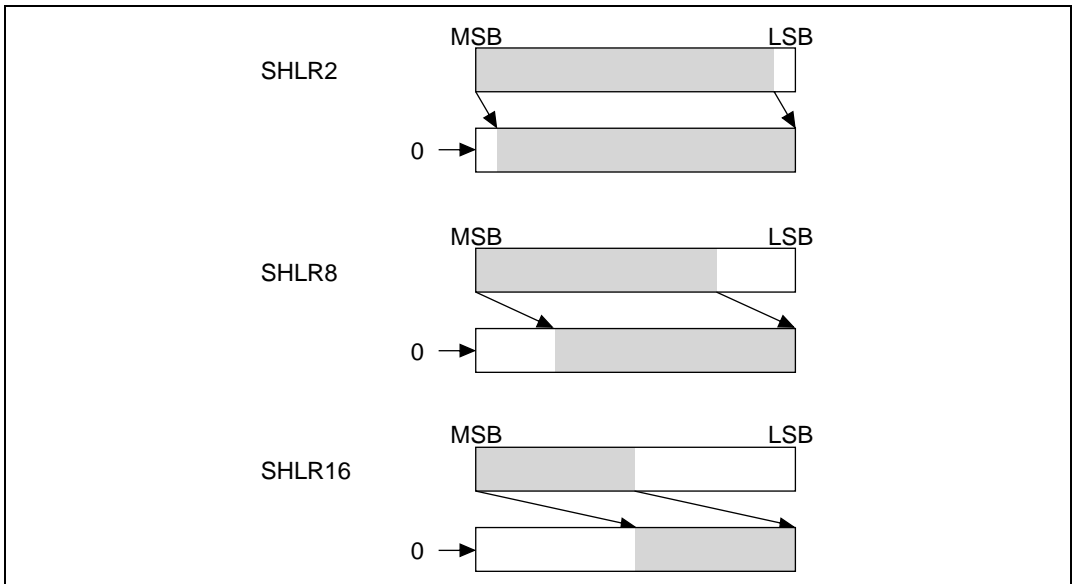


Figure 7.12 Shift Logical Right n Bits

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
```

7.2.57 SLEEP (Sleep): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
SLEEP	Sleep	0000000000011011	3	—

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:

```
SLEEP() /* SLEEP */
{
    PC-=2;
    wait_for_exception;
}
```

Example:

```
SLEEP ; Enters power-down mode
```

7.2.58 STC (Store Control Register): System Control Instruction (Interrupt Disabled Instruction)

Format	Abstract	Code	Cycle	T Bit
STC SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC.L SR,@-Rn	Rn - 4 → Rn, SR → (Rn)	0100nnnn00000011	2	—
STC.L GBR,@-Rn	Rn - 4 → Rn, GBR → (Rn)	0100nnnn00010011	2	—
STC.L VBR,@-Rn	Rn - 4 → Rn, VBR → (Rn)	0100nnnn00100011	2	—

Description: Stores control register SR, GBR, or VBR data into a specified destination.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

Operation:

```

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);
    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'FFFFFFF, SR = H'00000000
                ; After execution:   R0 = H'00000000

STC.L  GBR,@-R15 ; Before execution: R15 = H'10000004
                ; After execution:   R15 = H'10000000, @R15 = GBR

```

7.2.59 STS (Store System Register): System Control Instruction (Interrupt Disabled Instruction)

Format	Abstract	Code	Cycle	T Bit
STS MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS PR,Rn	PR → Rn	0000nnnn00101010	1	—
STS.L MACH,@-Rn	Rn - 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L MACL,@-Rn	Rn - 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L PR,@-Rn	Rn - 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—

Description: Stores data from system register MACH, MACL, or PR into a specified destination.

Note: No interrupts are accepted between this instruction and the next instruction. Address errors are accepted.

Operation:

```
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;
}
```



```

STSMACH(long n) /* STS.L MACH,@-Rn */
{
    R[n]--=4;
    Write_Long(R[n],MACH);
    PC+=2;
}

STSMACL(long n) /* STS.L MACL,@-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'FFFFFFF, MACH = H'0000000
           ; After execution:    R0 = H'0000000

STS.L  PR,@-R15  ; Before execution: R15 = H'10000004
           ; After execution:    R15 = H'10000000, @R15 = PR

```

7.2.60 SUB (Subtract Binary): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
SUB Rm,Rn	$Rn - Rm \rightarrow Rn$	0011nnnnnnmmmm1000	1	—

Description: Subtracts general register Rm data from Rn data, and stores the result in Rn. To subtract immediate data, use ADD #imm,Rn.

Operation:

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

Example:

```
SUB R0,R1 ; Before execution: R0 = H'00000001, R1 = H'80000000  
          ; After execution:   R1 = H'7FFFFFFF
```

7.2.61 SUBC (Subtract with Carry): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
SUBC Rm,Rn	$Rn - Rm - T \rightarrow Rn$, Borrow $\rightarrow T$	0011nnnnnnmmmm1010	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:

```

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'FFFFFF
SUBC  R2,R0   ;Before execution: T = 1, R0 = H'00000000, R2 = H'00000000
          ;After execution:   T = 1, R0 = H'FFFFFF

```

7.2.62 SUBV (Subtract with V Flag Underflow Check): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
SUBV Rm,Rn	$Rn - Rm \rightarrow Rn$, underflow $\rightarrow T$	0011nnnnnnmmmm1011	1	Underflow

Description: Subtracts Rm data from general register Rn data, and stores the result in Rn. If an underflow occurs, the T bit is set to 1.

Operation:

```
SUBV(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
```

7.2.63 SWAP (Swap Register Halves): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
SWAP.B Rm,Rn	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnmmmm1000	1	—
SWAP.W Rm,Rn	Rm → Swap upper and lower word → Rn	0110nnnnmmmm1001	1	—

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 */
{
    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 */
{
    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

7.2.64 TAS (Test and Set): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
TAS.B @Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	4	Test results

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:

```
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
```

7.2.65 TRAPA (Trap Always): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
TRAPA #imm	PC/SR → Stack area, (imm × 4 + VBR) → PC	11000011iiiiiii	8	—

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.

Operation:

```
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.l 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)
.....
.....
10000000 XOR R0,R0 ; ← Trap routine entrance
10000002 RTE ; Returns to the TST instruction
10000004 NOP ; Executes NOP before RTE
```


7.2.66 TST (Test Logical): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
TST Rm,Rn	Rn & Rm, when result is 0, 1 → T	0010nnnnnmmmm1000	1	Test results
TST #imm,R0	R0 & imm, when result is 0, 1 → T	11001000iiiiiii	1	Test results
TST.B #imm, @(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiii	3	Test results

Description: Logically ANDs the contents of general registers Rn and Rm, and sets the T bit to 1 if the result is 0 or clears the T bit to 0 if the result is not 0. The Rn data does not change. The contents of general register R0 can also be ANDed with zero-extended 8-bit immediate data, or the contents of 8-bit memory accessed by indirect indexed GBR addressing can be ANDed with 8-bit immediate data. The R0 and memory data do not change.

Operation:

```
TST(long m,long n) /* TST Rm,Rn */
```

```
{
    if ((R[n]&R[m])==0) T=1;
    else T=0;
    PC+=2;
}
```

```
TSTI(long i) /* 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_Byte(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'FFFFFF7F
		; After execution: T = 1
TST.B	#H'A5,@(R0,GBR)	; Before execution: @(R0,GBR) = H'A5
		; After execution: T = 0

7.2.67 XOR (Exclusive OR Logical): Logic Operation Instruction

Format	Abstract	Code	Cycle	T Bit
XOR Rm,Rn	$Rn \wedge Rm \rightarrow Rn$	0010nnnnmmmm1010	1	—
XOR #imm,R0	$R0 \wedge imm \rightarrow R0$	11001010iiiiiii	1	—
XOR.B #imm, @(R0,GBR)	$(R0 + GBR) \wedge imm \rightarrow (R0 + GBR)$	11001110iiiiiii	3	—

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'FFFFFFF0

XOR.B #H'A5, @(R0, GBR) ; Before execution: @(R0, GBR) = H'A5
; After execution: @(R0, GBR) = H'00

7.2.68 XTRCT (Extract): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
XTRCT Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnnnmmmm1101	1	—

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 7.13).

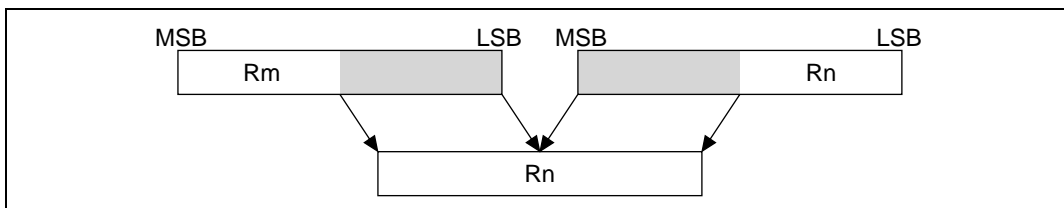


Figure 7.13 Extract

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
```

7.3 Floating Point Instructions and FPU Related CPU Instructions

The functions used in the descriptions of the operation of FPU calculations are as follows.

```
long FPSCR;
int T;

int load_long(long *address, *data)
{
    /* This function is defined in CPU part */
}
int store_long(long *address, *data)
{
    /* This function is defined in CPU part */
}
int sign_of(long *src)
{
    return(*src >> 31);
}
int data_type_of(long *src)
{
    float abs;
    abs = *src & 0x7fffffff;
    if(abs < 0x00800000) {
        if(sign_of (src) == 0) return(PZERO);
        else return(NZERO);
    }
    else if((0x00800000 <= abs) && (abs < 0x7f800000))
        return(NORM);
    else if(0x7f800000 == abs) {
        if(sign_of (src) == 0) return(PINF);
        else return(NINF);
    }
}
```

```
else if(0x00400000 & abs)          return(sNaN);
else                               return(qNaN);
    }
}

clear_cause_VZ(){ FPSCR &= (~CAUSE_V & ~CAUSE_Z); }
set_V(){ FPSCR |= (CAUSE_V | FLAG_V); }
set_Z(){ FPSCR |= (CAUSE_Z | FLAG_Z); }

invalid(float *dest)
{
    set_V();
    if((FPSCR & ENABLE_V) == 0) qnan(dest);
}

dz(float *dest, int sign)
{
    set_Z();
    if((FPSCR & ENABLE_Z) == 0) inf (dest,sign);
}

zero(float *dest, int sign)
{
    if(sign == 0)          *dest = 0x00000000;
    else                  *dest = 0x80000000;
}

int(float *dest, int sign)
{
    if(sign == 0)          *dest = 0x7f800000;
    else                  *dest = 0xff800000;
}

qnan(float *dest)
{
    *dest = 0x7fbfffff;
}
```

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

Format	Abstract	Code	Cycle	T Bit
FABS FRn	FRn → FRn	1111nnnn01011101	1	—

Description: Obtains arithmetic absolute value (as a floating point number) of the contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```
FABS(float *Frn) /* FABS FRn */
{
    clear_cause_VZ();
    case(data_type_of(FRn))
        NORM:      if(sign_of(FRn) == 0) *FRn = *FRn;
                   else *FRn = -*FRn;
                   break;
        PZERO :
        NZERO :    zero(FRn, 0);
                   break;
        PINF  :
        NINF  :    inf(FRn, 0);
                   break;
        qnan  :    qnan(FRn);
                   break;
        sNaN  :    invalid(FRn);
                   break;
    }
    pc += 2;
}
```

FABS Special Cases

FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
FABS(FRn)	ABS	+0	+0	+INF	+INF	qNaN	Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

FABS FR2 ; Floating point absolute value
 ; Before execution FR2=H' C0800000/*-4 in base 10*/
 ; After execution FR2=H' 40800000/*4 in base 10*/

7.3.2 FADD (Floating Point Add): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FADD FRm,FRn	FRn + FRm → FRn	1111nnnnmmmm0000	1	—

Description: Arithmetically adds (as floating point numbers) the contents of floating point registers FRm and FRn. The calculation result is stored in FRn.

Operation:

```

FADD (float *FRm,FRn)                                /* FADD FRm,FRn */
{
    clear_cause_VZ();
    if((data_type_of(FRm) == sNaN)                    ||
        (data_type_of(FRn) == sNaN))                  invalid(FRn);
    else if((data_type_of(FRm) == qNaN)               ||
        (data_type_of(FRn) == qNaN))                  qnan(FRn);
    else case(data_type_of(FRm))                      {
NORM:
        case(data_type_of(FRn))                      {
            PINF      :      inf(FRn,0);                break;
            NINF      :      inf(FRn,1);                break;
            default   :      *FRn = *FRn + *FRm;        break;
        }
        PZERO:
            case(data_type_of(FRn))                   {
                NORM      :      *FRn = *FRn + *FRm;    break;
                PZERO     :      ;                       break;
                NZERO     :      zero(FRn,0);           break;
                PINF      :      inf(FRn,0);            break;
                NINF      :      inf(FRn,1);            break;
            }
        NZERO:
            case(data_type_of(FRn)) {
                NORM      :      *FRn = *FRn + *FRm;    break;
                PZERO     :      zero(FRn,0);           break;
                NZERO     :      zero(FRn,1);           break;
                PINF      :      inf(FRn,0);            break;
            }
    }
}

```

```

        NINF      :      inf(FRn,1);                break;
    }
    PINF:
        case(data_type_of(FRn))                    {
            NINF      :      invalid(FRn);          break;
            default   :      inf(FRn,0);           break;
        }
    NINF:
        case(data_type_of(FRn)){
            PINF      :      invalid(FRn);          break;
            default   :      inf(FRn,1);           break;
        }
    }
    pc += 2;
}

```

FADD Special Cases

FRm	FRn						
	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	ADD				-INF	qNaN	Invalid
+0	+0						
-0	-0						
+INF				+INF	Invalid		
-INF	-INF			Invalid	-INF		
qNaN	qNaN						
sNaN							Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FADD    FR2,FR3           ; Floating point add
                          ; Before execution: FR2=H' 40400000/*3 in base 10*/
                          ;                   FR3=H' 3F800000/*1 in base 10*/
                          ; After execution:  FR2=H' 40400000
                          ;                   FR3=H' 40800000/*4 in base 10*/

FADD    FR5,FR4           ;
                          ; Before execution: FR5=H' 40400000/*3 in base 10*/
                          ;                   FR4=H' C0000000/*-2 in base 10*/
                          ; After execution:  FR5=H' 40400000
                          ;                   FR4=H' 3F800000/*1 in base 10*/
```

7.3.3 FCMP (Floating Point Compare): Floating Point Instruction

Format	Abstract	Code	Cycle	T Bit
FCMP/ EQ FRm,FRn	(FRn == FRm)? 1:0 → T	1111nnnnnnmmmm0100	1	Comparison result
FCMP/GT FRm,FRn	(FRn > FRm)? 1:0 → T	1111nnnnnnmmmm0101	1	Comparison result

Description: Arithmetically compares (as floating point numbers) the contents of floating point registers FRm and FRn. The calculation result (true/false) is written to the T bit.

Operation:

```

FCMP_EQ(float *FRm,FRn)    /* FCMP/EQ FRm,FRn */
{
    clear_cause_VZ();
    if (fcmp_chk(FRm,FRn) == INVALID) {fcmp_invalid(0); }
    else if(fcmp_chk(FRm,FRn) == EQ)      T = 1;
    else                                  T = 0;
    pc += 2;
}

FCMP_GT(float *FRm,FRn)    /* FCMP/GT FRm,FRn */
{
    clear_cause_VZ();
    if (fcmp_chk(FRm,FRn)==INVALID) || {fcmp_chk(FRm,FRn)==UO} {
        fcmp_invalid(0);}
    else if(fcmp_chk(FRm,FRn) == GT)      T = 1;
    else                                  T = 0;
    pc += 2;
}

fcmp_chk(float *FRm,*FRn)
{
    if((data_type_of(FRm) == sNaN) ||
        (data_type_of(FRn) == sNaN))    return(INVALID);
    else if((data_type_of(FRm) == qNaN) || ||
        (data_type_of(FRn) == qNaN))    return(UO);
}

```

```

else      case(data_type_of(FRm))      {
          NORM      : case(data_type_of(FRn))      {
                    PINF      : return(GT);      break;
                    NINF      : return(NOTGT);    break;
                    default    :                  break;
          }
          PZERO      :
          NZERO      : case(data_type_of(FRn))      {
                    PZERO      :
                    NZERO      : return(EQ);      break;
                    PINF      : return(GT);      break;
                    NINF      : return(NOTGT);    break;
                    default    :                  break;
          }
          PINF      : case(data_type_of(FRn))      {
                    PINF      : return(EQ)      break;
                    default    : return(NOTGT);  break;
          }
          NINF      : case(data_type_of(FRn))      {
                    NINF      : return(EQ);      break;
                    default    : return(GT);      break;
          }
        }
if(*FRn == *FRm)      return(EQ);
else if(*FRn > *FRm) return(GT);
else                  return(NOTGT);
}
fcmp_invalid(int cmp_flag)
{
  set_V();
  if((FPSCR & ENABLE_V) == 0) T = cmp_flag;
}

```

FCMP Special Cases

FRm	FRn						
	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	CMP	EQ		GT	!GT	UO	Invalid
+0							
-0							
+INF	!GT		EQ	EQ			
-INF	GT						
qNaN							
sNaN							

- Notes: 1. UO if result is FCMP/EQ, invalid if result is FCMP/GT.
 2. Non-normalized values are treated as zero.

Exceptions: Invalid operation

Note: Four comparison operations that are independent of each other are defined in the IEEE standard, but the SH-2E supports FCMP/EQ and FCMP/GT only. However, all comparison conditions can be supported by using these two FCMP instructions in combination with the BT and BF instructions.

(FRm = FRn)	<code>fcmp/eq FRm, FRn ; bt</code>
(FRm != FRn)	<code>fcmp/eq FRm, FRn ; bf</code>
(FRm < FRn)	<code>fcmp/gt FRm, FRn ; bt</code>
(FRm <= FRn)	<code>fcmp/gt FRn, FRm ; bt</code>
(FRm > FRn)	<code>fcmp/gt FRn, FRm ; bt</code>
(FRm >= FRn)	<code>fcmp/gt FRm, FRn ; bf</code>
Unorder FRm, FRn	<code>fcmp/eq FRm, FRm ; bf</code>

Examples:

```

FCMP/EQ:
    FLDI1        FR6            ;FR6=H' 3F800000/*1 in base 10*/
    FLDI1        FR7            ;FR7=H' 3F800000
    CLRT                    ;T Bit =0
    FCMP/EQ      FR6,FR7        ; Floating point compare, equal
    BF           TRGET_F        ; Don't branch (T=1)
    NOP
    BT/S        TRGET_T        ; Branch
    FADD        FR6,FR7        ; Delay slot, FR7=H' 40000000/*2 in base 10*/
    NOP
TRGET_F FCMP/EQ      FR6,FR7
    BT/S  TRGET_T        ; Don't branch (T=0)
    FLDI1        FR7            ; Delay slot
TRGET_T FCMP/EQ      FR6,FR7    ; T bit = 0
    BF  TRGET_F        ; Branch first time only
    NOP                    ;FR6=FR7=H' 3F800000/*1 in base 10*/
    .END

FCMP/GT:
    FLDI1        FR2            ;FR2=H' 3F800000/*1 in base 10*/
    FLDI1        FR7            ;FR7=H' 40000000/*2 in base 10*/
    FADD        FR2,FR7        ;FR7=H' 40000000/*2 in base 10*/
    CLRT                    ; T bit = 0
    FCMP/GT      FR2,FR7        ; Floating point compare, greater than
    BT/S        TRGET_T        ; Branch (T=1)
    FLDI1        FR7            ;
TRGET_T FCMP/GT      FR2,FR7    ; T bit = 0
    BT          TRGET_T        ; Don't branch (T=0)
    .END

```


7.3.4 FDIV (Floating Point Divide): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FDIV FRm, FRn	FRn/FRm → FRn	1111nnnnmmmm0011	13	—

Description: Arithmetically divides (as floating point numbers) the contents of floating point register FRn by the contents of floating point register FRm. The calculation result is stored in FRn.

Operation:

```

FDIV(float *FRm, *FRn) /* FDIV FRm, FRn */
{
    clear_cause_VZ();
    if((data_type_of(FRm) == sNaN) | |
        (data_type_of(FRn) == sNaN))    invalid(FRn);
    else if((data_type_of(FRm) == qNaN) | |
        (data_type_of(FRn) == qNaN))    qnan(FRn);
    else case((data_type_of(FRm)
NORM :
case(data_type_of(FRn))
    PINF      :
    NINF      :  inf(FRn, sign_of(FRm)^sign_of(FRn));  break;
    default   :  *FRn = *FRn / *FRm;                    break;
    }
PZERO :
NZERO :
case(data_type_of(FRn))
    PZERO      :
    NZERO      :  invalid(FRn);                          break;
    PINF      :
    NINF      :  inf(FN, Sign_of(FRm)^sign_of(FRn));    break;
    default   :  dz(FRn, sign_of(FRm)^sign_of(FRn));    break;
    }
PINF :
NINF :
case(data_type_of(FRn))

```

```

        PINF      :
        NINF      :  invalid(FRn);                break;
        default   : zero (FRn, sign_of(FRm)^sign_of(FRn)); break
                                                    break;
    }
    pc += 2;
}

```

FDIV Special Cases

FRm	FRn						
	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	DIV	0		INF		qNaN	Invalid
+0	DZ	Invalid					
-0							
+INF	0	+0	-0	Invalid			
-INF		-0	+0				
qNaN	qNaN						
sNaN	Invalid						

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation, divide by zero

Examples:

```

FDIV    FR6, FR5    ; Floating point divide
                ; Before execution:  ;FR5=H'40800000/*4 in base 10*/
                ;                    ;FR6=H'40400000/*3 in base 10*/
                ; After execution:   ;FR5=H'3FAAAAAA/*1.33... in base 10*/
                ;                    ;FR6=H'40400000

```

7.3.5 FLDI0 (Floating Point Load Immediate 0): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FLDI0 FRn	H'00000000 → FRn	1111nnnn10001101	1	—

Description: Loads the floating point number 0 (0x00000000) in floating point register FRn.

Operation:

```

FLDI0(float *FRn)                                /* FLDI0 FRn */
{
    *FRn = 0x00000000;
    pc += 2;
}

```

Exceptions: None

Examples:

```

FLDI0    FR1          ; Load immediate 0
           ; Before execution: FR1=x (don't care)
           ; After execution:  FR1=00000000

```

7.3.6 FLDI1 (Floating Point Load Immediate 1): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FLDI1 FRn	H'3F800000 → FRn	1111nnnn100111101	1	—

Description: Loads the floating point number 1 (0x3F800000) in floating point register FRn.

Operation:

```

FLDI1(float *FRn)                                /* FLDI1 FRn */
{
    *FRn = 0x3F800000;
    pc += 2;
}

```

Exceptions: None

Examples:

```

FLDI1    FR2                ; Load immediate 1
                          ; Before execution: FR2=x (don't care)
                          ; After execution:  FR2=H' 3F800000/*1 in base 10*/

```

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

Format	Abstract	Code	Cycles	T Bit
FLDS FRm,FPUL	FRm → FPUL	1111nnnn00011101	1	—

Description: Loads the contents of floating point register FRm to system register FPUL.

Operation:

```

FLDS(float *FRm, *FPUL)          /* FLDS FRm,FPUL */
{
    *FPUL = *FRm;
    PC += 2;
}

```

Exceptions: None

Examples:

```

; Before execution of FLDS and FSTS:
FLDI1    FR6          ; FR6=H'3F800000/*1 in base 10*/
FLDI0    FR2          ; FR2=0

; After execution of FLDS and FSTS:
FLDS     FR6, FPUL    ; FPUL=H'3F800000
FSTS     FPUL, FR2    ; FR2= H'3F800000

```

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

Format	Abstract	Code	Cycles	T Bit
FPUL,FRn	(float) FPUL → FRn	1111nnnn00101101	1	—

Description: Interprets the contents of FPUL as an integer value and converts it into a floating point number. The result is stored in floating point register FRn.

Operation:

```

FLOAT(int, *FPUL, float *FRn)          /* FLOAT FRn */
{
    clear_cause_VZ();
    *FRn = (float)*FPUL;
    pc += 2;
}

```

Exceptions: None

Examples:

```

; Floating Point Convert from Integer
; Before execution of FLOAT instruction:
MOV.L   #H'00000003, R1      ; R1=H'00000003
FLDIO   FR2                  ; FR2=0
; After execution of FLOAT instruction:
LDS     R1, FPUL             ; FPUL=H'00000003
FLOAT   FPUL, FR2           ; FR2=H'40400000/*3 in base 10*/

```

7.3.9 FMAC (Floating Point Multiply Accumulate): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FMAC FR0, FRm, FRn	$FR0 \times FRm + FRn \rightarrow FRn$	1111nmmmmmmmm1110	1	—

Description: Arithmetically multiplies (as floating point numbers) the contents of floating point registers FR0 and FRm. To this calculation result is added the contents of floating point register FRn, and the result is stored in FRn.

Operation:

```

FMAC(float *FR0, *FRm, *FRn)      /* FMAC FR0, FRm, FRn */
{
long      tmp_FPSCR;
float     *tmp_FMUL = *FRm;
          FMUL(F0, tmp_FMUL);
          pc -= 2;                /* correct pc */
          tmp_FPSCR = FPSCR;     /* save cause field for FR0*FRm */
          FADD(tmp_FMUL, FRn);
          FPSCR |= tmp_FPSCR;    /* reflect cause field for F0*FRm */
}

```

FMAC Special Cases

FRn	FR0	FRm								
		+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN	
NORM	NORM	MAC				INF				
	0					Invalid				
	+INF	+INF	-INF	Invalid		+INF	-INF			
	-INF	-INF	+INF			-INF	+INF			
+0	NORM	MAC				INF				
	0					Invalid				
	+INF	+INF	-INF	Invalid		+INF	-INF			
	-INF	-INF	+INF			-INF	+INF			
-0	+NORM	MAC			+0	-0	+INF	-INF		
	-NORM				-0	+0	-INF	+INF		
	+0	+0	-0	+0	-0	Invalid				
	-0	-0	+0	-0	+0					
	+INF	+INF	-INF	Invalid		+INF	-INF			
	-INF	-INF	+INF			-INF	+INF			
+INF	+NORM	+INF				Invalid				
	-NORM					Invalid				
	0					Invalid				
	+INF	Invalid				+INF				
	-INF	Invalid	+INF			+INF				
-INF	+NORM	-INF				Invalid				
	-NORM					Invalid				
	0					Invalid				
	+INF	Invalid			Invalid		-INF			
	-INF	-INF			-INF		Invalid			
qNaN	0					Invalid				
	INF					Invalid				
	!sNaN									
!NaN	qNaN					qNaN				
All types	sNaN									
sNaN	All types									

Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```

FMAC FR0, FR3, FR5    ;Floating point multiply accumulate
                        FR0*FR3+FR5->FR5
                        ;Before execution:  FR0=H' 40000000/*2 in base 10*/
                        ;                   FR3=H' 40800000/*4 in base 10*/
                        ;                   FR5=H' 3F800000/*1 in base 10*/
                        ;After execution:    FR0=H' 40000000/*2 in base 10*/
                        ;                   FR3=H' 40800000/*4 in base 10*/
                        ;                   FR5=H' 41100000/*9 in base 10*/

```

```

FMAC FR0, FR0, FR5    ;FR0*FR0+FR5->FR5
                        ;Before execution:  FR0=H' 40000000/*2 in base 10*/
                        ;                   FR5=H' 3F800000/*1 in base 10*/
                        ;After execution:    FR0=H' 40000000/*2 in base 10*/
                        ;                   FR5=H' 40A00000/*5 in base 10*/

```

```

FMAC FR0, FR5, FR0    ;FR0*FR5+FR0->FR5
                        ;Before execution:  FR0=H' 40000000/*2 in base 10*/
                        ;                   FR5=H' 40A00000/*5 in base 10*/
                        ;After execution:    FR0=H' 41400000/*12 in base 10*/
                        ;                   FR5=H' 40A00000/*5 in base 10*/

```

7.3.10 FMOV (Floating Point Move): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
1. FMOV FRm,FRn	FRm → FRn	1111nnnnnnmmmm1100	1	—
2. FMOV.S @Rm,FRn	(Rm) → FRn	1111nnnnnnmmmm1000	1	—
3. FMOV.S FRm,@Rn	FRm → (Rn)	1111nnnnnnmmmm1010	1	—
4. FMOV.S @Rm+,FRn	(Rm) → FRn, Rm+ = 4	1111nnnnnnmmmm1001	1	—
5. FMOV.S FRm,@-Rn	Rn- = 4, FRm → (Rn)	1111nnnnnnmmmm1011	1	—
6. FMOV.S @(R0,Rm),FRn	(R0 + Rm) → FRn	1111nnnnnnmmmm0110	1	—
7. FMOV.S FRm,@(R0,Rn)	FRm → (R0 + Rn)	1111nnnnnnmmmm0111	1	—

Description:

1. Moves the contents of floating point register FRm to floating point register FRn.
2. Loads the contents of the memory addresses specified by general-use register Rm to floating point register FRn.
3. Stores the contents of floating point register FRm in the memory address position specified by general-use register Rm.
4. Loads the contents of the memory addresses specified by general-use register Rm to floating point register FRn. After the load completes successfully, increments the value of Rm by 4.
5. Stores the contents of floating point register FRm in the memory address position specified by general-use register Rn-4. After the store completes successfully, the decremented value (Rn-4) becomes the value of Rm.
6. Loads the contents of the memory addresses specified by general-use registers Rm and R0 to floating point register FRn.
7. Stores the contents of floating point register FRm in the memory address position specified by general-use registers Rn and R0.

Operation:

```

FMOV(float *FRm,*FRn)          /* FMOV.S FRm,FRn */
{
    *FRn = *FRm;
    pc += 2;
}
FMOV_LOAD(long *Rm,float *FRn)      /* FMOV @Rm,FRn */
{
    if(load_long(Rm,FRn) !=Address_Error)
        load_long(Rm,FRn);
    pc += 2;
}
FMOV_STORE(float *FRm,long *Rn)      /* FMOV.S FRm,@Rn */
{
    if(store_long(FRm,tmp_address) !=Address_Error)
        store_long(FRm,Rn);
    pc += 2;
}
FMOV_RESTORE(long *Rm,float *FRn)    /* FMOV.S @Rm+,FRn */
{
    if(load_long(Rm,FRn) !=Address_Error)
        *Rm += 4;
    pc += 2;
}
FMOV_SAVE(float *FRm,long *Rn)       /*FMOV.S FRm,@-Rn */
{
long    *tmp_address =*Rn -4;
        if(store_long(FRm,tmp_address) !=Address_Error)
            Rn = tmp_address;
        pc += 2;
}
FMOV_LOAD_index(long *Rm, long *R0, float *FRn)/* FMOV.S @(R0,Rm),FRn*/
{
    if (load_long(&(*Rm+*R0),FRn), != Address_Error);
    pc += 2;
}

```

```

FMOV_STORE_index(float *FRm,long *R0, long *Rn)/* FMOV.S FRm,@(R0,Rn)*/
{
    if (store_long(FRm,&(*Rn+*R0)), != Address_Error);
    pc += 2;
}

```

Exceptions: Address error

Examples:

```

FMOV.S   @R1, FR2       ;Load
                                     ;Before execution:  @R1=H' 00ABCDEF
                                     ;                    FR2=0
                                     ;After execution:    @R1=H' 00ABCDEF
                                     ;                    FR2=H' 00ABCDEF

```

```

FMOV.S   FR2, @R3       ;Store
                                     ;Before execution:  @R3=0
                                     ;                    FR2=H' 40800000
                                     ;After execution:    @R3=H' 40800000
                                     ;                    FR2=H' 40800000

```

```

FMOV.S   @R3+, FR3      ;Restore
                                     ;Before execution:  R3=H' 0C700028
                                     ;                    @R3=H' 40800000
                                     ;                    FR3=0
                                     ;After execution:    R3=H' 0C70002C
                                     ;                    FR3=H' 40800000

```

```

FMOV.S   FR4, @-R3      ;Save
                                     ;Before execution:  R3=H' 0C700044
                                     ;                    @R3=0
                                     ;                    FR4=H' 01234567

```

```

; After execution:    R3=H'0C700040
;
;                    @R3=H'01234567
;                    FR4=H'01234567

FMOV.S    @(R0, R3), FR4 ; Load with index
; Before execution:  R0=H'00000004
;
;                    R3=H'0C700040
;                    @H'0C700044=H'00ABCDEF
;
;                    FR=4
; After execution:  R0=H'00000004
;
;                    R3=H'0C700040
;
;                    FR4=H'00ABCDEF

FMOV.S    FR5, @(R0,R3) ; Store with index
; Before execution:  R0=H'00000028
;
;                    R3=H'0C700040
;                    @H'0C700068=0
;                    FR5=H'76543210
; After execution:  R0=H'00000028
;
;                    R3=H'0C700040
;                    @H'0C700068=H'76543210
;

FMOV.S    FR5, FR6      ; Register file contents
; Before execution:  FR5=H'76543210
;
;                    FR6=x(don't care)
; After execution:  FR5=H'76543210
;
;                    FR6=H'76543210

```

7.3.11 FMUL (Floating Point Multiply): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FMUL FRm,FRn	$FRn \times FRm \rightarrow FRn$	1111nnnnnnmmmm0010	1	—

Description: Arithmetically multiplies (as floating point numbers) the contents of floating point registers FRm and FRn. The calculation result is stored in FRn.

Operation:

```

FMUL(float *FRm,*FRn) /* FMUL FRm,FRn */
{
    clear_cause_VZ();
    if((data_type_of(FRm) == sNaN) ||
        (data_type_of(FRn) == sNaN))    invalid(FRn);
    else if((data_type_of(FRm) == qNaN) ||
        (data_type_of(FRn) == qNaN))    qnan(FRn);
    else case(data_type_of(FRm) {
        NORM      :
        case(data_type_of(FRn)) {
            PINF   :
            NINF   : inf(FRn,sign_of(FRm)^sign_of(FRn)); break;
            default: *FRn=(*FRm)*(*FRm);                break;
        }
        PZERO     :
        NZERO     :
        case(data_type_of(FRn)) {
            PINF   :
            NINF   : invalid(FRn);                        break;
            default: zero(FRn,sign_of(FRm)^sign_of(FRn)); break;
        }
        PINF     :
        NINF     :
        case(data_type_of(FRn)) {
            PZERO  :
            NZERO  : invalid(FRn);                        break;
        }
    }
}

```

```

                                default:inf (FRn,sign_of(FRm)^sign_of(FRn)); break
                                }
                                break;
    }
    pc += 2;
}

```

FMUL Special Cases

FRm	FRn						
	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	MUL	0		INF		qNaN	Invalid
+0	0	+0	-0	Invalid			
-0		-0	+0				
+INF	INF	Invalid		+INF	-INF		
-INF				-INF	+INF		
qNaN	qNaN						
sNaN	Invalid						

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```

FMUL    FR2, FR3    ; Floating point multiply
                    ; Before execution: FR2=H'40000000/*2 in base 10*/
                    ;                   FR3=H'40800000/*4 in base 10*/
                    ; After execution:  FR2=H'40000000
                    ;                   FR3=H'41000000/*8 in base 10*/

```

7.3.12 FNEG (Floating Point Negate): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FNEG FRn	-FRn → FRn	1111nnnn01001101	1	—

Description: Arithmetically negates (as a floating point number) the contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```
FNEG(float *Frn)          /* FNEG FRn */
{
    clear_cause_VZ();
    case(data_type_of(FRn))
    {
        qNaN      :    qnan(FRn);      break;
        sNaN      :    invalid(FRn);   break;
        default   :    *FRn = -(*Frn); break;
    }
    pc += 2;
}
```

FNEG Special Cases

FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
FNEG(FRn)	NEG	-0	+0	-INF	+INF	qNaN	Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FNEG    FR2          ; Floating point negate
          ; Before execution: FR2=H' 40800000 /*4 in base 10*/
          ; After execution:  FR2=H' C0800000 /*-4 in base 10*/
```


7.3.13 FSTS (Floating Point Store From System Register): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FSTS FPUL,FRn	FPUL → FRn	1111nnnn00001101	1	—

Description: Copies the contents of system register FPUL to floating point register FRn.

Operation:

```
FSTS(float *FRn,*FPUL)      /* FSTS FPUL,FRn */
{
    *FRn = *FPUL;
    pc += 2;
}
```

Exceptions: None

Examples:

```
MOV.L   #H'00000002, R2      ;Before execution of FSTS instruction: ;R2=H'00000002
FLDI0   FR5                  ;FR5=0
LDS     R2,FPUL              ;After execution of FSTS instruction: ;R2=H'00000002
FSTS    FPUL, R5             ;FR5= H'00000002
```

7.3.14 FSUB (Floating Point Subtract): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FSUB FRm, FRn	FRn – FRm → FRn	1111nnnnmmmm0001	1	—

Description: Arithmetically subtracts (as floating point numbers) the contents of floating point register FRm from contents of floating point register FRn. The calculation result is stored in FRn.

Operation:

```

FSUB(float *FRm,FRn)                                /* FSUB FRm,FRn */
{
    clear_cause_VZ();
    if((data_type_of(FRm) == sNaN) | |
        (data_type_of(FRn) == sNaN))                invalid(FRn);
    else if((data_type_of(FRm) == qNaN) | |
            (data_type_of(FRn) == qNaN))            qnan(FRn);
    else case(data_type_of(FRm))                    {
        NORM      :
        case(data_tyoe_of(FRn)) {
            PINF   :          inf(FRn,0);          break;
            NINF   :          inf(FRn,1);          break;
            default :          *FRn = *FRn - *FRm;  break;
        }
        PZERO     :
        case(data_type_of(FRn)) {
            NORM   :          *FRn = *FRn- *FRm;   break;
            PZERO  :          zero(FRn,0);         break;
            NZERO  :          zero(FRn,1);         break;
            PINF   :          inf(FRn,0);          break;
            NINF   :          inf(FRn,1);          break;
        }
        NZERO     :
        case(data_type_of(FRn)) {
            NORM   :          *FRn = *FRn - *FRm;  break;
            PZERO  :
    
```

```

NZERO      :          zero(FRn,0);          break;
PINF       :          inf(FRn,0);          break;
NINF       :          inf(FRn,1);          break;
}                                                  break;
PINF       :
case(data_type_of(FRn)) {
  NINF      :          invalid(FRn);        break;
  default   :          inf(FRn,1);         break;
}                                                  break;
NINF       :
case(data_type_of(FRn)) {
  PINF      :          invalid(FRn);        break;
  default   :          inf(FRn,0);         break;
}                                                  break;
}
pc += 2;
}

```

FSUB Special Cases

FRm	FRn						
	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	SUB			+INF	-INF	qNaN	Invalid
+0							
-0	+0						
+INF	-INF			Invalid			
-INF	+INF			Invalid			
qNaN	qNaN						
sNaN							

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```
FSUB    FR0, FR3    ;Floating point subtract
          ;Before execution:  ;FR0=H'3F800000/*1 in base 10*/
          ;                    ;FR3=H'40E00000/*7 in base 10*/
          ;After execution:   ;FR0=H'3F800000/*1 in base 10*/
          ;                    ;FR3=H'40C00000/*6 in base 10*/

FSUB    FR3, FR2    ;
          ;Before execution:  ;FR2=H'40800000/*4 in base 10*/
          ;                    ;FR3=H'40C00000/*6 in base 10*/
          ;After execution:   ;FR2=H'C0000000/*-2 in base 10*/
          ;                    ;FR3=H'40C00000/*6 in base 10*/
```

7.3.15 FTRC (Floating Point Truncate And Convert To Integer): Floating Point Instruction

Format	Abstract	Code	Cycles	T Bit
FTRC FRm, FPUL	(long) FRm → FPUL	1111nnnn00111101	1	—

Description: Interprets the contents of floating point register FRm as a floating point number and converts it to an integer by truncating everything after the decimal point. The calculation result is stored in FRn.

Operation:

```
#define N_INT_RANGE 0xCF000000          /* 01.000000 * 2^16 */
#define P_INT_RANGE 0x47FFFFFF          /* 1.fffffe * 2^30 */

FTRC(float *FRm,int *FPUL)             /* FTRC FRm,FPUL */
{
    clear_cause_VZ();
    case(ftrc_type_of(FRm)) {
        NORM      :      *FPUL = (long)(*FRm);break;
        PINF      :      ftrc_invalid(0);      break;
        NINF      :      ftrc_invalid(1);      break;
    }
    pc += 2;
}

int ftrc_type_of(long *src)
{
    long abs;
        abs = *src & 0x7FFFFFFF;
    if(sign_of(src) == 0) {
        if(abs > 0x7F800000) return(NINF); /* NaN*/
        else if(abs > P_INT_RANGE) return(PINF); /* out of range,+INF */
        else return(NORM); /* +0,+NORM */
    }
    else {
        if(*src > N_INT_RANGE) return(NINF); /* out of range ,+INF,NaN*/
    }
}
```

```

        else                return(NORM); /* -0,-NORM*/
    }
}
ftrc_invalid(long *dest,int sign)
{
    set_V();
    if((FPSCR & ENABLE_V) == 0) {
        if(sign == 0)      *dest = 0x7FFFFFFF;
        else                *dest = 0x80000000;
    }
}

```

FTRC Special Cases

FRn	NORM	+0	-0	positive out of range	negative out of range	+INF	-INF	qNaN	sNaN
FTRC (FRn)	TRC	0	0	7FFFFFFF F	80000000 0	Invalid +MAX Invalid	-MAX Invalid	-MAX Invalid	-MAX Invalid

Note: Non-normalized values are treated as zero.

Exceptions: Invalid operation

Examples:

```

MOV.L   #H'402ED9EB, R2
LDS     R2, FPUL
FSTS    FPUL, FR6          ;FR6=H'402ED9EB/*2.7320 in base 10*/
FTRC    FR6, FPUL
STS     FPUL, R2          ;R2=H'00000002/*2 in base 10*/
                          ; Before execution of FTRC and STS:
                          ;     R2=H'402ED9EB
                          ;     FR6=H'402ED9EB
                          ; After execution of FTRC and STS:
                          ;     R2=H'00000002
                          ;     FR6=H'402ED9EB

```

7.3.16 LDS (Load to System Register): FPU Related CPU Instruction

Format	Abstract	Code	Cycles	T Bit
1. LDS Rm, FPUL	Rm → FPUL	0100nnnn01011010	1	—
2. LDS.L @Rm+,FPUL	(Rm) → FPUL, Rm+ = 4	0100nnnn01010110	1	—
3. LDS Rm,FPSCR	Rm → FPSCR	0100nnnn01101010	1	—
4. LDS.L @Rm+,FPSCR	(Rm) → FPSCR, Rm+ = 4	0100nnnn01100110	1	—

Description:

1. Moves the contents of general-use register Rm to system register FPUL.
2. Loads the contents of the memory addresses specified by general-use register Rm to system register FPUL. After the load completes successfully, increments the value of Rm by 4.
3. Moves the contents of general-use register Rm to system register FPSCR. Previously defined bits in FPSCR are not changed.
4. Loads the contents of the memory addresses specified by general-use register Rm to system register FPSCR. After the load completes successfully, increments the value of Rm by 4. Previously defined bits in FPSCR are not changed.

Operation:

```
#define FPSCR_MASK 0x00018C60
LDS(long *Rm, *FPUL) /* LDS Rm,FPUL */
{
    *FPUL = *Rm;
    pc += 2;
}
LDS_RESTORE(long *Rm, *FPUL) /* LDS.L @Rm+,FPUL */
{
    if(load_long(Rm,FPUL) != Address_Error) *Rm += 4 ;
    pc += 2;
}

LDS(long *Rm, *FPSCR) /* LDS Rm,FPSCR */
{
    *FPSCR = *Rm & FPSCR_MASK;
    pc += 2;
}
```

```

}
LDS_RESTORE(long *Rm, *FPSCR)          /* LDS.L @Rm+,FPSCR */
{
    long *tmp_FPSCR;
    if(load_long(Rm, tmp_FPSCR) != Address_Error){
        *FPSCR =*tmp_FPSCR & FPSCR_MASK;
        *Rm += 4 ;
    }
    pc += 2;
}

```

Exceptions: Address error

Examples:

- LDS

Example 1

```

MOV.L   #H'12345678, R2           ; Before execution of LDS and FSTS instructions:
                                           ;
                                           ;           R2=H'12345678
FLDIO   FR3                       ;           FR3=0
LDS     R2, FPUL                   ; After execution of LDS and FSTS instructions:
                                           ;
                                           ;           R2=H'12345678
FSTS    FPUL, FR3                 ;           FR3= H'12345678

```

Example 2

```

MOV.L   #H'00040801, R4           ; After execution of LDS instruction:
LDS     R4, FPSCR                 ; FPSCR=00040801

```


- LDS.L

Example 1

```

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

```

Example 2

```

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

```

7.3.17 STS (Store from FPU System Register): FPU Related CPU Instruction

Format	Abstract	Code	Cycles	T Bit
1. STS FPUL,Rn	FPUL → Rn	0000nnnn01011010	1	—
2. STS.L FPUL,@-Rn	Rn- = 4, FPUL → @(Rn)	0100nnnn01010010	1	—
3. STS FPSCR,Rn	FPSCR → Rn	0000nnnn01101010	1	—
4. STS.L FPSCR,@-Rn	Rn- = 4, FPSCR → @(Rn)	0100nnnn01100010	1	—

Description:

1. Moves the contents of system register FPUL to general-use register Rn.
2. Stores contents of system register FPUL at the memory address position specified by general-use register Rn-4. After the store completes successfully, the decremented value becomes the value of Rn.
3. Moves the contents of system register FPSCR to general-use register Rn.
4. Stores contents of system register FPSCR at the memory address position specified by general-use register Rn-4. After the store completes successfully, the decremented value becomes the value of Rn.

Operation:

```

STS(long *FPUL, *Rn)                                /* STS.L FPUL,Rn */
{
    *Rn = *FPUL;
    pc += 2;
}
STS_SAVE(long *FPUL, *Rn)                            /* STS.L FPUL,@-Rn */
{
    long *tmp_address = *Rn - 4;
    if(store_long(FPUL,tmp_address) != Address_Error)
        Rn = tmp_address;
    pc += 2;
}
STS(long *FPSCR, *Rn)                                /* STS FPSCR,Rn */
{
    *Rn = *FPSCR;

```

```

        pc += 2;
    }

```

STS SToRe from FPU System register

```

STS_RESTORE long *FPSCR, *Rn)          /* STS.L FPSCR, @-Rn */
{
long *tmp_address = *Rn - 4;
    if(store_long(FPSCR tmp_address) != Address_Error)
        Rn = tmp_address
    pc += 2;
}

```

Exceptions: Address error

Examples:

- STS

Example 1

```

MOV.L    #H'12ABCDEF, R12
LDS.L    @R12, FPUL
STS      FPUL, R13

```

; After execution of STS instruction:

```
;          R13 = 12ABCDEF
```

Example 2

```
STS      FPSCR, R2
```

; After execution of STS instruction:

```
;          Contents of FPSCR at that point stored in R2 register
```

• STS.L

Example 1

```
MOV.L    #H'0C700148, R7
```

```
STS      FPUL, @-R7
```

```
        ; Before execution of STS.L instruction:
```

```
        ;           R7 = H'0C700148
```

```
        ; After execution of STS.L instruction:
```

```
        ;           R7 = H'0C700144, contents of FPUL saved at  
        ;           address H'0C700144
```

```
        ;           location H'0C700144
```

Example 2

```
MOV.L    #H'0C700154, R8
```

```
STS.L    FPSCR, @-R8
```

```
        ; After execution of STS.L instruction:
```

```
        ;           Contents of FPSCR saved at address H'0C700150
```

Section 8 Pipeline Operation

This section describes the operation of the pipelines for each instruction. This information is provided to allow calculation of the required number of CPU instruction execution states (system clock cycles).

8.1 Basic Configuration of Pipelines

The Five-Stage Pipeline: Pipelines are composed of the following five stages:

- IF (Instruction fetch)
Fetches instruction from the memory where the program is stored.
- ID (Instruction decode)
Decodes the instruction fetched.
- EX (Instruction execution)
Does data operations and address calculations according to the results of decoding.
- MA (Memory access)
Accesses data in memory. Generated by instructions that involve memory access, with some exceptions.
- WB (Write back)
Returns the results of the memory access (data) to a register. Generated by instructions that involve memory loads, with some exceptions.

These stages flow with the execution of the instructions and thereby constitute a pipeline. At a given instant, five instructions are being executed simultaneously. The basic pipeline flow is as shown in figure 8.1. The period in which a single stage is operating is called a slot and is indicated by two-way arrows (\longleftrightarrow).

All instructions have at least the 3 stages IF, ID and EX, but not all have stages MA and WB. The way the pipeline flows also varies with the type of instruction, with some having two MA stages, some accessing the FPU (mm), and so on. Finally, conflicts can occur, for example between IF and MA. When such a conflict occurs, the pipeline flow changes.

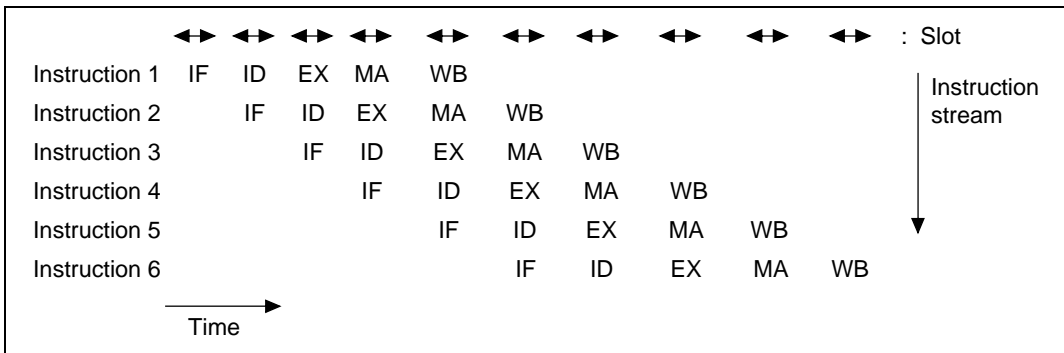


Figure 8.1 Basic Structure of Pipeline Flow

FPU Pipeline: The durations of the stages in the FPU pipeline are the same as those of the stages in the CPU pipeline. In both pipelines, the first stage is instruction fetch (IF). The FPU pipeline also has the following four additional stages:

- DF (Decode FPU)
Decodes the fetched instruction.
- E1 (FPU execution stage 1)
Initializes the floating-point operation.
- E2 (FPU execution stage 2)
Completes the floating-point operation.
- SF (Store FPU)
Stores the result in the FPU register.

All instructions pass through both the CPU and the FPU pipelines. Depending on the instruction, operations are performed either by the CPU pipeline alone or by both pipelines.

In the case of floating-point instructions and FPU-related CPU instructions, the FPU pipeline and CPU pipeline operate simultaneously in parallel.

In the case of instructions involving the CPU only, the FPU pipeline does not operate; only the CPU pipeline operates.

Refer to 8.8 Instruction Pipeline Operation for details.

Slot Length: The number of states (system clock cycles) S for the execution of one slot is calculated with the following conditions:

- $S =$ (the cycles of the stage with the highest number of cycles of all instruction stages contained in the slot). This means that the instruction with the longest stage stalls others with shorter stages.
- The number of execution cycles for each stage:
 - IF The number of memory access cycles for instruction fetch
 - ID Always one cycle
 - EX Always one cycle
 - MA The number of memory access cycles for data access
 - WB Always one cycle

As an example, figure 8.4 shows the flow of a pipeline in which the IF (memory access for instruction fetch) of instructions 1 and 2 are two cycles, the MA (memory access for data access) of instruction 1 is three cycles and all others are one cycle. The dashes indicate the instruction is being stalled.

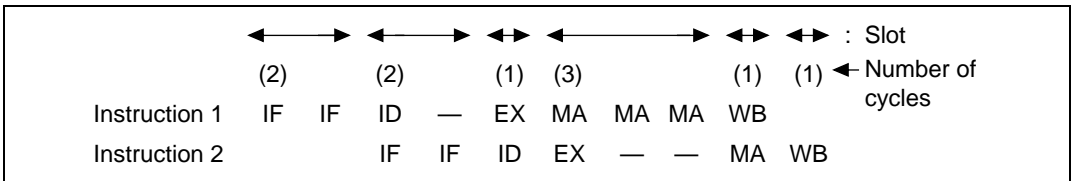


Figure 8.4 Slots Requiring Multiple Cycles

8.3 Number of Instruction Execution Cycles

The number of instruction execution cycles is counted as the interval between execution of EX stages. The number of cycles between the start of the EX stage for instruction 1 and the start of the EX stage for the following instruction (instruction 2) is the execution time for instruction 1.

For example, in a pipeline flow like that shown in figure 8.5, the EX stage interval between instructions 1 and 2 is five cycles, so the execution time for instruction 1 is five cycles. Since the interval between EX stages for instructions 2 and 3 is one cycle, the execution time of instruction 2 is one cycle.

If a program ends with instruction 3, the execution time for instruction 3 should be calculated as the interval between the EX stage of instruction 3 and the EX stage of a hypothetical instruction 4, using a MOV Rm, Rn that follows instruction 3. (In figure 8.5, the execution time of instruction 3 would thus be one cycle.) In this example, the MA of instruction 1 and the IF of instruction 4 are in contention. For operation during the contention between the MA and IF, see section 8.4, Contention between Instruction Fetch (IF) and Memory Access (MA).

The total execution time for instructions 1 through 3 in Figure 8 is seven cycles (5 + 1 + 1).

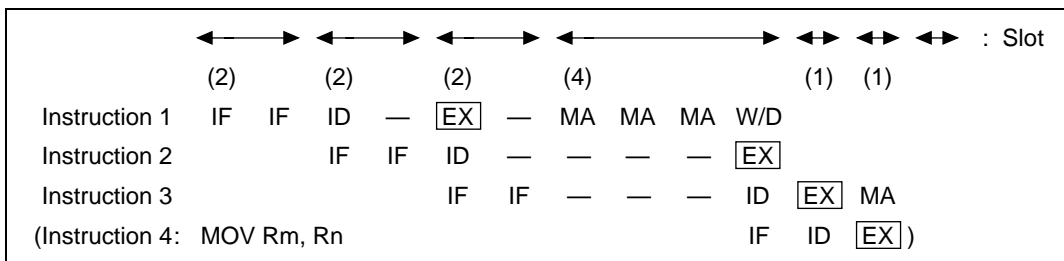


Figure 8.5 Method for Counting Instruction Execution Cycles

8.4 Contention between Instruction Fetch (IF) and Memory Access (MA)

Basic Operation when IF and MA Are in Contention: The IF and MA stages both access memory, so they cannot operate simultaneously. When the IF and MA stages both try to access memory within the same slot, the slot splits as shown in figure 8.6. When there is a WB, it is executed immediately after the MA ends.

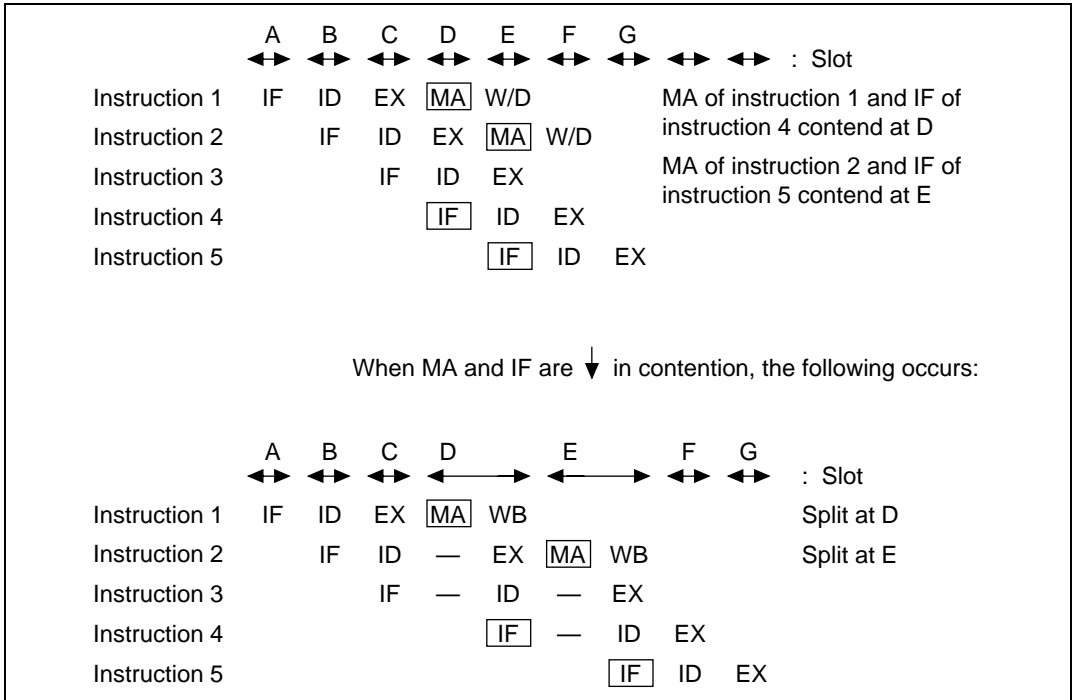


Figure 8.6 Operation when IF and MA Are in Contention

The slots in which MA and IF contend are split into two cycles. MA is given priority to execute in the first half (when there is a WB, it immediately follows the MA), and the EX, ID, and IF are executed simultaneously in the latter half. For example, in figure 8.6 the MA of instruction 1 is executed in slot D while the EX of instruction 2, the ID of instruction 3 and IF of instruction 4 are executed simultaneously thereafter. In slot E, the MA of instruction 2 is given priority and the EX of instruction 3, the ID of instruction 4 and the IF of instruction 5 executed thereafter.

The number of cycles for a slot in which MA and IF are in contention is the sum of the number of memory access cycles for the MA and the number of memory access cycles for the IF.

Relationship between Locations of Instructions in Memory and IF Stages: The SH-2E accesses instructions in memory in the 32-bit mode. Since all of the SH-2E instructions have a fixed length of 16 bits, it is basically possible to access two instructions per IF stage. Whether the IF fetches one instruction or two depends on where in memory the instruction(s) are located (word/longword boundary).

If an instruction is located at a longword boundary, it is possible to fetch two instructions using a single IF operation. This means that the IF for the next instruction does not generate a separate bus cycle in order to fetch the instruction. In addition, the IF for the instruction after that fetches two instructions, and therefore the IF for the instruction which follows again generates no bus cycle.

In other words, IF stages for instructions located in memory at longword boundaries (instructions for which the bottom two address bits are 00: $A1 = 0, A0 = 0$) actually fetch two instructions. Therefore no bus cycle is generated by the IF for the following instruction. These instruction fetches that do not generate bus cycles are indicated in lower case as "if" rather than IF. An "if" is always one cycle.

On the other hand, if due to branching or the like an instruction at a word boundary (instructions for which the bottom two address bits are 10: $A1 = 1, A0 = 0$) is fetched, only one instruction can be fetched in the IF bus cycle. Consequently, the IF for the next instruction generates a bus cycle. Then two instructions are fetched from the subsequent IF onward. Figure 8.7 illustrates the operations described above.

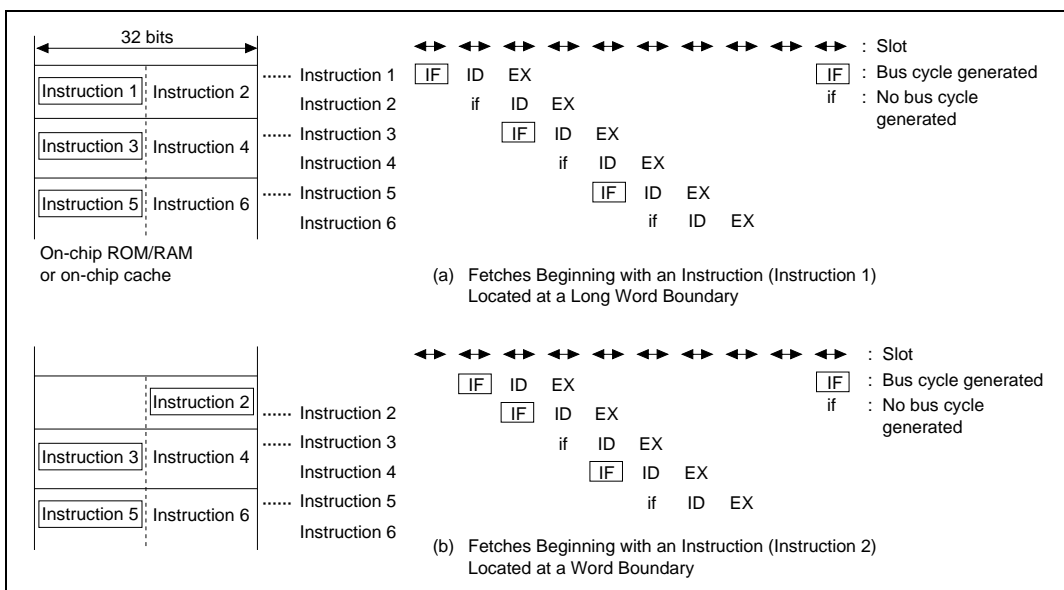


Figure 8.7 Relationship between Locations of Instructions in Memory and IF Stages

Relationship between Position of Instructions Located in On-Chip Memory and Contention between IF and MA: When an instruction is located in on-chip memory, there are instruction fetch stages (“if”, written in lower case) that do not generate bus cycles. When an if is in contention with an MA, the slot will not split, as it does when an IF and an MA are in contention, because ifs and MAs can be executed simultaneously. Such slots execute in the number of cycles the MA requires for memory access. This is illustrated in Figure 8.8.

When programming, avoid contention of MA and IF whenever possible and pair MAs with ifs to increase the instruction execution speed. In other words, if an instruction with a four (five) stage pipeline consisting of IF, ID, EX, MA, (MB) is located at a memory longword boundary (the instruction's bottom two address bits are 00: A1 = 0, A0 = 0), the MA stage uses the same slot as the if following it, so no stall occurs.

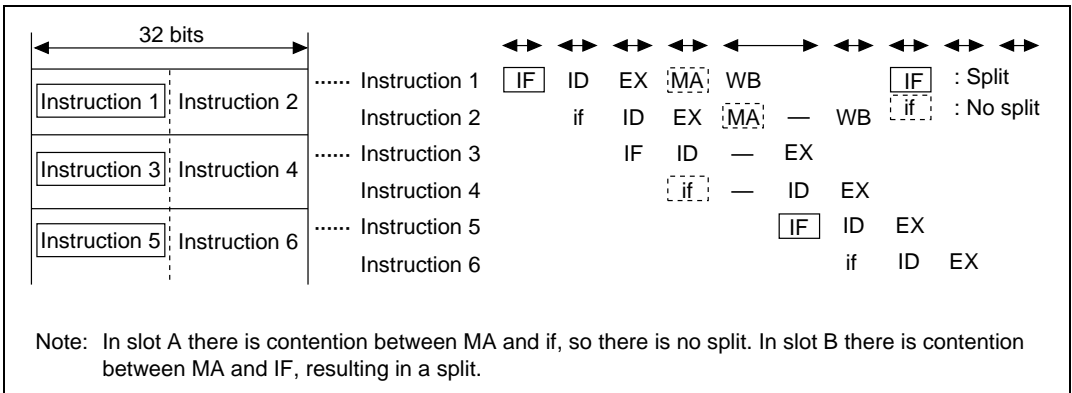


Figure 8.8 Relationship between Position of Instructions Located in On-chip Memory and Contention between IF and MA

8.5 Effects of Memory Load Instructions on the Pipeline

Instructions that involve loading from memory return data to the destination register during the WB stage, which comes at the end of the pipeline. The WB stage of such a load instruction (load instruction 1) will thus not have ended before after the EX stage of the instruction that immediately follows it (instruction 2) begins.

When instruction 2 uses the same destination register as load instruction 1, the contents of that register will not be ready, so any slot containing the MA of instruction 1 and EX of instruction 2 will split. When the destination register of load instruction 1 is the same as the destination, not the source, of instruction 2 it will still split.

When the destination of load instruction 1 is the status register (SR) and the flag in it is fetched by instruction 2 (as ADDC does), a split occurs. No split occurs, however, in the following cases:

- When instruction 2 is a load instruction and its destination is the same as that of load instruction 1
- When instruction 2 is MAC @Rm+,@Rn+ and the destinations of Rm and load instruction 1 were the same

The number of cycles in the slot generated by the split is the number of MA cycles plus the number of IF (or if) cycles, as shown in figure 8.9. This means the execution speed will be lowered if the instruction that will use the results of the load instruction is placed immediately after the load instruction. The instruction that uses the result of the load instruction will not slow down the program if placed one or more instructions after the load instruction.

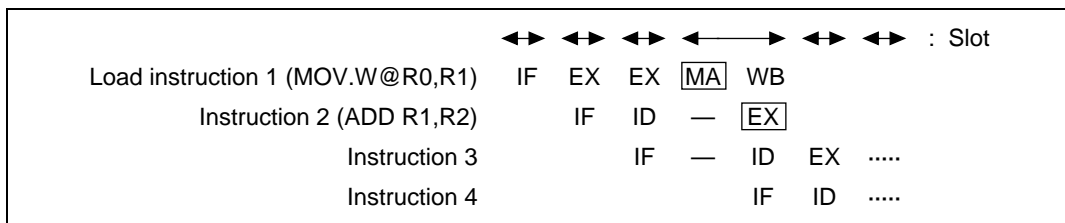


Figure 8.9 Effects of Memory Load Instructions on the Pipeline (1)

8.6 FPU Contention

In addition to the LDS and STS instructions, which move data between the CPU and FPU, loading and storing floating point numbers also uses the MA stage of the pipeline. Consequently, such instructions create contention with the IF stage.

If the register (FR0 to FR15, FPUL) to which the result of a floating point arithmetic calculation instruction, the FMOV instruction, or a floating point number load instruction is stored is read (used as the source register) by the next instruction, the execution of this instruction (the next instruction) is delayed by one slot cycle (Figure 8.10).

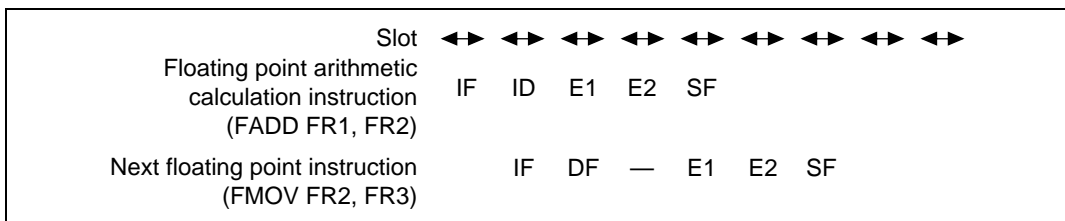


Figure 8.10 FPU Contention 1

If the LDS or LDS.L instruction is used to change the value of FPSCR, the execution of the next instruction is delayed by two slot cycle (Figure 8.11).

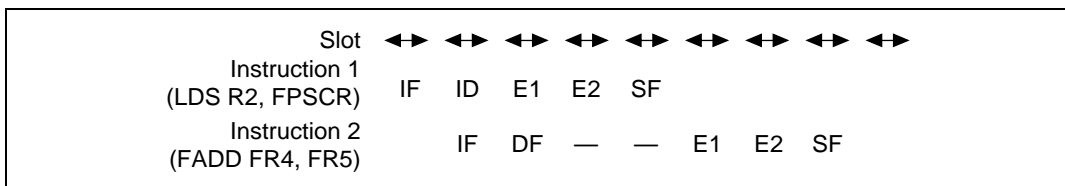


Figure 8.11 FPU Contention 2

If the STS or STS.L instruction is used to read the value of FPSCR the execution is delayed by two slot cycle (Figure 8.12).

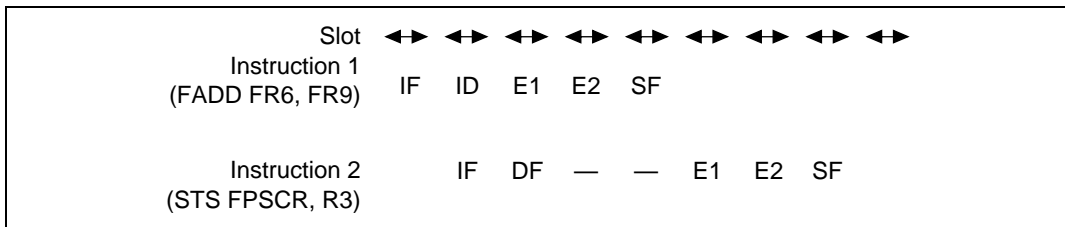


Figure 8.12 FPU Contention 3

The FDIV instruction require 13 cycles in the E1 stage. During this period, no other floating point instruction or FPU-related CPU instruction may enter the E1 stage. If another floating point instruction or FPU-related CPU instruction are encountered before the FDIV instruction has finished using the E1 stage, the fixed slot duration for the execution of that instruction is delayed, and the instruction enters the E1 stage only after the FDIV instruction has finished using the SF stage (Figure 8.13).

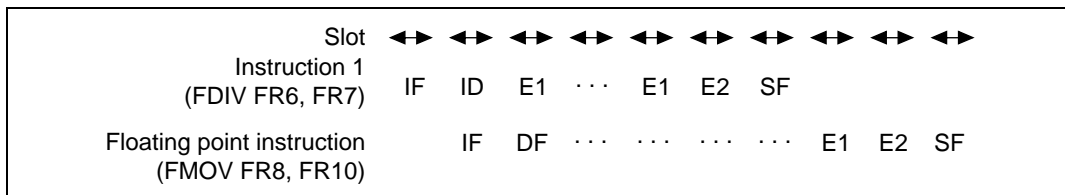


Figure 8.13 FPU Contention 4

8.7 Programming Guide

When writing programs, follow the guidelines below in order to increase instruction execution speed.

- Instructions with memory accesses (MA) should be located in memory at longword boundaries (position where the instruction's bottom two address bits are 00: A1 = 0, A0 = 0). This will prevent contention between MA and instruction fetch (IF).
- The instruction immediately following a memory load instruction should not use the same register as the destination register of the load instruction.
- Instructions that use the FPU should be arranged so that they are not sequential. Also, instructions that access registers MACH and MACL in order to fetch the results of operations performed by the FPU should no be situated immediately following instructions that use the FPU.
- The instruction immediately preceding a floating-point arithmetic operation instruction should not use the destination register of the floating-point operation instruction.
- As far as possible, avoid placing a floating-point instruction or FPU-related CPU instruction within the 14 instructions following the FDIV instruction.

8.8 Operation of Instruction Pipelines

This section describes the operation of the instruction pipelines. By combining these with the rules described so far, the way pipelines flow in a program and the number of instruction execution cycles can be calculated.

In the following figures, “Instruction A” refers to the instruction being discussed. When “IF” is written in the instruction fetch stage, it may refer to either “IF” or “if”. When there is contention between IF and MA, the slot will split, but the manner of the split is not discussed in the tables, with a few exceptions. When a slot has split, see section 8.4, Contention between Instruction Fetch (IF) and Memory Access (MA). Base your response on the rules for pipeline operation given there.

Table 8.1 shows the number of instruction stages and number of execution cycles as follows:

- Type: Given by function
- Category: Categorized by differences in instruction operation
- Stages: The number of stages in the instruction
- Cycles: The number of execution cycles when there is no contention
- Contention: Indicates the contention that occurs
- Instructions: Gives a mnemonic for the instruction concerned

Table 8.1 Number of Instruction Stages and Execution Cycles

Type	Category	Stages	Cycles	Contention	Instruction
Data transfer instructions	Register-register transfer instructions	3	1	—	MOV #imm, Rn
					MOV Rm, Rn
					MOVA @(disp, PC), R0
					MOVT Rn
					SWAP.B Rm, Rn
					SWAP.W Rm, Rn
					XTRCT Rm, Rn
Memory load instructions	5	1	<ul style="list-style-type: none"> • Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF 	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	
				MOV.W @Rm+, Rn	
				MOV.L @Rm+, Rn	
				MOV.B @(disp, Rm), R0	
				MOV.W @(disp, Rm), R0	
				MOV.L @(disp, Rm), Rn	
				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					

Type	Category	Stages	Cycles	Contention	Instruction
Data transfer instructions (cont)	Memory store instructions	4	1	MA contends with IF	MOV.B @Rm, Rn
					MOV.W @Rm, Rn
					MOV.L @Rm, Rn
					MOV.B Rm, @-Rn
					MOV.W Rm, @-Rn
					MOV.L Rm, @-Rn
					MOV.B R0, @(disp, Rn)
					MOV.W R0, @(disp, Rn)
					MOV.L Rm, @(disp, Rn)
					MOV.B Rm, @(R0, Rn)
					MOV.W Rm, @(R0, Rn)
					MOV.L Rm, @(R0, Rn)
					MOV.B R0, @(disp, GBR)
					MOV.W R0, @(disp, GBR)
					MOV.L R0, @(disp, GBR)
Arithmetic instructions	Arithmetic instructions between registers (except multiplication instructions)	3	1	—	ADD Rm, Rn
					ADD #imm, Rn
					ADDC Rm, Rn
					ADDV Rm, Rn
					CMP/EQ #imm, R0
					CMP/EQ Rm, Rn
					CMP/HS Rm, Rn
					CMP/GE Rm, Rn
					CMP/HI Rm, Rn
					CMP/GT Rm, Rn
					CMP/PZ Rn
					CMP/PL Rn
					CMP/STR Rm, Rn
					DIV1 Rm, Rn
					DIV0S Rm, Rn
DIV0U					

Type	Category	Stages	Cycles	Contention	Instruction
Arithmetic instructions (cont)					DT Rn
					EXTS.B Rm, Rn
					EXTS.W Rm, Rn
					EXTU.B Rm, Rn
					EXTU.W Rm, Rn
					NEG Rm, Rn
					NEGC Rm, Rn
					SUB Rm, Rn
				SUBC Rm, Rn	
				SUBV Rm, Rn	
Multiply/add instructions	7	$3/(2)^{*1}$	<ul style="list-style-type: none"> If an instruction that uses the FPU follows this instruction, FPU contention occurs. MA contends with IF 	MAC.W @Rm+, @Rn+	
Double-length multiply/accumulate instruction	9	$3/(2 \text{ to } 4)^{*1}$	<ul style="list-style-type: none"> If an instruction that uses the FPU follows this instruction, FPU contention occurs. MA contends with IF 	MAC.L @Rm+, @Rn+	
Multiplication instructions	6	$1 \text{ to } 3^*1$	<ul style="list-style-type: none"> If an instruction that uses the FPU follows this instruction, FPU contention occurs. MA contends with IF 	MULS.W Rm, Rn MULU.W Rm, Rn	
Double-length multiply/accumulate instruction	9	$2 \text{ to } 4^*1$	<ul style="list-style-type: none"> If an instruction that uses the FPU follows this instruction, FPU contention occurs. MA contends with IF 	DMULS.L Rm, Rn DMULU.L Rm, Rn MUL.L Rm, Rn	

Type	Category	Stages	Cycles	Contention	Instruction	
Logic operation instructions	Register-register logic operation instructions	3	1	—	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
	Memory logic operations instructions	6	3	MA contends with IF	AND.B	#imm, @(R0, GBR)
					OR.B	#imm, @(R0, GBR)
					TST.B	#imm, @(R0, GBR)
	TAS instruction	6	4	MA contends with IF	TAS.B	@Rn
Shift instructions	Shift instructions	3	1	—	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					

Type	Category	Stages	Cycles	Contention	Instruction
Branch instructions	Conditional branch instructions	3	3/1*2	—	BF label
					BT label
	Delayed conditional branch instructions	3	2/1*2	—	BF/S label BT/S label
Unconditional branch instructions	3	2	—	BRA label	
				BRAF Rm	
				BSR label	
				BSRF Rm	
				JMP @Rm	
				JSR @Rm	
				RTS	
System control instructions	System control ALU instructions	3	1	—	CLRT
					LDC Rm, SR
					LDC Rm, GBR
					LDC Rm, VBR
					LDS Rm, PR
					NOP
					SETT
					STC SR, Rn
					STC GBR, Rn
					STC VBR, Rn
STS PR, Rn					
LDS.L instructions (PR)	5	1	<ul style="list-style-type: none"> Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction MA contends with IF 	LDS.L @Rm+, PR	

Type	Category	Stages	Cycles	Contention	Instruction
System control instructions (cont)	STS.L instruction (PR)	4	1	MA contends with IF	STS.L PR, @-Rn
	LDC.L instructions	5	3	<ul style="list-style-type: none"> Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction MA contends with IF 	LDC.L @Rm+, SR
					LDC.L @Rm+, GBR
					LDC.L @Rm+, VBR
	STC.L instructions	4	2	MA contends with IF	STC.L SR, @-Rn STC.L GBR, @-Rn STC.L VBR, @-Rn
Register → MAC transfer instruction	4	1	<ul style="list-style-type: none"> Contention occurs with multiplier MA contends with IF 	CLRMAC LDS Rm, MACH LDS Rm, MACL	
Memory → MAC transfer instructions	4	1	<ul style="list-style-type: none"> Contention occurs with multiplier MA contends with IF 	LDS.L @Rm+, MACH LDS.L @Rm+, MACL	
MAC → register transfer instruction	5	1	<ul style="list-style-type: none"> Contention occurs with multiplier Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction MA contends with IF 	STS MACH, Rn STS MACL, Rn	

Type	Category	Stages	Cycles	Contention	Instruction	
System control instructions (cont)	MAC → memory transfer instruction	4	1	<ul style="list-style-type: none"> • Contention occurs with multiplier • MA contends with IF 	STS.L	MACH, @-Rn
	RTE instruction	5	4	—	RTE	
	TRAP instruction	9	8	—	TRAPA	#imm
	SLEEP instruction	3	3	—	SLEEP	
FPU-related CPU instruction	FPUL load instruction	5 (FPU pipeline) 4 (CPU pipeline)	1	<ul style="list-style-type: none"> • Contention occurs if next instruction reads FPUL • MA in CPU pipeline contends with IF 	LDS LDS.L	Rm, FPUL @Rm+, FPUL
	FPSCR load instruction	5 (FPU pipeline) 4 (CPU pipeline)	1	• Contention occurs as shown in Figure 8.11	LDS LDS.L	Rm, FPSCR @Rm+, FPSCR
	FPUL store instruction (STS)	4 (FPU pipeline) 5 (CPU pipeline)	1	<ul style="list-style-type: none"> • Contention occurs if next instruction uses Rn • MA in CPU pipeline contends with IF 	STS	FPUL, Rn
	FPUL store instruction (STS.L)	4 (FPU pipeline) 4 (CPU pipeline)	1	• MA in CPU pipeline contends with IF	STS.L	FPUL, @-Rn

Type	Category	Stages	Cycles	Contention	Instruction
FPU-related CPU instruction (cont)	FPSCR store instruction (STS)	4 (FPU pipeline)	1	<ul style="list-style-type: none"> Contention occurs as shown in Figure 8.12 Contention occurs if next instruction uses Rn MA in CPU pipeline contends with IF 	STS FPSCR, Rn
	FPSCR store instruction (STS.L)	4 (FPU pipeline) 4 (CPU pipeline)	1	<ul style="list-style-type: none"> Contention occurs as shown in Figure 8.12 MA in CPU pipeline contends with IF 	STS.L FPSCR, @-Rn
Floating-point instruction	Floating-point register transfer instruction	5 (FPU pipeline)	1	<ul style="list-style-type: none"> Contention occurs if next instruction reads destination register 	FLDS FRm, FPUL
		3 (CPU pipeline)			FMOV FRm, FRn
	Floating-point register immediate instruction	5 (FPU pipeline)	1	<ul style="list-style-type: none"> Contention occurs if next instruction reads destination register 	FSTS FPUL, FRn
		3 (CPU pipeline)			FLDI0 FRn
Floating-point register load instruction	Floating-point register store instruction	5 (FPU pipeline)	1	<ul style="list-style-type: none"> Contention occurs if next instruction reads destination register MA in CPU pipeline contends with IF 	FLDI1 FRn
		4 (CPU pipeline)			FMOV.S @Rm, FRn
					FMOV.S @Rm+, FRn
					FMOV.S @(R0, Rm), FRn
					FMOV.S FRm, @Rn
					FMOV.S FRm, @-Rn
					FMOV.S FRm, @(R0, Rn)

Type	Category	Stages	Cycles	Contention	Instruction	
Floating-point instruction (cont)	Floating-point register operation instruction (other than FDIV)	5 (FPU pipeline)	1	• Contention occurs if next instruction reads destination register	FABS FRn FADD FRm, FRn FLOAD FPUL, FRn FMAC FR0, FRm, FRn FMUL FRm, FRn FNEG FRn FSUB FRm, FRn FTRC FRm, FPUL	
	Floating-point register operation instruction (FDIV)	17 (FPU pipeline)	13		• Contention occurs as shown in Figure 8.13	FDIV FRm, FRn
	Floating-point register compare instruction	3 (FPU pipeline)	1			FCMP/EQ FRm, FRn FCMP/GT FRm, FRn

- Notes:
1. The normal minimum number of execution cycles. The number in parentheses is the number of cycles when there is contention with following instructions.
 2. One state when there is no branch.

8.8.1 Data Transfer Instructions

Register-Register Transfer Instructions

Instruction Types:

- MOV #imm, Rn
- MOV Rm, Rn
- MOVA @(disp, PC), R0
- MOVT Rn
- SWAP.B Rm, Rn
- SWAP.W Rm, Rn
- XTRCT Rm, Rn

Pipeline:

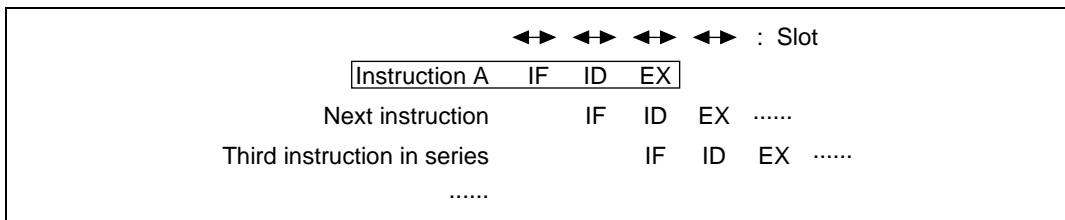


Figure 8.14 Register-Register Transfer Instruction Pipeline

Operation:

The pipeline ends after three stages: IF, ID, and EX. Data is transferred in the EX stage via the ALU.

Memory Load Instructions

Instruction Types:

- 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
- MOV.W @Rm+, Rn
- MOV.L @Rm+, Rn
- MOV.B @(disp, Rm), R0
- MOV.W @(disp, Rm), R0
- MOV.L @(disp, Rm), Rn
- 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

Pipeline:

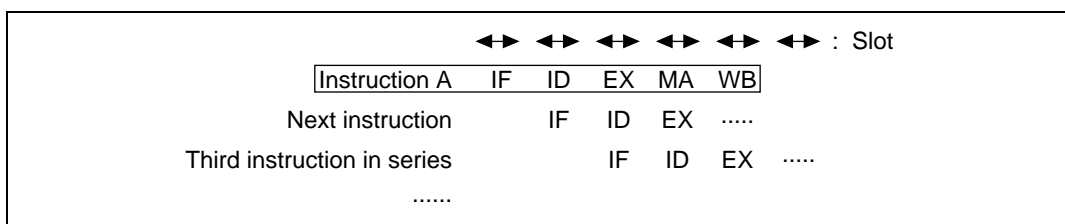


Figure 8.15 Memory Load Instruction Pipeline

Operation:

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.15). If an instruction that uses the same destination register as this instruction is placed immediately after it, contention will occur. (See section 8.5 Effects of Memory Load Instructions on the Pipeline)

Memory Store Instructions

Instruction Types:

- MOV.B Rm, @Rn
- MOV.W Rm, @Rn
- MOV.L Rm, @Rn
- MOV.B Rm, @-Rn
- MOV.W Rm, @-Rn
- MOV.L Rm, @-Rn
- MOV.B R0, @(disp, Rn)
- MOV.W R0, @(disp, Rn)
- MOV.L Rm, @(disp, Rn)
- MOV.B Rm, @(R0, Rn)
- MOV.W Rm, @(R0, Rn)
- MOV.L Rm, @(R0, Rn)
- MOV.B R0, @(disp, GBR)
- MOV.W R0, @(disp, GBR)
- MOV.L R0, @(disp, GBR)

Pipeline:

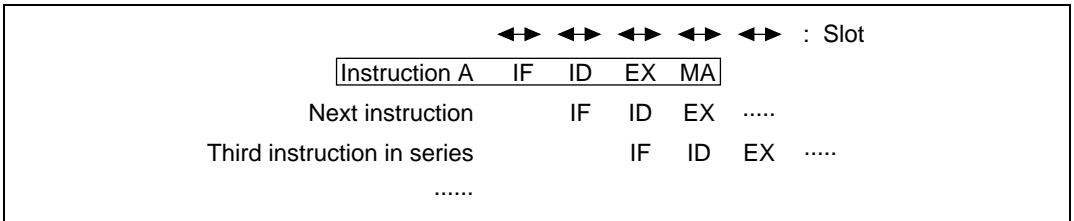


Figure 8.16 Memory Store Instructions Pipeline

Operation:

The pipeline has four stages: IF, ID, EX, and MA (figure 8.16). Data is not returned to the register so there is no WB stage.

8.8.2 Arithmetic Instructions

Arithmetic Instructions between Registers (Except Multiplication Instructions): Include the following instruction types:

- | | | | |
|-----------|----------|----------|--------|
| • ADD | Rm, Rn | • DIV1 | Rm, Rn |
| • ADD | #imm, Rn | • DIV0S | Rm, Rn |
| • ADDC | Rm, Rn | • DIV0U | |
| • ADDV | Rm, Rn | • DT | Rn |
| • CMP/EQ | #imm, R0 | • EXTS.B | Rm, Rn |
| • CMP/EQ | Rm, Rn | • EXTS.W | Rm, Rn |
| • CMP/HS | Rm, Rn | • EXTU.B | Rm, Rn |
| • CMP/GE | Rm, Rn | • EXTU.W | Rm, Rn |
| • CMP/HI | Rm, Rn | • NEG | Rm, Rn |
| • CMP/GT | Rm, Rn | • NEGC | Rm, Rn |
| • CMP/PZ | Rn | • SUB | Rm, Rn |
| • CMP/PL | Rn | • SUBC | Rm, Rn |
| • CMP/STR | Rm, Rn | • SUBV | Rm, Rn |

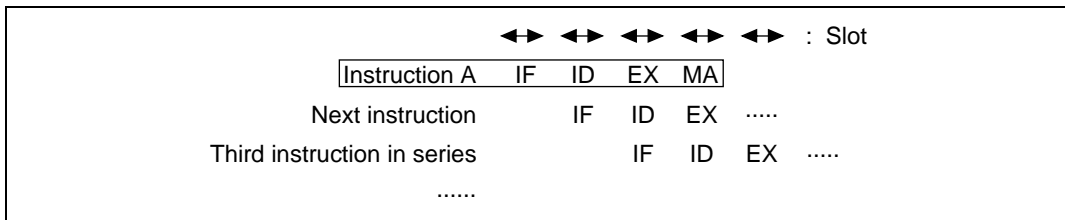


Figure 8.17 Pipeline for Arithmetic Instructions between Registers Except Multiplication Instructions

The pipeline has three stages: IF, ID, and EX (figure 8.17). The data operation is completed in the EX stage via the ALU.

Multiply/Accumulate Instruction: Includes the following instruction type:

- MAC.W @Rm+, @Rn+

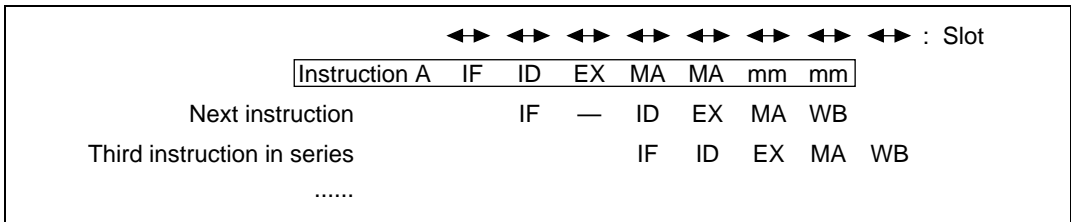


Figure 8.18 Multiply/Accumulate Instruction Pipeline

The pipeline has seven stages: IF, ID, EX, MA, MA, mm, and mm. The second MA reads the memory and accesses the multiplier. mm indicates that the multiplier is operating. mm operates for two cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC.W instruction is stalled for 1 slot. The two MAs of the MAC.W instruction, when they contend with IF, split the slots as described in Section 8.4, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier comes after the MAC.W instruction, the MAC.W instruction may be considered to be a five-stage pipeline instruction of IF, ID, EX, MA, MA. In such cases, the ID of the next instruction simply stalls one slot and thereafter operates like a normal pipeline. When an instruction that uses the multiplier comes after the MAC.W instruction, however, contention occurs with the multiplier, so operation is different from normal.

The following cases are possible:

- MAC.W instruction follows immediately after MAC.W instruction
- MAC.L instruction follows immediately after MAC.W instruction
- MULS.W instruction follows immediately after MAC.W instruction
- DMULS.L instruction follows immediately after MAC.W instruction
- STS (register) instruction follows immediately after MAC.W instruction
- STS.L (memory) instruction follows immediately after MAC.W instruction
- LDS (register) instruction follows immediately after MAC.W instruction
- LDS.L (memory) instruction follows immediately after MAC.W instruction

(a) MAC.W instruction follows immediately after MAC.W instruction

The second MA of MAC.W instruction does not contend with the mm generated by the preceding multiply instruction.

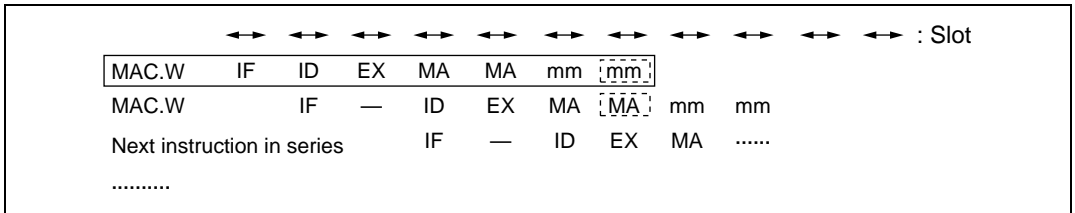


Figure 8.19 MAC.W Instruction Follows Immediately after MAC.W Instruction (1)

If the MAC.W instruction occurs twice in succession, contention between MA and IF could cause a delay in instruction execution. Refer to the diagram below. This diagram takes into account the possibility of contention between MA and IF.

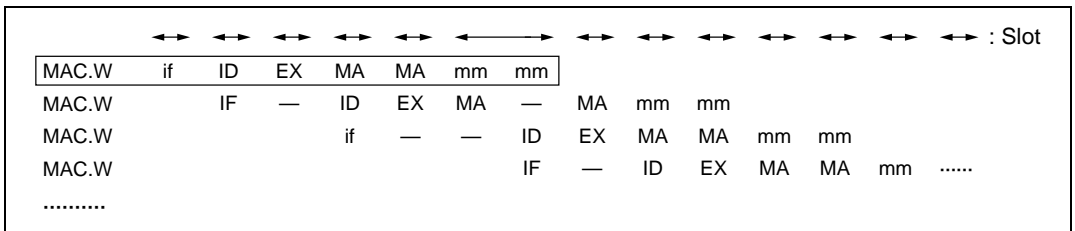


Figure 8.20 MAC.W Instruction Follows Immediately after MAC.W Instruction (2)

If contention occurs between the second MA of the MAC.W instruction and IF, the slot splits normally. Refer to the diagram below. This diagram takes into account the possibility of contention between MA and IF.

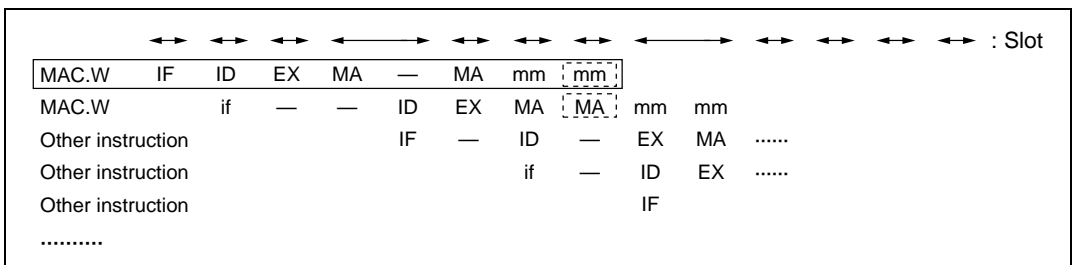


Figure 8.21 MAC.W Instruction Follows Immediately after MAC.W Instruction (3)

(f) STS.L (memory) instruction follows immediately after MAC.W instruction

If the STS instruction is used to store the contents of the MAC register in memory, the STS instruction will include an MA stage for accessing the multiplier and writing to memory, as described below. These diagrams take into account the possibility of contention between MA and IF.

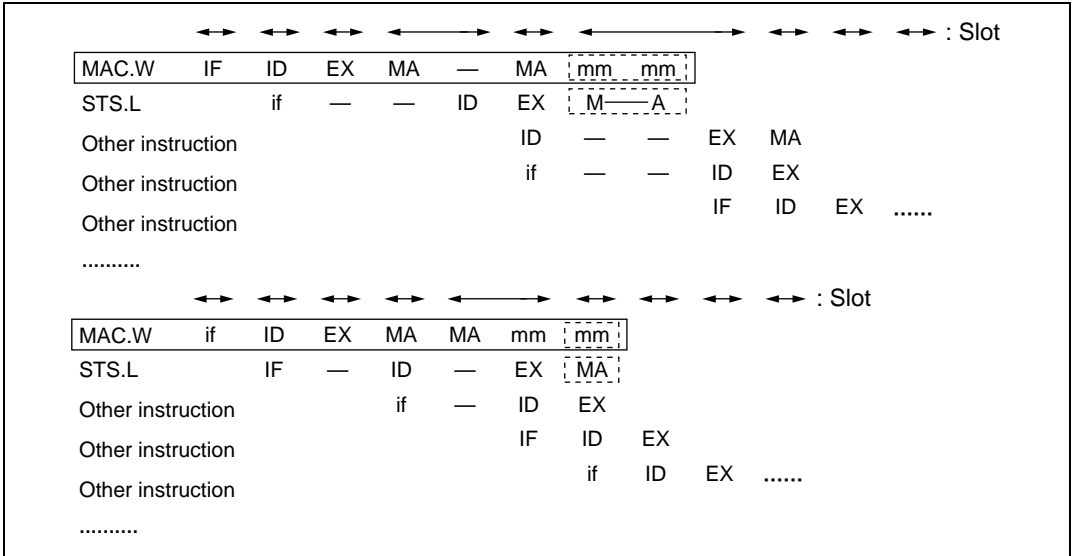


Figure 8.26 STS.L (Memory) Instruction Follows Immediately after MAC.W Instruction

(g) LDS (register) instruction follows immediately after MAC.W instruction

If the LDS instruction is used to load the contents of the MAC register from a general-use register, the LDS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

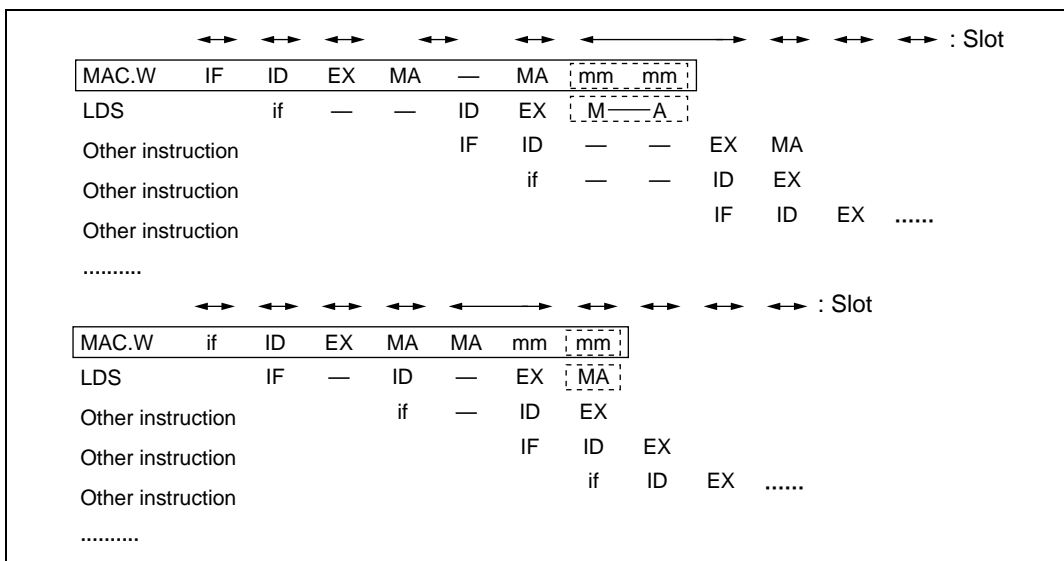


Figure 8.27 LDS (Register) Instruction Follows Immediately after MAC.W Instruction

(h) LDS.L (memory) instruction follows immediately after MAC.W instruction

If the LDS instruction is used to load the contents of the MAC register from memory, the LDS instruction will include an MA stage for accessing memory and accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

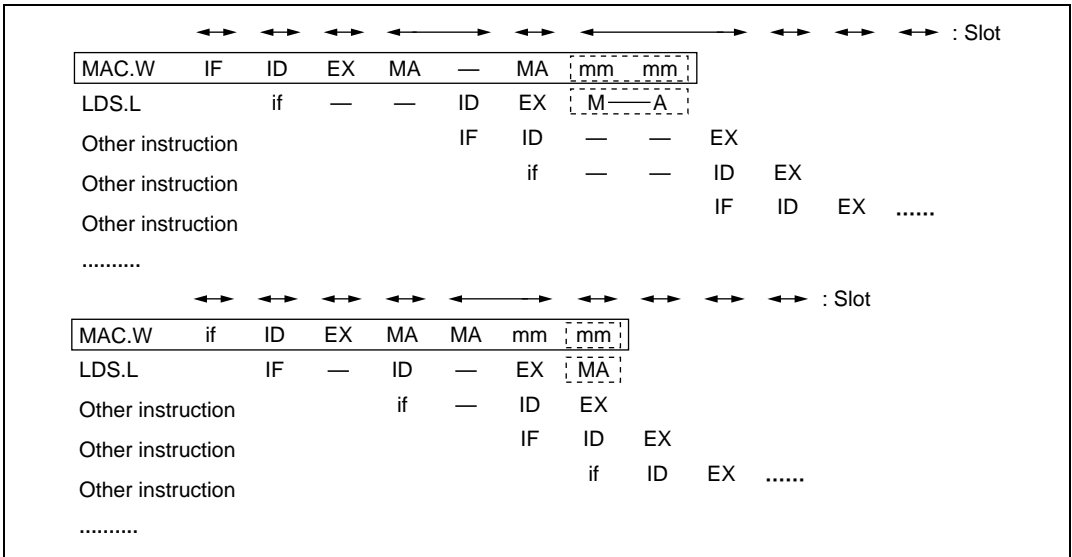


Figure 8.28 LDS.L (Memory) Instruction Follows Immediately after MAC.W Instruction

Double-Length Multiply/Accumulate Instruction: Includes the following instruction type:

- MAC.L @Rm+, @Rn+

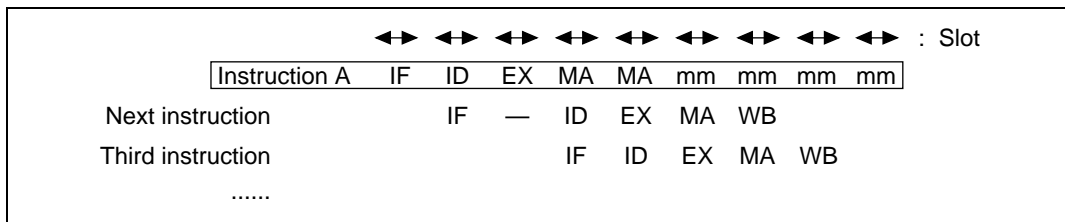


Figure 8.29 Double-Length Multiply/Accumulate Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 8.29). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for four cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC.L instruction is stalled for one slot. The two MAs of the MAC.L instruction, when they contend with IF, split the slots as described in section 8.4, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier follows the MAC.L instruction, the MAC.L instruction may be considered to be a five-stage pipeline instruction of IF, ID, EX, MA, MA. In such cases, the ID of the next instruction simply stalls one slot and thereafter the pipeline operates normally. When an instruction that uses the multiplier comes after the MAC.L instruction, contention occurs with the multiplier, so operation is different from normal.

The following cases are possible:

- MAC.L instruction follows immediately after MAC.L instruction
- MAC.W instruction follows immediately after MAC.L instruction
- DMULS.L instruction follows immediately after MAC.L instruction
- MULS.W instruction follows immediately after MAC.L instruction
- STS (register) instruction follows immediately after MAC.L instruction
- STS.L (memory) instruction follows immediately after MAC.L instruction
- LDS (register) instruction follows immediately after MAC.L instruction
- LDS.L (memory) instruction follows immediately after MAC.L instruction

(a) MAC.L instruction follows immediately after MAC.L instruction

If the second MA of the MAC.L instruction contends with the mm generated by the preceding multiply instruction, that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot.

If there are two or more instructions that do not use the multiplier located between the one MAC.L instruction and a second MAC.L instruction, no contention occurs the two MAC.L instructions and there is no delay.

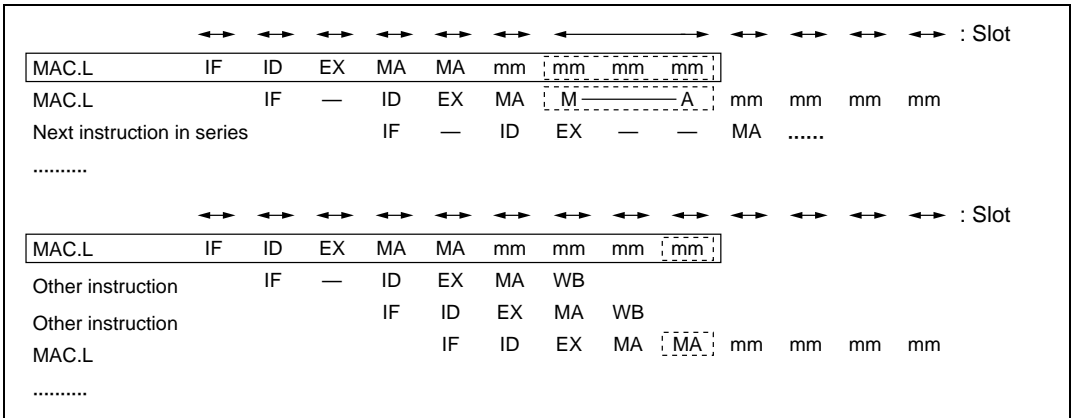


Figure 8.30 MAC.L Instruction Follows Immediately after MAC.L Instruction (1)

Even if the succession of MAC.L instructions causes delays in execution due to contention between MA and IF, multiplier contention may be reduced in some cases. Refer to the diagram below. This diagram takes into account the possibility of contention between MA and IF.

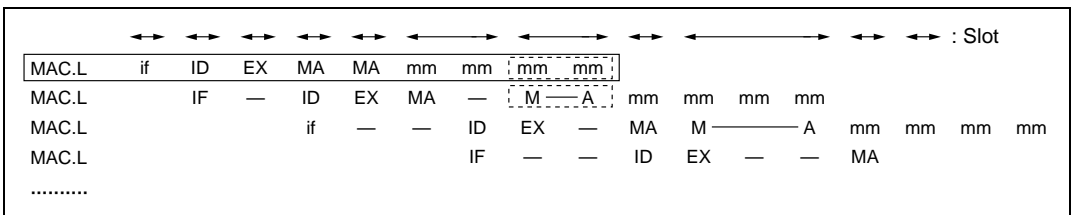


Figure 8.31 MAC.L Instruction Follows Immediately after MAC.L Instruction (2)

(c) DMULS.L instruction follows immediately after MAC.L instruction

The DMULS.L instruction has an MA stage for accessing the multiplier. If contention with the second MA of DMULS.L occurs during the MAC.L instruction's multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. If there are two or more instructions that do not use the multiplier located between the MAC.L instruction and the DMULS.L instruction, no contention occurs between MAC.L and DMULS.L, and there is no delay. Note that the slot splits if there is contention between the MA of DMULS.L and IF.

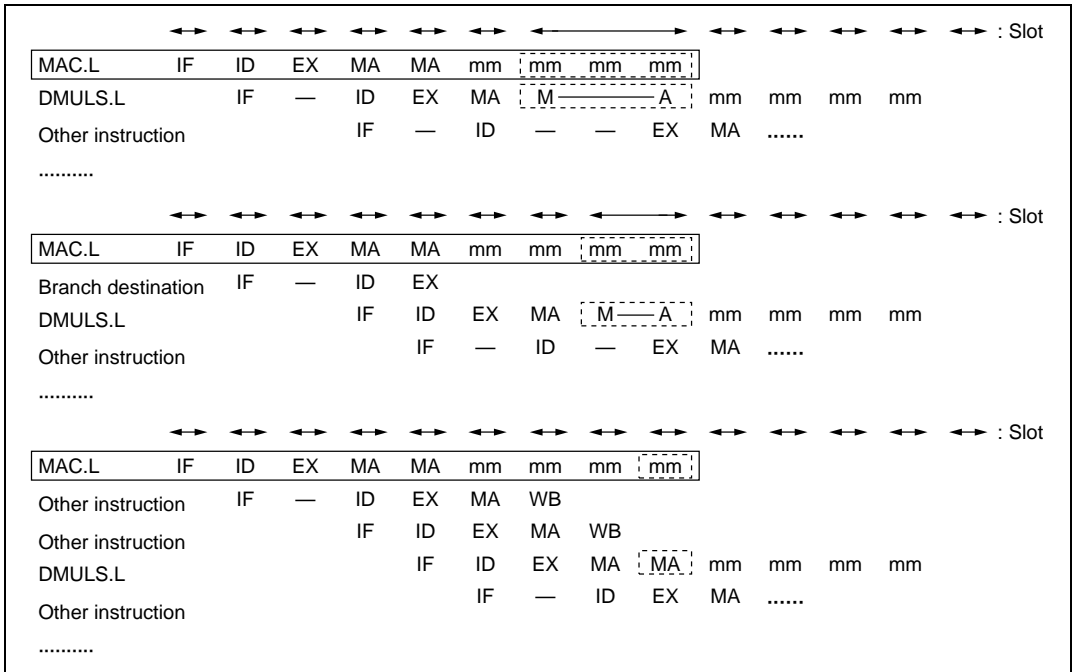


Figure 8.34 DMULS.L Instruction Follows Immediately after MAC.L Instruction

(d) MULS.W instruction follows immediately after MAC.L instruction

The MULS.W instruction has an MA stage for accessing the multiplier. If contention with the MA of MULS.W occurs during the MAC.L instruction's multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. If there are three or more instructions that do not use the multiplier located between MAC.L and MULS.W, no contention occurs between MAC.L and MULS.W and there is no delay. Note that the slot splits if there is contention between the MA of MULS.W and IF.

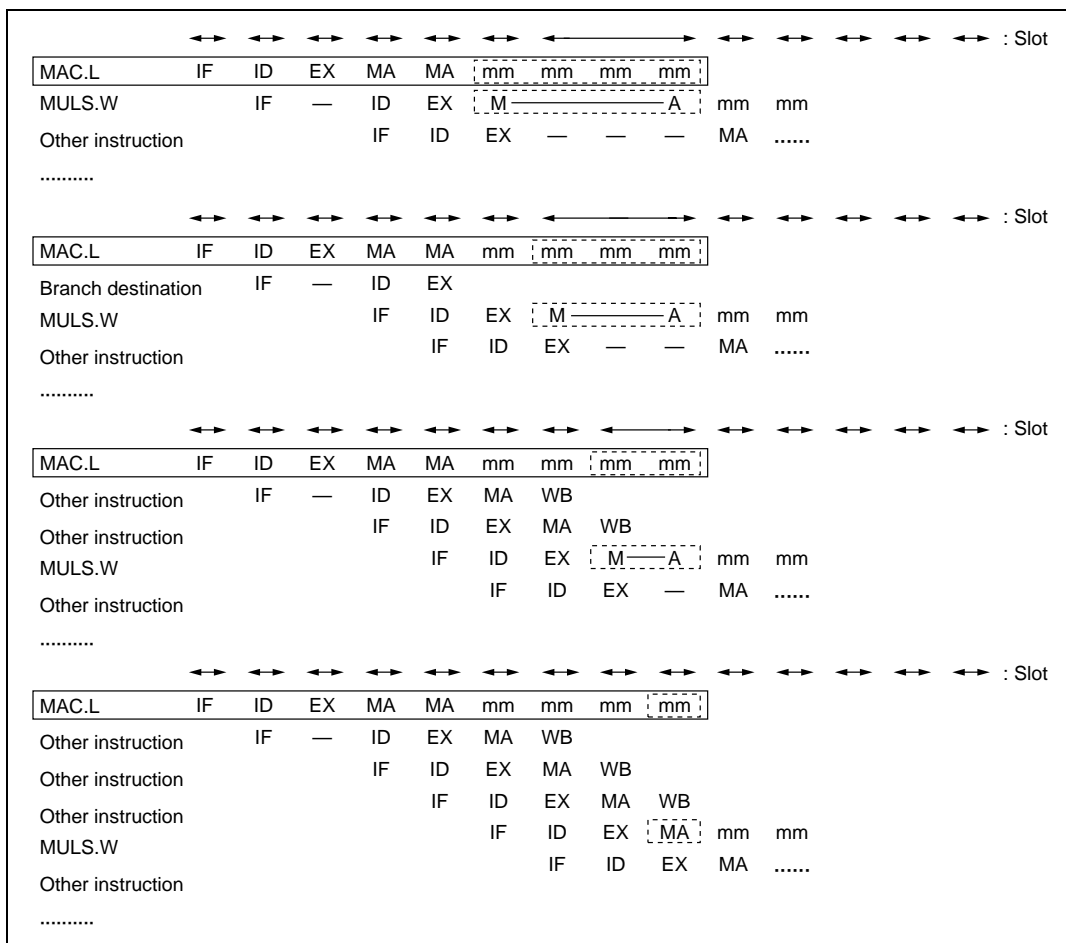


Figure 8.35 MULS.W Instruction Follows Immediately after MAC.L Instruction

(e) STS (register) instruction follows immediately after MAC.L instruction

If the STS instruction is used to store the contents of the MAC register to a general-use register, the STS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of STS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of STS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

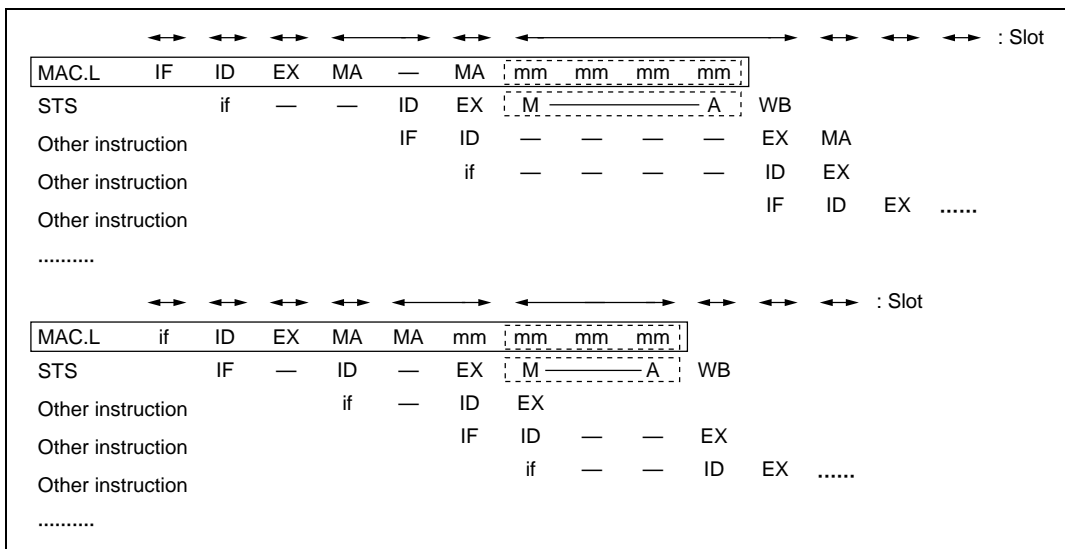


Figure 8.36 STS (Register) Instruction Follows Immediately after MAC.L Instruction

(f) STS.L (memory) instruction follows immediately after MAC.L instruction

If the STS instruction is used to store the contents of the MAC register in memory, the STS instruction will include an MA stage for accessing the multiplier and writing to memory, as described below. Also, the MA of STS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

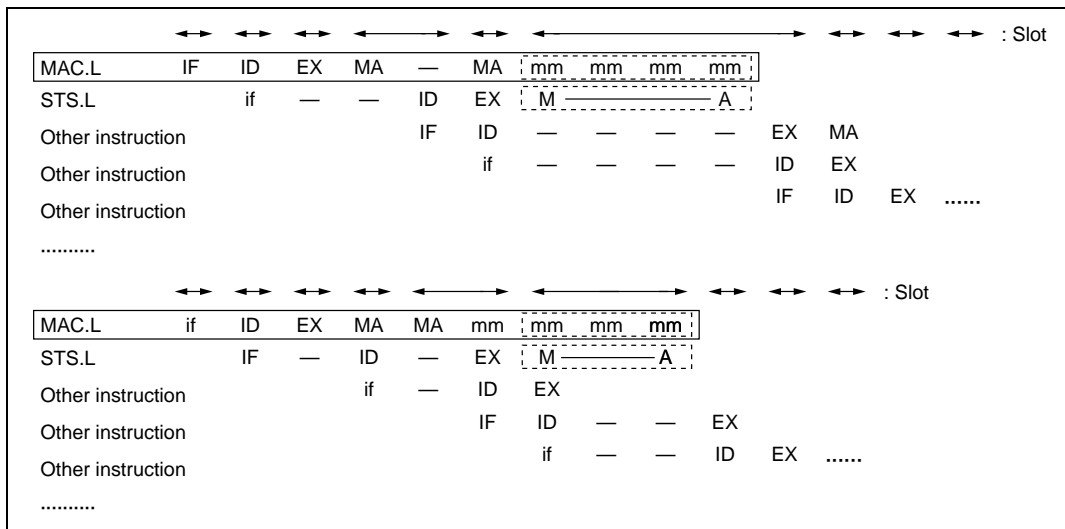


Figure 8.37 STS.L (Memory) Instruction Follows Immediately after MAC.L Instruction

(g) LDS (register) instruction follows immediately after MAC.L instruction

If the LDS instruction is used to load the contents of the MAC register from a general-use register, the LDS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

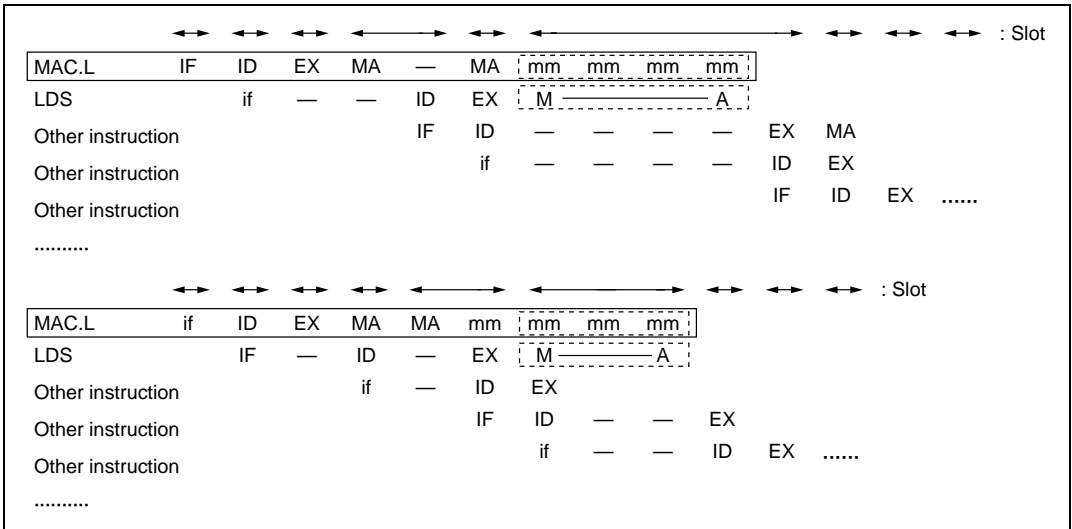


Figure 8.38 LDS (Register) Instruction Follows Immediately after MAC.L Instruction

(h) LDS.L (memory) instruction follows immediately after MAC.L instruction

If the LDS instruction is used to load the contents of the MAC register from memory, the LDS instruction will include an MA stage for accessing memory and accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

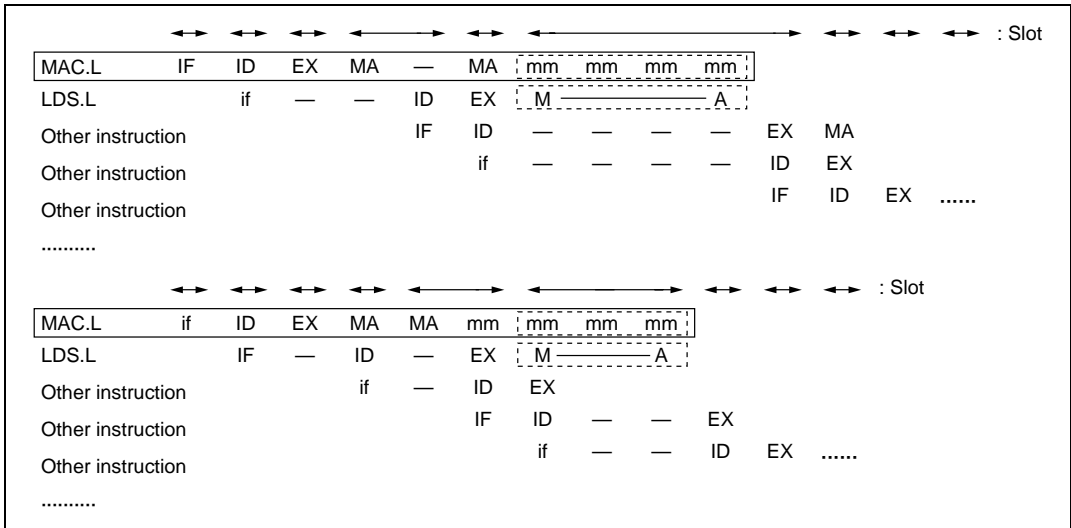


Figure 8.39 LDS.L (Memory) Instruction Follows Immediately after MAC.L Instruction

Multiplication Instructions: Include the following instruction types:

- MULS.W Rm, Rn
- MULU.W Rm, Rn

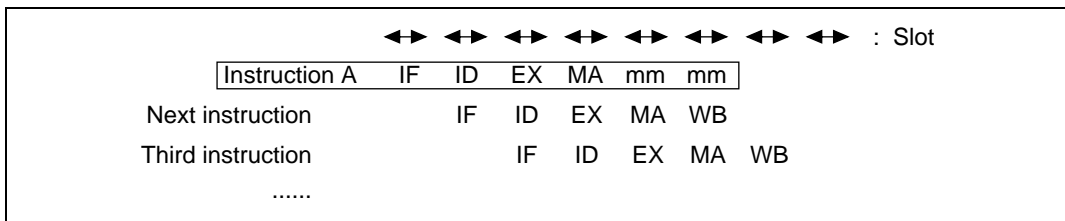


Figure 8.40 Multiplication Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, mm, and mm. The MA accesses the multiplier. mm indicates that the multiplier is operating. mm operates for three cycles after the MA ends, regardless of slot. The MA of the MULS.W instruction, when it contends with IF, splits the slot as described in Section 8.4, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier comes after the MULS.W instruction, the MULS.W instruction may be considered to be a four-stage pipeline instruction of IF, ID, EX, and MA. In such cases, it operates like a normal pipeline. When an instruction that uses the multiplier comes after the MULS.W instruction, however, contention occurs with the multiplier, so operation is different from normal.

The following cases are possible:

- (a) MAC.W instruction follows immediately after MULS.W instruction
- (b) MAC.L instruction follows immediately after MULS.W instruction
- (c) MULS.W instruction follows immediately after MULS.W instruction
- (d) DMULS.L instruction follows immediately after MULS.W instruction
- (e) STS (register) instruction follows immediately after MULS.W instruction
- (f) STS.L (memory) instruction follows immediately after MULS.W instruction
- (g) LDS (register) instruction follows immediately after MULS.W instruction
- (h) LDS.L (memory) instruction follows immediately after MULS.W instruction

(a) MAC.W instruction follows immediately after MULS.W instruction

The second MA of the MAC.W instruction does not contend with the mm generated by the preceding multiply instruction.

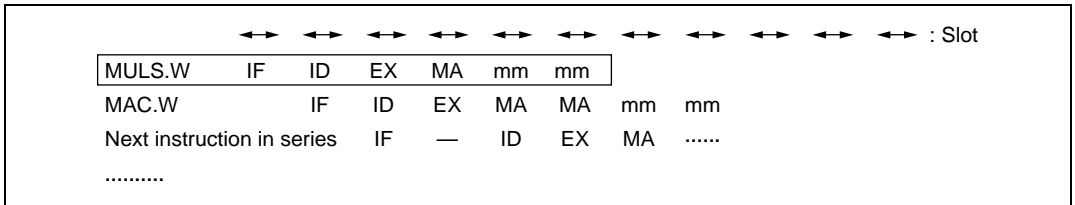


Figure 8.41 MAC.W Instruction Follows Immediately after MULS.W Instruction

(b) MAC.L instruction follows immediately after MULS.W instruction

The second MA of the MAC.W instruction does not contend with the mm generated by the preceding multiply instruction.

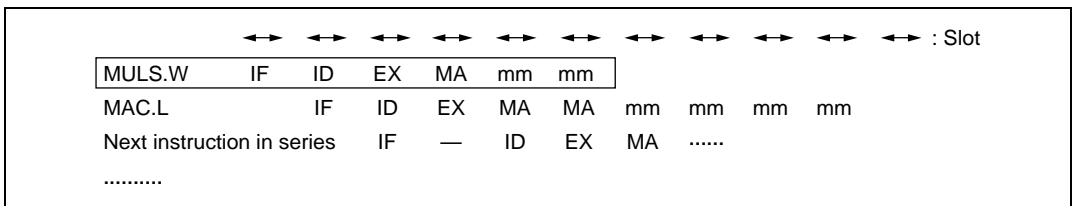


Figure 8.42 MAC.L Instruction Follows Immediately after MULS.W Instruction

(c) MULS.W instruction follows immediately after MULS.W instruction

The MULS.W instruction has an MA stage for accessing the multiplier. If contention with the MA of the other MULS.W occurs during the MULS.W instruction's multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. If there is one or more instruction that does not use the multiplier located between MULS.W and MULS.W, no contention occurs between MULS.W and MULS.W and there is no delay. Note that the slot splits if there is contention between the MA of MULS.W and IF.

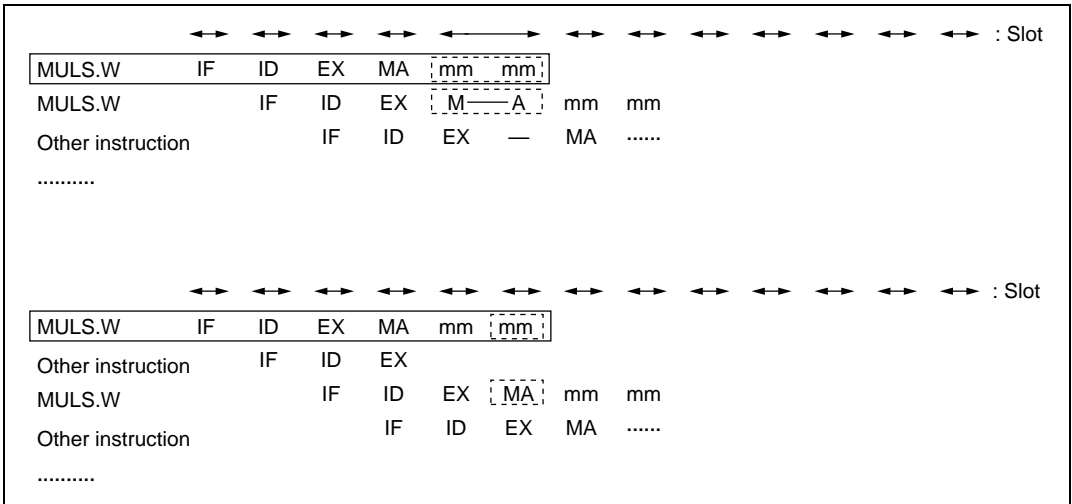


Figure 8.43 MULS.W Instruction Follows Immediately after MULS.W Instruction (1)

If the MA of the MULS.W instruction is delayed until the mm finishes, and that MA contends with IF, the slot splits normally. Refer to the diagram below. This diagram takes into account the possibility of contention between MA and IF.

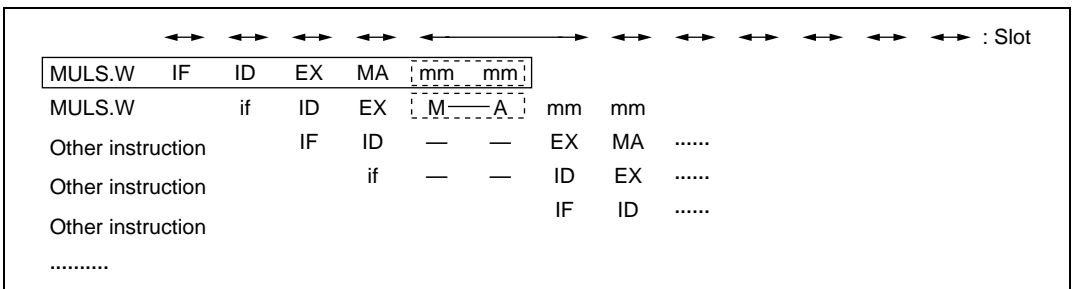


Figure 8.44 MULS.W Instruction Follows Immediately after MULS.W Instruction (2)

(d) DMULS.L instruction follows immediately after MULS.W instruction

The second MA of the DMULS.L accesses the multiplier, but there is no contention with the mm generated by the MULS.W instruction.

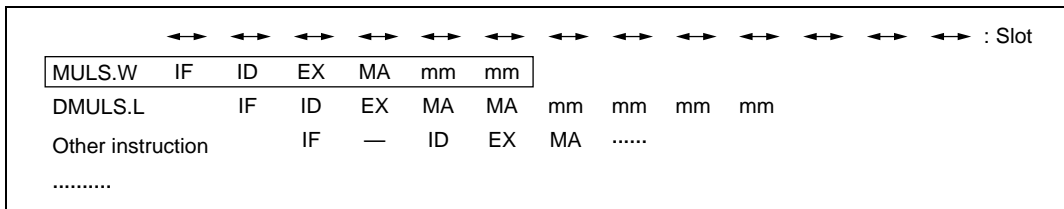


Figure 8.45 DMULS.L Instruction Follows Immediately after MULS.W Instruction

(e) STS (register) instruction follows immediately after MULS.W instruction

If the STS instruction is used to store the contents of the MAC register to a general-use register, the STS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of STS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of STS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

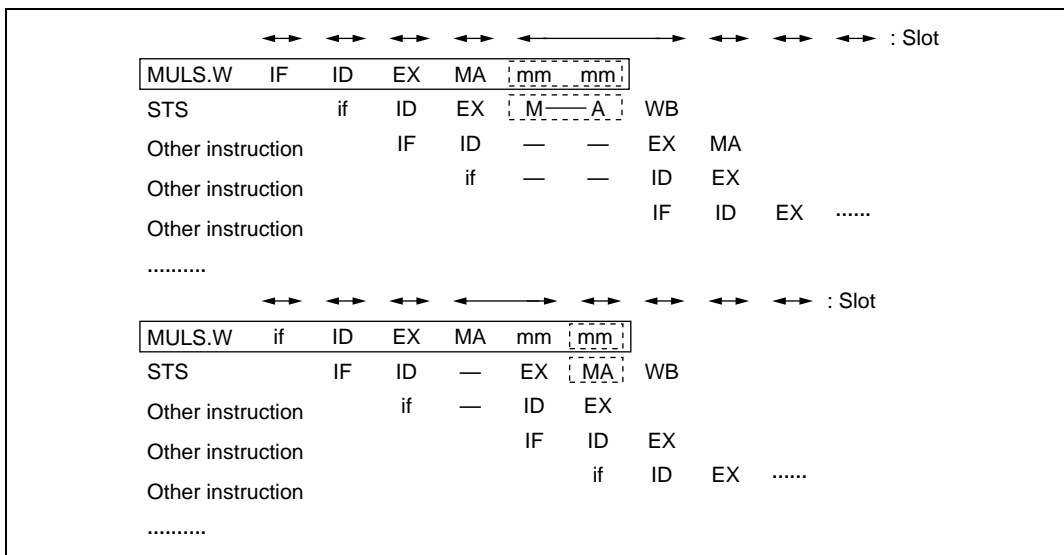


Figure 8.46 STS (Register) Instruction Follows Immediately after MULS.W Instruction

(f) STS.L (memory) instruction follows immediately after MULS.W instruction

If the STS instruction is used to store the contents of the MAC register in memory, the STS instruction will include an MA stage for accessing the multiplier and writing to memory, as described below. Also, the MA of STS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

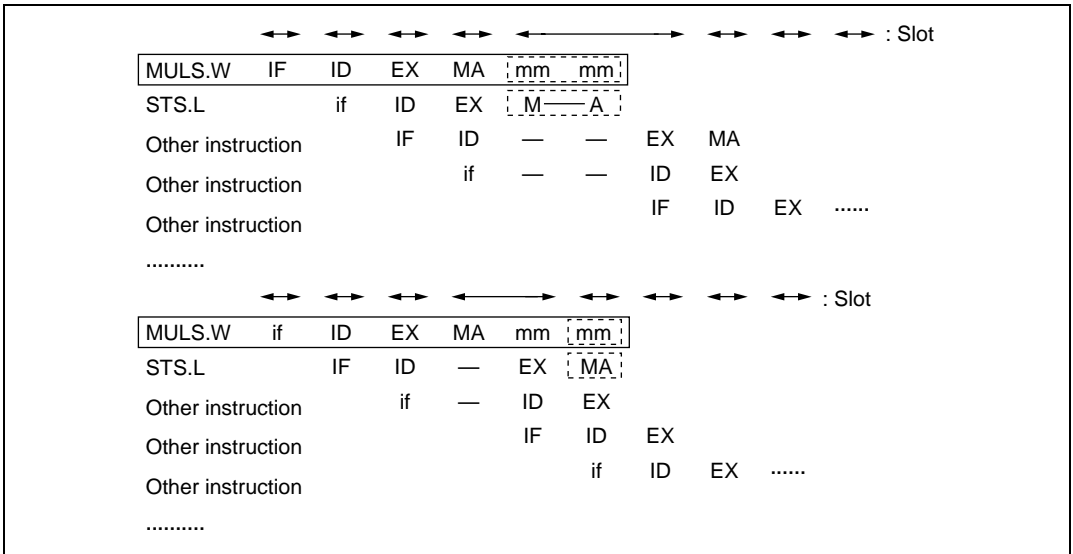


Figure 8.47 STS.L (Memory) Instruction Follows Immediately after MULS.W Instruction

(g) LDS (register) instruction follows immediately after MULS.W instruction

If the LDS instruction is used to load the contents of the MAC register from a general-use register, the LDS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

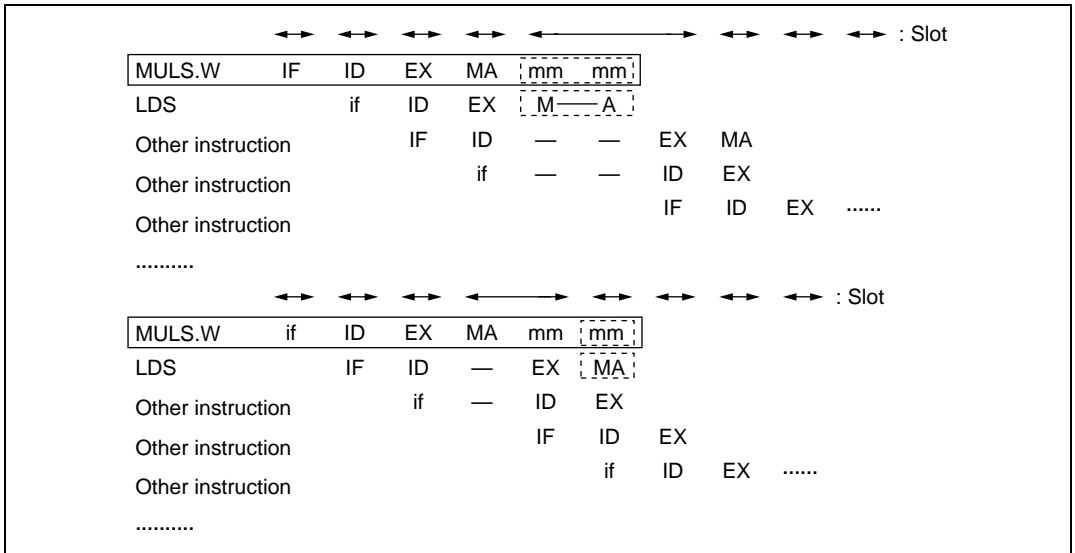


Figure 8.48 LDS (Register) Instruction Follows Immediately after MULS.W Instruction

(h) LDS.L (memory) instruction follows immediately after MULS.W instruction

If the LDS instruction is used to load the contents of the MAC register from memory, the LDS instruction will include an MA stage for accessing memory and accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

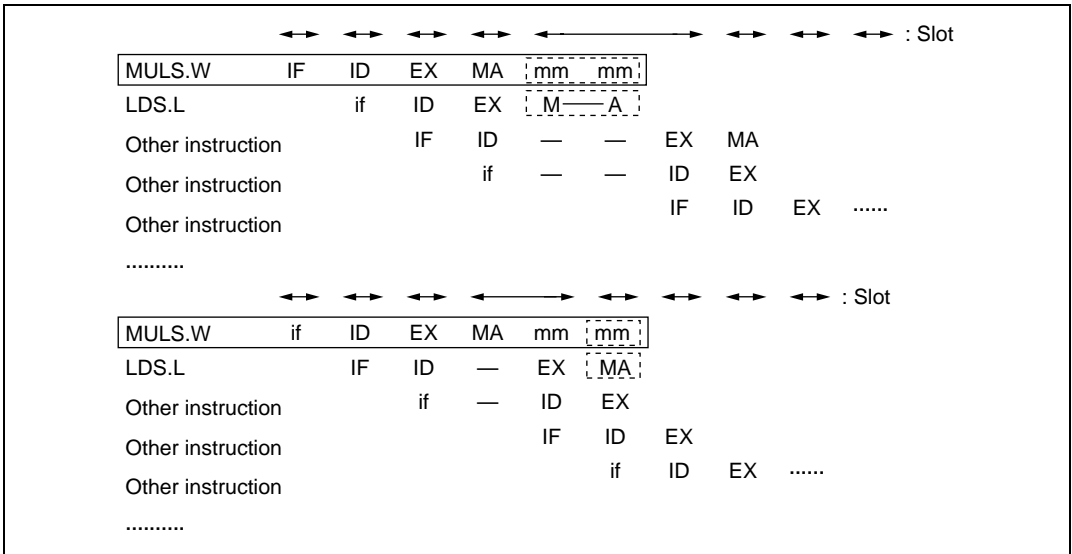


Figure 8.49 LDS.L (Memory) Instruction Follows Immediately after MULS.W Instruction

Double-Length Multiplication Instructions: Include the following instruction types:

- DMULS.L Rm, Rn
- DMULU.L Rm, Rn
- MUL.L Rm, Rn

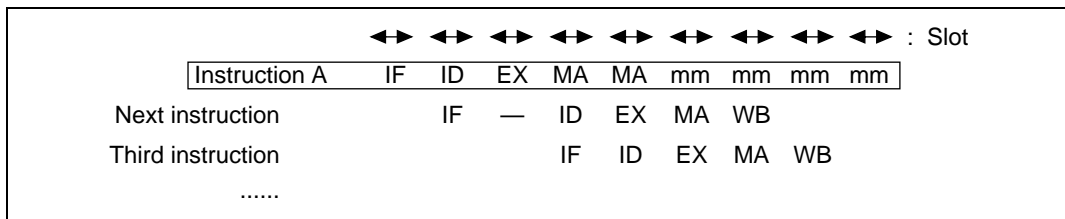


Figure 8.50 Multiplication Instruction Pipeline

The pipeline has nine stages: IF, ID, EX, MA, MA, mm, mm, mm, and mm (figure 8.50). The second MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for four cycles after the MA ends, regardless of slot. The ID of the instruction following the DMULS.L instruction is stalled for 1 slot (see the description of the Multiply/Accumulate instruction). The two MA stages of the DMULS.L instruction, when they contend with IF, split the slot as described in section 8.4, Contention between Instruction Fetch (IF) and Memory Access (MA).

When an instruction that does not use the multiplier comes after the DMULS.L instruction, the DMULS.L instruction may be considered to be a five-stage pipeline instruction of IF, ID, EX, MA, and MA. In such cases, it operates like a normal pipeline. When an instruction that uses the multiplier comes after the DMULS.L instruction, however, contention occurs with the multiplier, so operation is different from normal.

The following cases are possible:

- (a) MAC.L instruction follows immediately after DMULS.L instruction
- (b) MAC.W instruction follows immediately after DMULS.L instruction
- (c) DMULS.L instruction follows immediately after DMULS.L instruction
- (d) MULS.W instruction follows immediately after DMULS.L instruction
- (e) STS (register) instruction follows immediately after DMULS.L instruction
- (f) STS.L (memory) instruction follows immediately after DMULS.L instruction
- (g) LDS (register) instruction follows immediately after DMULS.L instruction
- (h) LDS.L (memory) instruction follows immediately after DMULS.L instruction

(a) MAC.L instruction follows immediately after DMULS.L instruction

If the second MA of the MAC.L instruction contends with the mm generated by the preceding multiply instruction, the bus cycle of that MA is extended until the mm finishes (M -- A in the diagram below), thereby forming a single slot.

If there are two or more instructions that do not use the multiplier located between the DMULS.L instruction and the MAC.L instruction, no contention occurs between DMULS.L and MAC.L, and there is no delay.

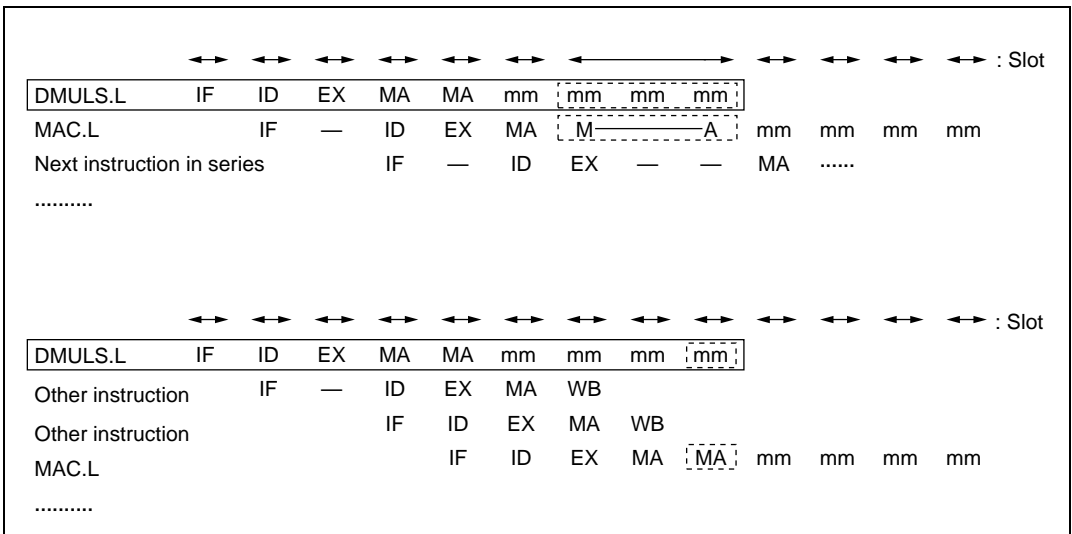


Figure 8.51 MAC.L Instruction Follows Immediately after DMULS.L Instruction

(b) MAC.W instruction follows immediately after DMULS.L instruction

If the second MA of the MAC.W instruction contends with the mm generated by the preceding multiply instruction, the bus cycle of that MA is extended until the mm finishes (M -- A in the diagram below), thereby forming a single slot.

If there are two or more instructions that do not use the multiplier located between the DMULS.L instruction and the MAC.W instruction, no contention occurs between DMULS.L and MAC.W, and there is no delay.

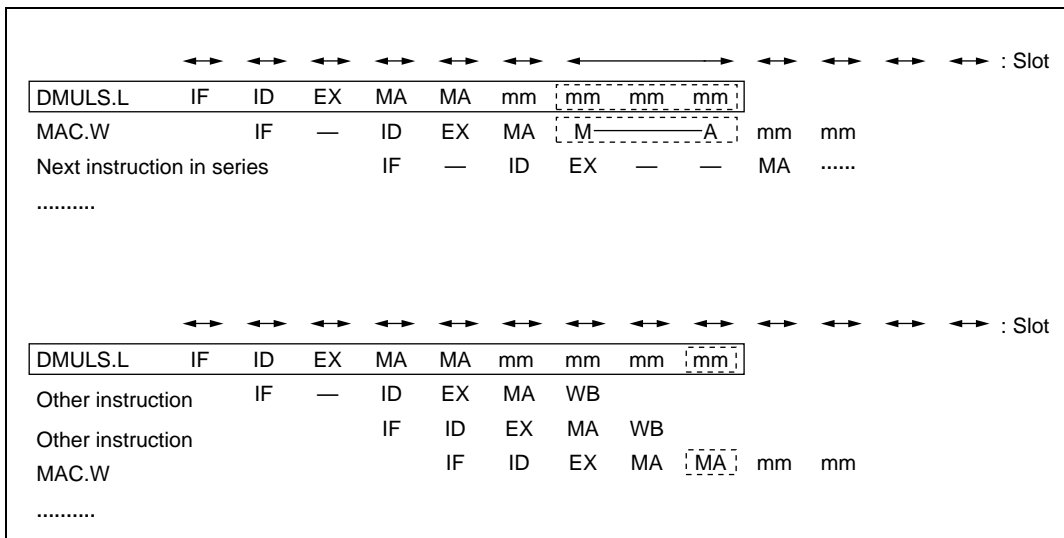


Figure 8.52 MAC.W Instruction Follows Immediately after DMULS.L Instruction

(c) DMULS.L instruction follows immediately after DMULS.L instruction

The DMULS.L instruction has an MA stage for accessing the multiplier. If contention with the MA of DMULS.L occurs during the other DMULS.L instruction's multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. If there are two or more instructions that do not use the multiplier located between DMULS.L and DMULS.L, no contention occurs between DMULS.L and DMULS.L and there is no delay. Note that the slot splits if there is contention between the MA of DMULS.L and IF.

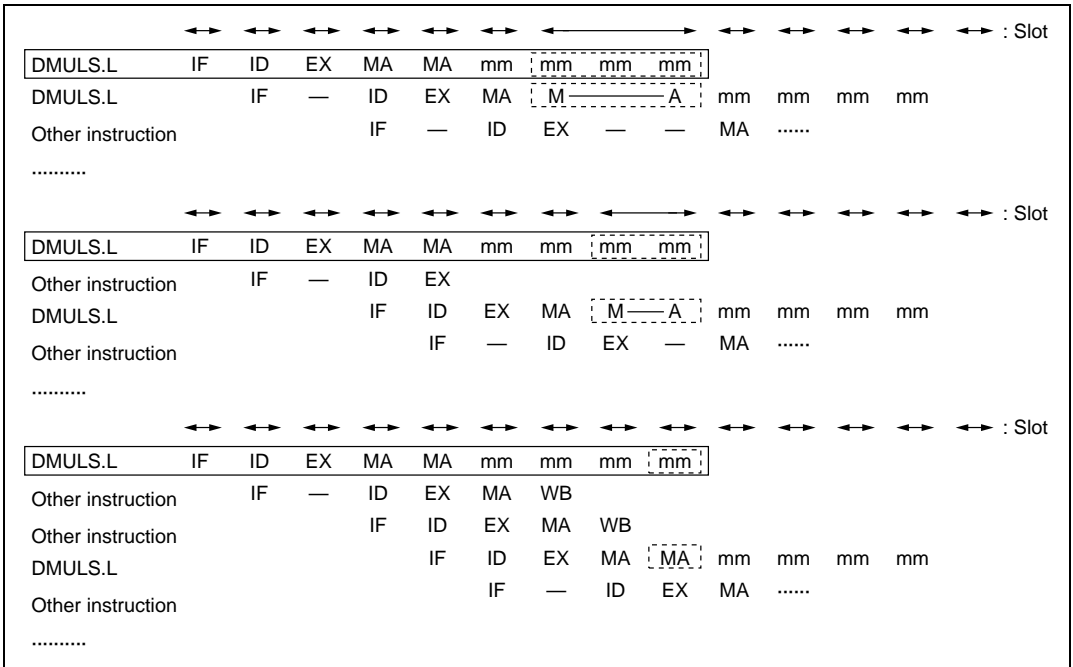


Figure 8.53 DMULS.L Instruction Follows Immediately after DMULS.L Instruction (1)

If the MA of the DMULS.L instruction is delayed until the mm finishes, and that MA contends with IF, the slot splits normally. Refer to the diagram below. This diagram takes into account the possibility of contention between MA and IF.

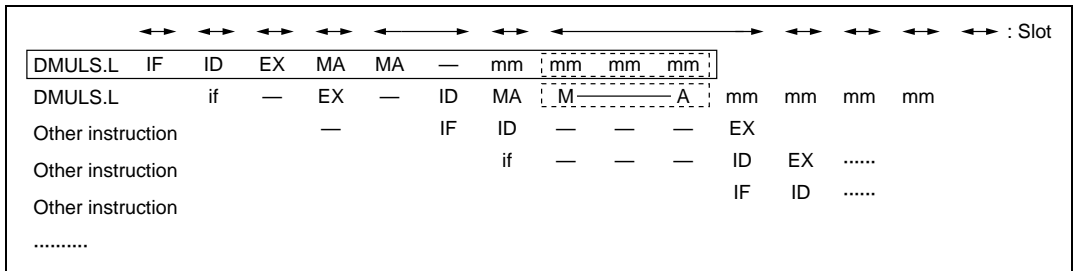


Figure 8.54 DMULS.L Instruction Follows Immediately after DMULS.L Instruction (2)

(d) MULS.W instruction follows immediately after DMULS.L instruction

The MULS.W instruction has an MA stage for accessing the multiplier. If contention with the MA of MULS.W occurs during the DMULS.L instruction's multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. If there are three or more instructions that do not use the multiplier located between DMULS.L and MULS.W, no contention occurs between DMULS.L and MULS.W and there is no delay. Note that the slot splits if there is contention between the MA of MULS.W and IF.

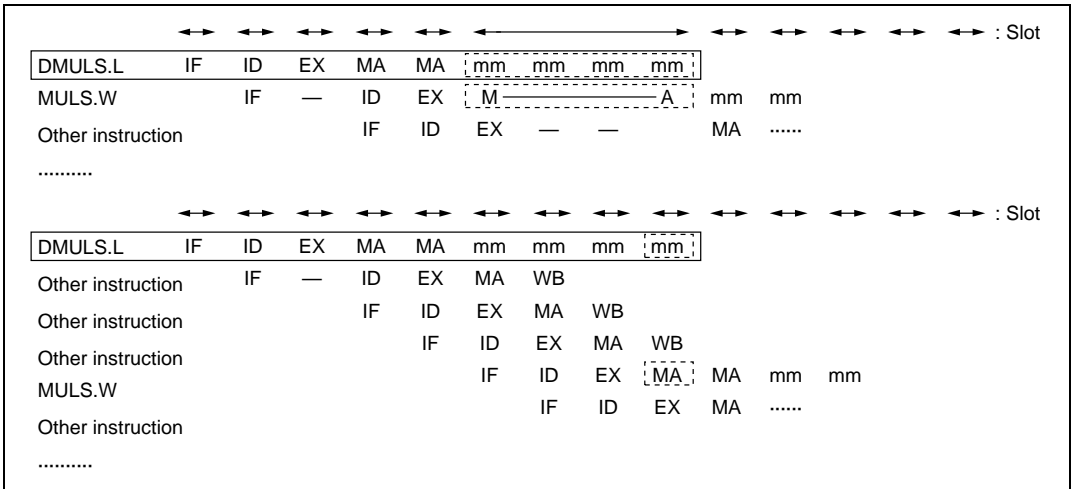


Figure 8.55 MULS.W Instruction Follows Immediately after DMULS.L Instruction (1)

If the MA of the DMULS.L instruction is delayed until the mm finishes, and that MA contends with IF, the slot splits normally. Refer to the diagram below. This diagram takes into account the possibility of contention between MA and IF.

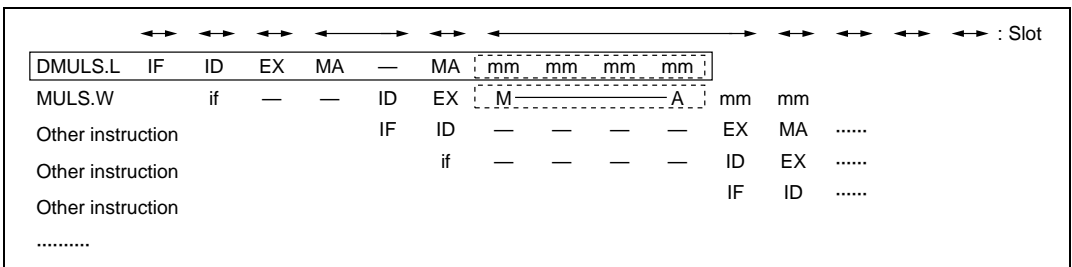


Figure 8.56 MULS.W Instruction Follows Immediately after DMULS.L Instruction (2)

(e) STS (register) instruction follows immediately after DMULS.L instruction

If the STS instruction is used to store the contents of the MAC register to a general-use register, the STS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of STS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of STS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

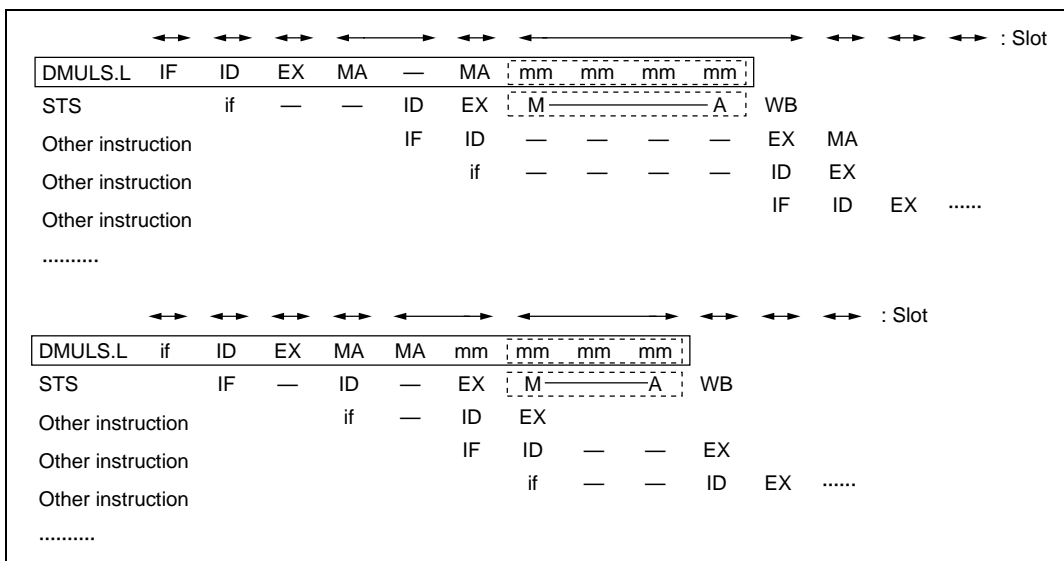


Figure 8.57 STS (Register) Instruction Follows Immediately after DMULS.L Instruction

(f) STS.L (memory) instruction follows immediately after DMULS.L instruction

If the STS instruction is used to store the contents of the MAC register in memory, the STS instruction will include an MA stage for accessing the multiplier and writing to memory, as described below. Also, the MA of STS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

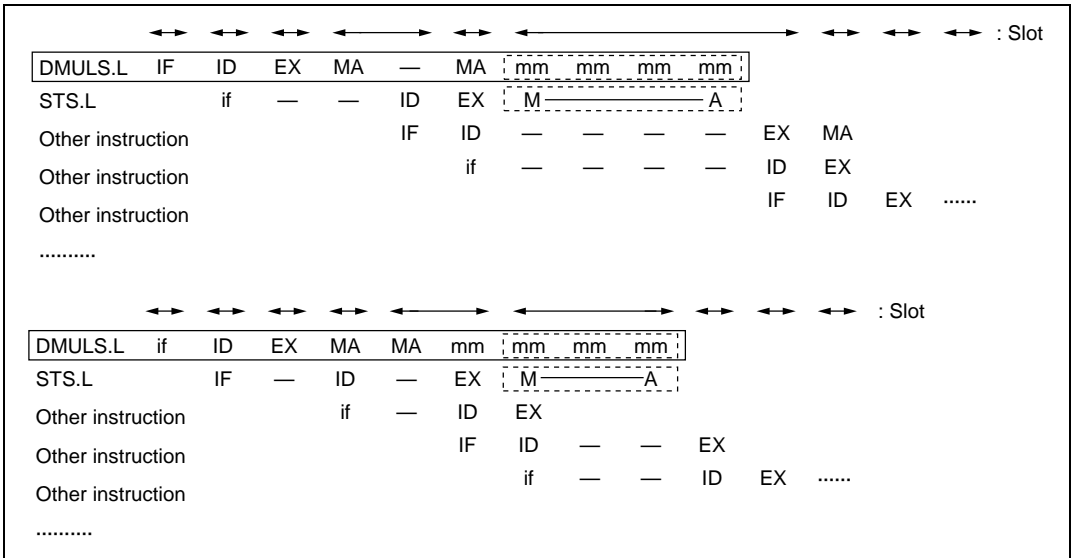


Figure 8.58 STS.L (Memory) Instruction Follows Immediately after DMULS.L Instruction

(g) LDS (register) instruction follows immediately after DMULS.L instruction

If the LDS instruction is used to load the contents of the MAC register from a general-use register, the LDS instruction will include an MA stage for accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

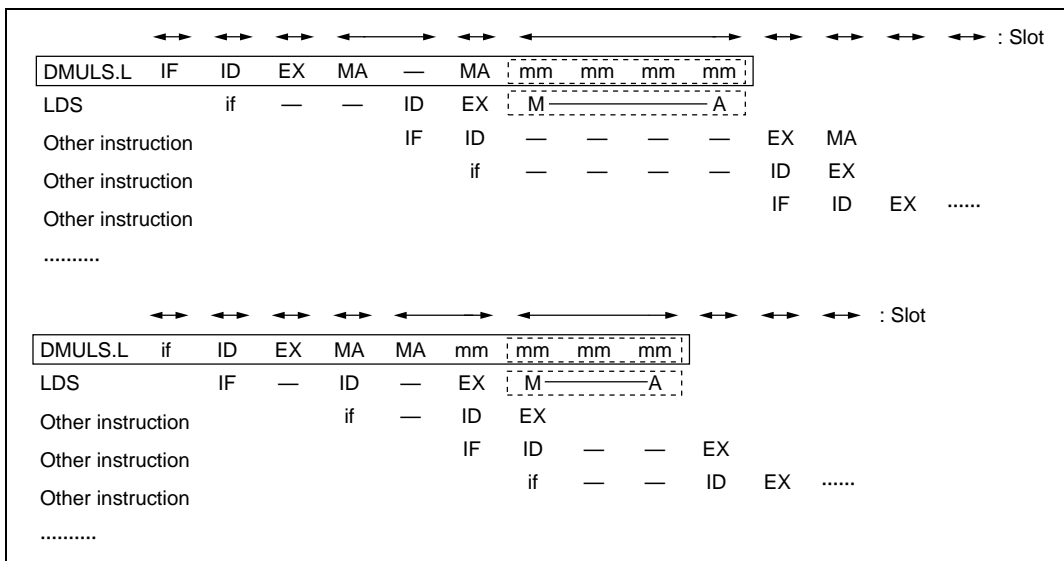


Figure 8.59 LDS (Register) Instruction Follows Immediately after DMULS.L Instruction

(h) LDS.L (memory) instruction follows immediately after DMULS.L instruction

If the LDS instruction is used to load the contents of the MAC register from memory, the LDS instruction will include an MA stage for accessing memory and accessing the multiplier, as described below. If contention with the MA of LDS occurs during the multiplier operation (mm), that MA is delayed until the mm finishes (M -- A in the diagram below), thereby forming a single slot. Also, the MA of LDS contends with IF. This situation is shown in the diagrams below. These diagrams take into account the possibility of contention between MA and IF.

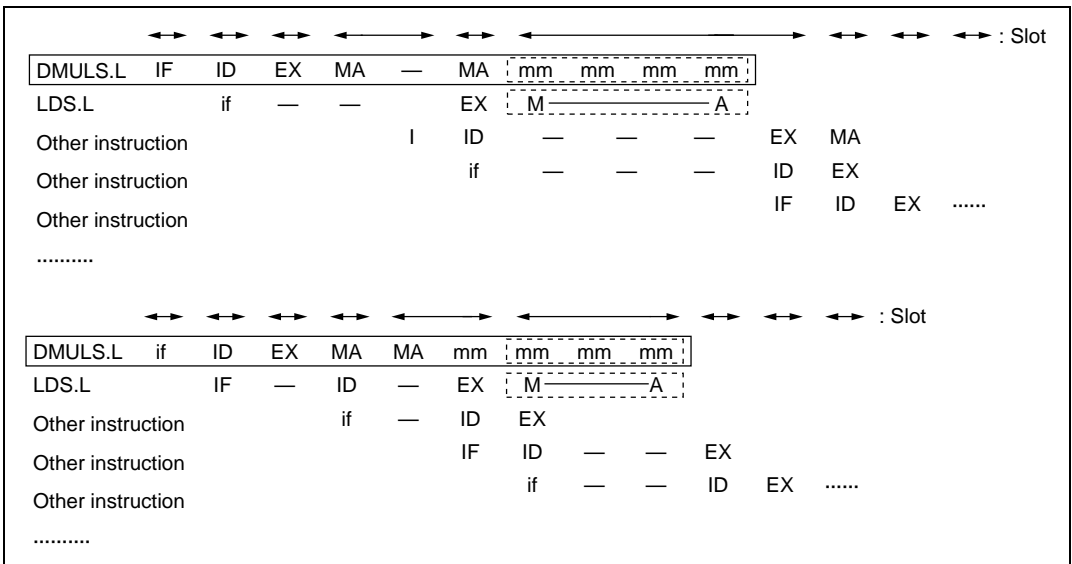


Figure 8.60 LDS.L (Memory) Instruction Follows Immediately after DMULS.L Instruction

8.8.3 Logic Operation Instructions

Register-Register Logic Operation Instructions: Include the following 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

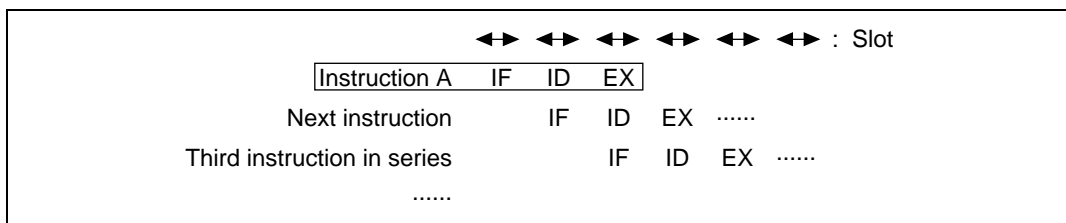


Figure 8.61 Register-Register Logic Operation Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.61). The data operation is completed in the EX stage via the ALU.

Memory Logic Operations Instructions: Include the following instruction types:

- AND.B #imm, @(R0, GBR)
- OR.B #imm, @(R0, GBR)
- TST.B #imm, @(R0, GBR)
- XOR.B #imm, @(R0, GBR)

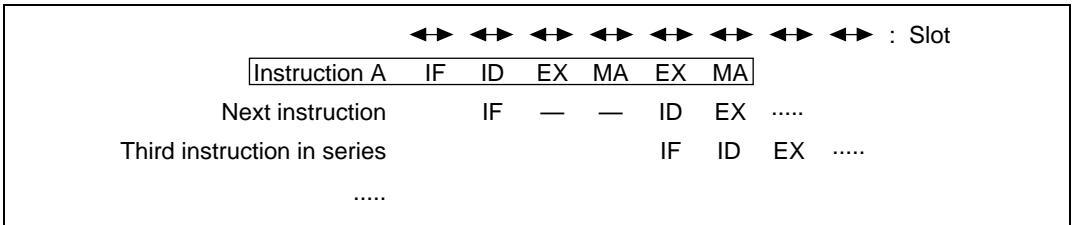


Figure 8.62 Memory Logic Operation Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 8.62). The ID of the next instruction stalls for 2 slots. The MAs of these instructions contend with IF.

TAS Instruction: Includes the following instruction type:

- TAS.B @Rn

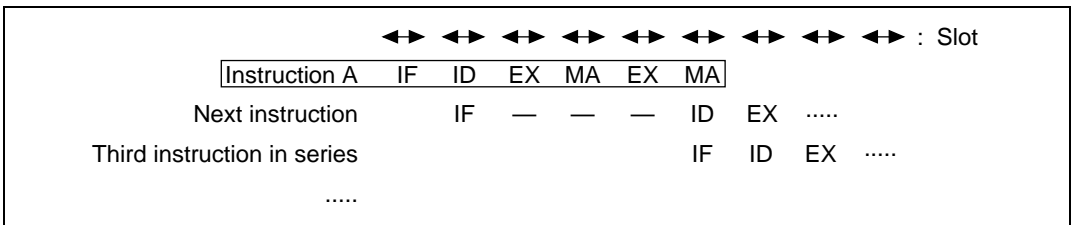


Figure 8.63 TAS Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 8.63). The ID of the next instruction stalls for 3 slots. The MA of the TAS instruction contends with IF.

8.8.4 Shift Instructions

General Shift Instructions: Include the following 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

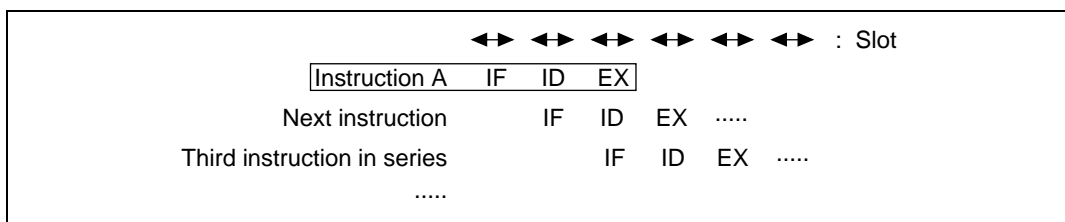


Figure 8.64 General Shift Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.64). The data operation is completed in the EX stage via the ALU.

8.8.5 Branch Instructions

Conditional Branch Instructions: Include the following instruction types:

- BF label
- BT label

The pipeline has three stages: IF, ID, and EX. Condition verification is performed in the ID stage. Conditionally branched instructions are not delay branched.

1. When condition is satisfied

The branch destination address is calculated in the EX stage. The two instructions after the conditional branch instruction (instruction A) are fetched but discarded. The branch destination instruction begins its fetch from the slot following the slot which has the EX stage of instruction A (figure 8.65).

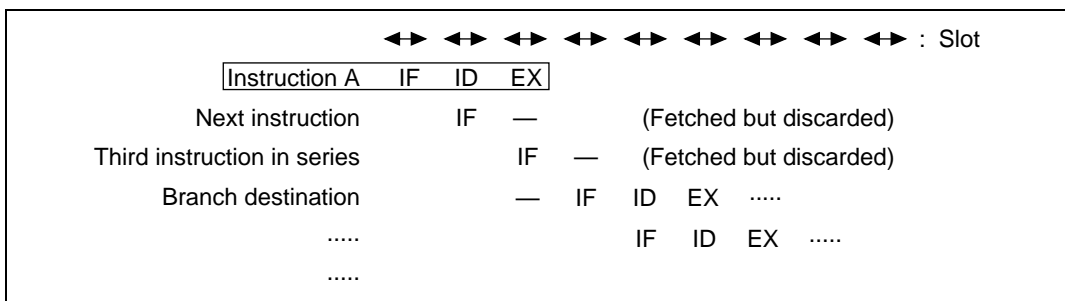


Figure 8.65 Branch Instruction when Condition Is Satisfied

2. When condition is not satisfied

If it is determined that conditions are not satisfied at the ID stage, the EX stage proceeds without doing anything. The next instruction also executes a fetch (figure 8.66).

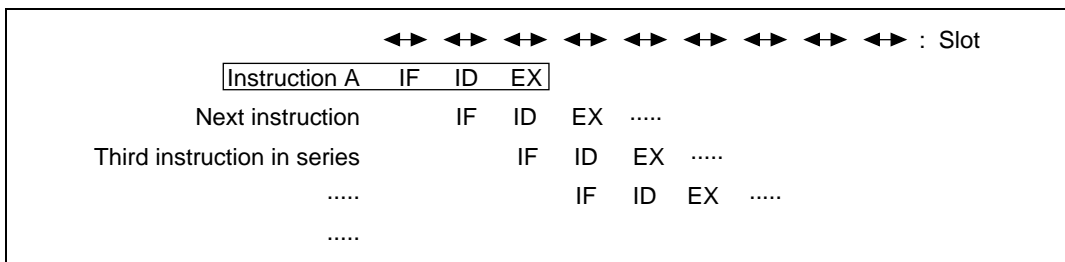


Figure 8.66 Branch Instruction when Condition Is Not Satisfied

Note: The SH-2E always fetches data as longwords. Consequently, a fetch performed by the instruction following the status "1. When condition is satisfied" will overlap two instructions if the address is at the 4n address boundary.

Delayed Conditional Branch Instructions: Include the following instruction types:

- BF/S label
- BT/S label

The pipeline has three stages: IF, ID, and EX. Condition verification is performed in the ID stage.

1. When condition is satisfied

The branch destination address is calculated in the EX stage. The instruction after the conditional branch instruction (instruction A) is fetched and executed, but the instruction after that is fetched and discarded. The branch destination instruction begins its fetch from the slot following the slot which has the EX stage of instruction A (figure 8.67).

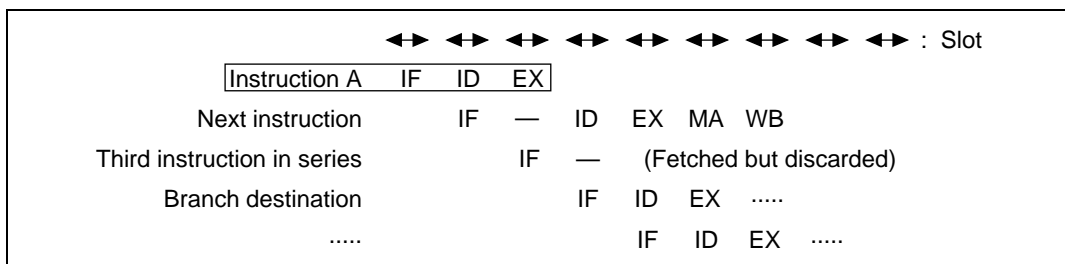


Figure 8.67 Branch Instruction when Condition Is Satisfied

2. When condition is not satisfied

If it is determined that a condition is not satisfied at the ID stage, the EX stage proceeds without doing anything. The next instruction also executes a fetch (figure 8.68).

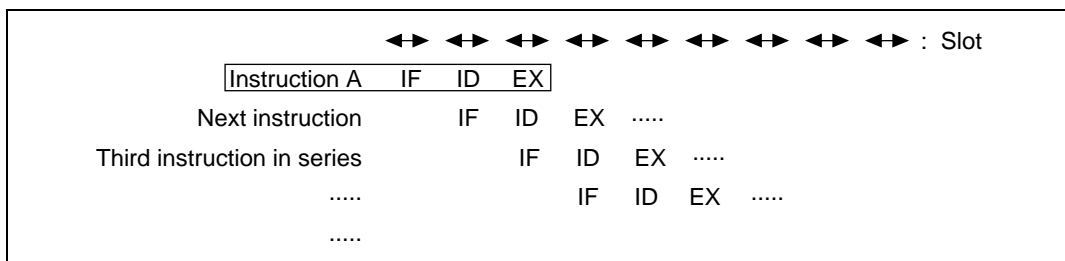


Figure 8.68 Branch Instruction when Condition Is Not Satisfied

Note: The SH-2E always fetches data as longwords. Consequently, a fetch performed by the instruction following the status "1. When condition is satisfied" will overlap two instructions if the address is at the 4n address boundary.

Unconditional Branch Instructions: Include the following instruction types:

- BRA label
- BRAF Rm
- BSR label
- BSRF Rm
- JMP @Rm
- JSR @Rm
- RTS

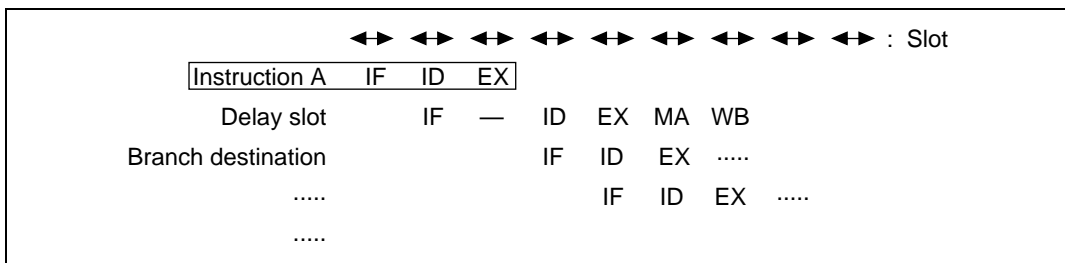


Figure 8.69 Unconditional Branch Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.69). Unconditionally branched instructions are delay branched. The branch destination address is calculated in the EX stage. The instruction following the unconditional branch instruction (instruction A), that is, the delay slot instruction is not fetched and discarded as conditional branch instructions are, but is instead executed. Note that the ID slot of the delay slot instruction does stall for one cycle. The branch destination instruction starts its fetch from the slot after the slot that has the EX stage of instruction A.

8.8.6 System Control Instructions

System Control ALU Instructions: Include the following instruction types:

- CLRT
- LDC Rm,SR
- LDC Rm,GBR
- LDC Rm,VBR
- LDS Rm,PR
- NOP
- SETT
- STC SR,Rn
- STC GBR,Rn
- STC VBR,Rn
- STS PR,Rn

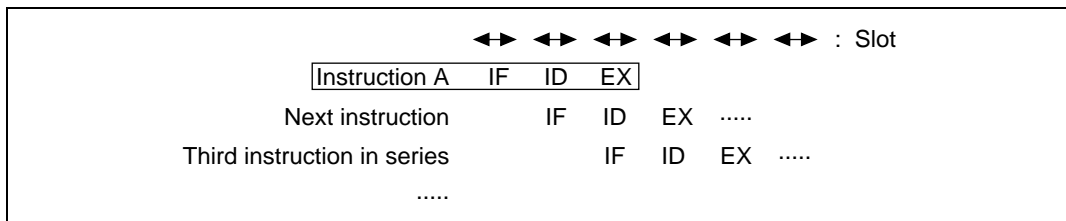


Figure 8.70 System Control ALU Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.70). The data operation is completed in the EX stage via the ALU.

LDC.L Instructions: Include the following instruction types:

- LDC.L @Rm+, SR
- LDC.L @Rm+, GBR
- LDC.L @Rm+, VBR

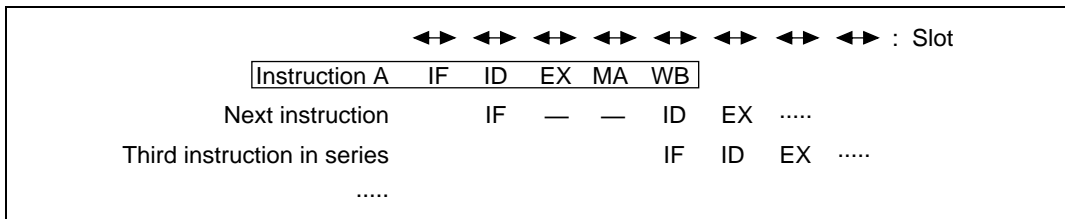


Figure 8.71 LDC.L Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and EX (figure 8.71). The ID of the following instruction is stalled two slots.

STC.L Instructions: Include the following instruction types:

- STC.L SR, @-Rn
- STC.L GBR, @-Rn
- STC.L VBR, @-Rn

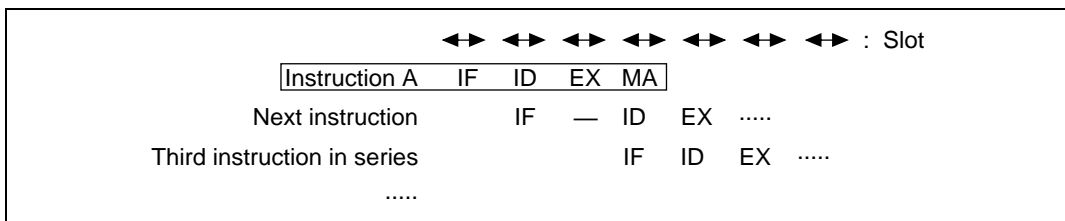


Figure 8.72 STC.L Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.72). The ID of the next instruction is stalled one slot.

LDS.L Instruction (PR): Includes the following instruction type:

- LDS.L @Rm+, PR

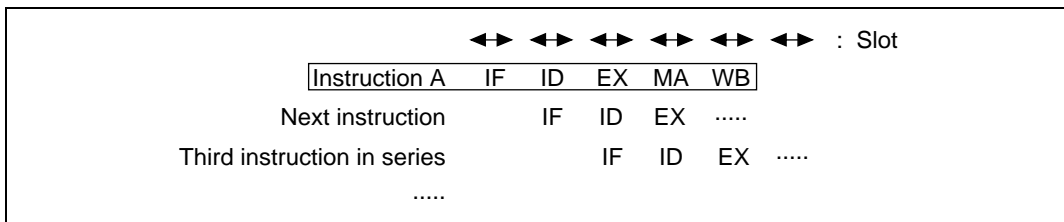


Figure 8.73 LDS.L Instructions (PR) Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.73). It is the same as an ordinary load instruction.

STS.L Instruction (PR): Includes the following instruction type:

- STS.L PR, @-Rn

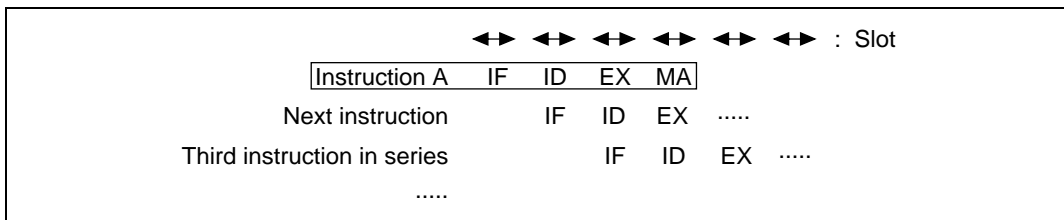


Figure 8.74 STS.L Instruction (PR) Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.74). It is the same as an ordinary load instruction.

Register → MAC Transfer Instructions: Include the following instruction types:

- CLRMAC
- LDS Rm, MACH
- LDS Rm, MACL

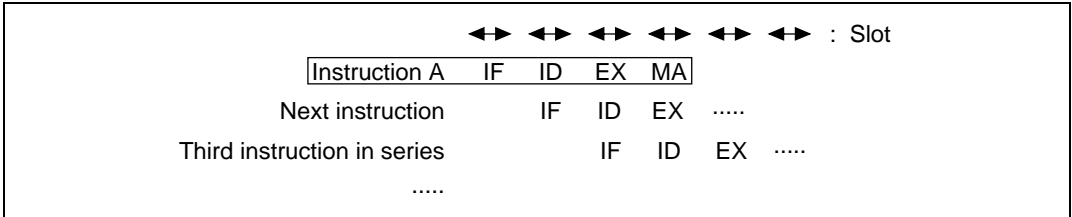


Figure 8.75 Register → MAC Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.75). MA is a stage for accessing the multiplier. MA contends with IF. This makes it the same as ordinary store instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

Memory → MAC Transfer Instructions: Include the following instruction types:

- LDS.L @Rm+, MACH
- LDS.L @Rm+, MACL

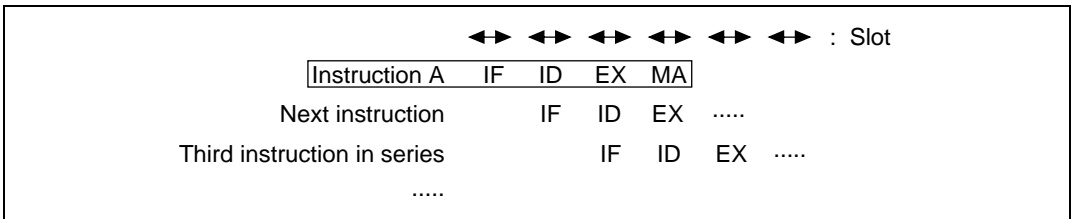


Figure 8.76 Memory → MAC Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.76). MA contends with IF. MA is a stage for memory access and multiplier access. This makes it the same as ordinary load instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

MAC → Register Transfer Instructions: Include the following instruction types:

- STS MACH, Rn
- STS MACL, Rn

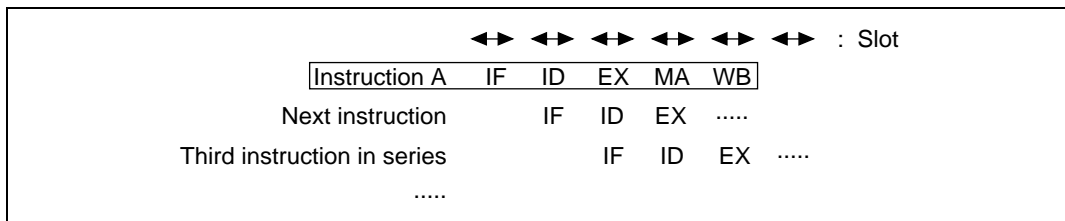


Figure 8.77 MAC → Register Transfer Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.77). MA is a stage for accessing the multiplier. MA contends with IF. This makes it the same as ordinary load instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

MAC → Memory Transfer Instructions: Include the following instruction types:

- STS.L MACH, @-Rn
- STS.L MACL, @-Rn

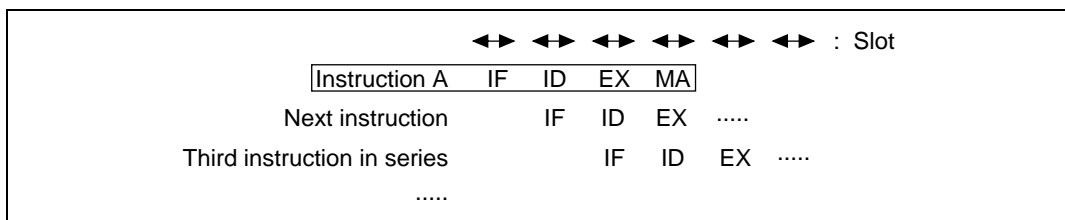


Figure 8.78 MAC → Memory Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.78). MA is a stage for accessing the memory and multiplier. MA contends with IF. This makes it the same as ordinary store instructions. Since the multiplier does contend with the MA, however, the items noted for the multiplication, Multiply/Accumulate, double-length multiplication, and double-length multiply/accumulate instructions apply.

The pipeline has three stages: IF, ID and EX (figure 8.81). It is issued until the IF of the next instruction. After the SLEEP instruction is executed, the CPU enters sleep mode or standby mode.

8.8.7 Exception Processing

Interrupt Exception Processing: The interrupt is received during the ID stage of the instruction and everything after the ID stage is replaced by the interrupt exception processing sequence. The pipeline has ten stages: IF, ID, EX, EX, MA, MA, EX, MA, EX, and EX (figure 8.82). Interrupt exception processing is not a delayed branch. In interrupt exception processing, an overrun fetch (IF) occurs. In branch destination instructions, the IF starts from the slot that has the final EX in the interrupt exception processing.

Interrupt sources are external interrupt request pins such as NMI, user breaks, IRQ, and on-chip peripheral module interrupts.

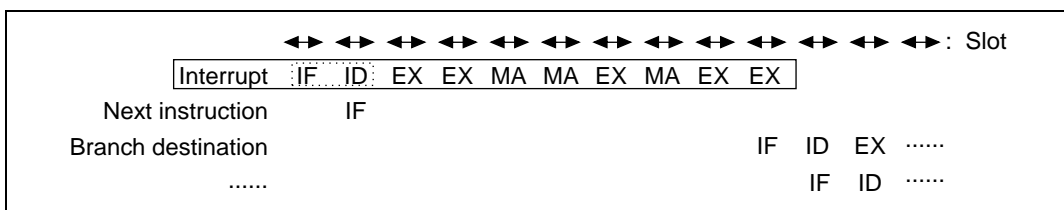


Figure 8.82 Interrupt Exception Processing Pipeline

Address Error Exception Processing: The address error is received during the ID stage of the instruction and everything after the ID stage is replaced by the address error exception processing sequence. The pipeline has ten stages: IF, ID, EX, EX, MA, MA, EX, MA, EX, and EX (figure 8.83). Address error exception processing is not a delayed branch. In address error exception processing, an overrun fetch (IF) occurs. In branch destination instructions, the IF starts from the slot that has the final EX in the address error exception processing.

Address errors are caused by instruction fetches and by data reads or writes. See the Hardware Manual for information on the causes of address errors.

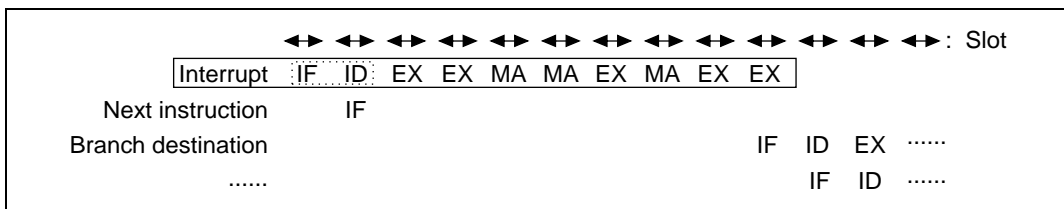


Figure 8.83 Address Error Exception Processing Pipeline

Illegal Instruction Exception Processing: The illegal instruction is received during the ID stage of the instruction and everything after the ID stage is replaced by the illegal instruction exception processing sequence. The pipeline has nine stages: IF, ID, EX, EX, MA, MA, MA, EX, and EX (figure 8.84). Illegal instruction exception processing is not a delayed branch. In illegal instruction exception processing, overrun fetches (IF) occur. Whether there is an IF only in the next instruction or in the one after that as well depends on the instruction that was to be executed. In branch destination instructions, the IF starts from the slot that has the final EX in the illegal instruction exception processing.

Illegal instruction exception processing is caused by ordinary illegal instructions and by instructions with illegal slots. When undefined code placed somewhere other than the slot directly after the delayed branch instruction (called the delay slot) is decoded, ordinary illegal instruction exception processing occurs. When undefined code placed in the delay slot is decoded or when an instruction placed in the delay slot to rewrite the program counter is decoded, an illegal slot instruction occurs.

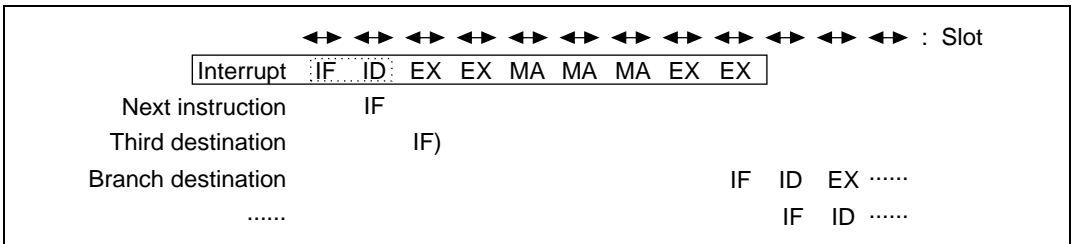


Figure 8.84 Illegal Instruction Exception Processing Pipeline

8.8.8 Relationship between Floating-point Instructions and FPU-related CPU Instructions

FPUL Load Instructions: Include the following instruction types:

- LDS Rm,FPUL
- LDS.L @Rm+,FPUL

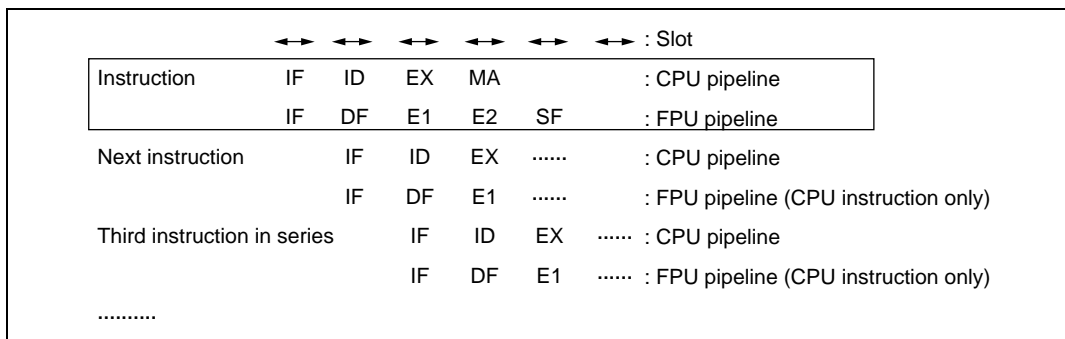


Figure 8.85 FPUL Load Instruction Pipeline

The CPU pipeline has four stages, IF, ID, EX, and MA (figure 8.85) ; and the FPU pipeline has five stages, IF, DF, E1, E2, and SF. The CPU MA stage contends with IF. Contention will also result if an instruction that reads FPUL follows immediately after this instruction.

FPSCR Load Instructions: Include the following instruction types:

- LDS Rm,FPSCR
- LDS.L @Rm+,FPSCR

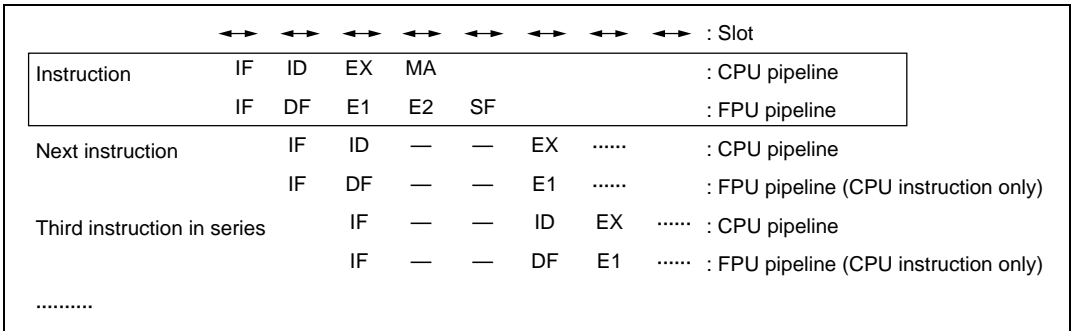


Figure 8.86 FPSCR Load Instruction Pipeline

The CPU pipeline has four stages, IF, ID, EX, and MA (figure 8.86) ; and the FPU pipeline has five stages, IF, DF, E1, E2, and SF. Contention occurs as shown in Figure 8.11, and execution of the next instruction is delayed by two slots.

FPUL Store Instruction (STS): Include the following instruction type:

- STS FPUL,Rn

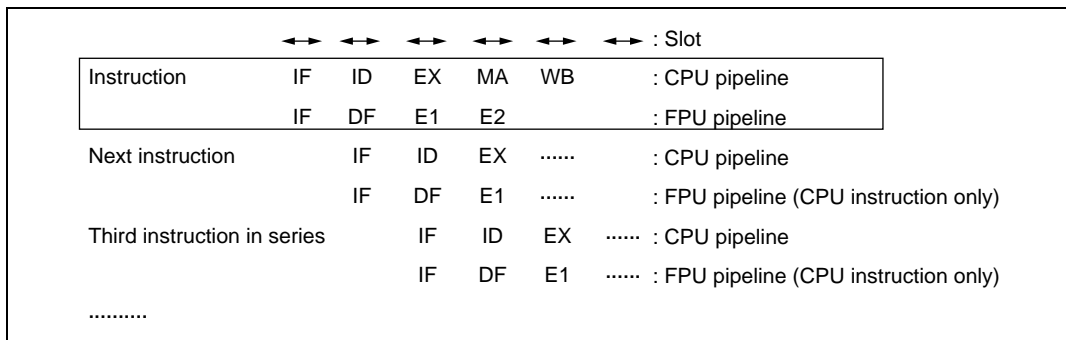


Figure 8.87 FPUL Store Instruction (STS) Pipeline

The CPU pipeline has five stages, IF, ID, EX, MA, and MB (figure 8.87) ; and the FPU pipeline has four stages, IF, DF, E1, and E2. The CPU MA stage contends with IF. Contention will also result if an instruction that uses the destination of this instruction follows immediately after it.

FPSCR Store Instruction (STS): Include the following instruction type:

- STS FPSCR,Rn

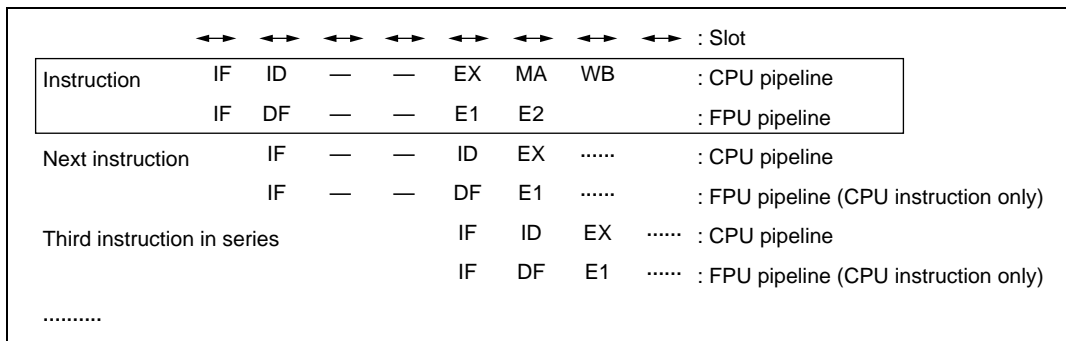


Figure 8.89 FPSCR Store Instruction (STS) Pipeline

The CPU pipeline has five stages, IF, ID, EX, MA, and MB (figure 8.89) ; and the FPU pipeline has four stages, IF, DF, E1, and E2. Contention occurs as shown in Figure 8.12, and execution of the next instruction is delayed by two slots. The CPU MA stage contends with IF. Contention will also result if an instruction that uses the destination of this instruction follows immediately after it.

FPSCR Store Instruction (STS.L): Include the following instruction type:

- STS.L FPSCR,@-Rn

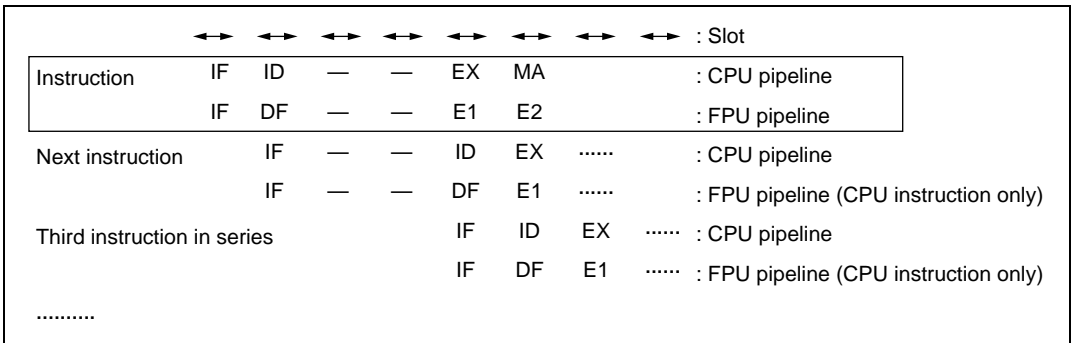


Figure 8.90 FPSCR Store Instruction (STS.L) Pipeline

The CPU pipeline has four stages, IF, ID, EX, and MA (figure 8.90) ; and the FPU pipeline has four stages, IF, DF, E1, and E2. Contention occurs as shown in Figure 8.12, and execution of the next instruction is delayed by two slots. The CPU MA stage contends with IF.

Floating-point Register Transfer Instructions: Include the following instruction types:

- FLDS FRm,FPUL
- FMOV FRm,FRn
- FSTS FPUL,FRn

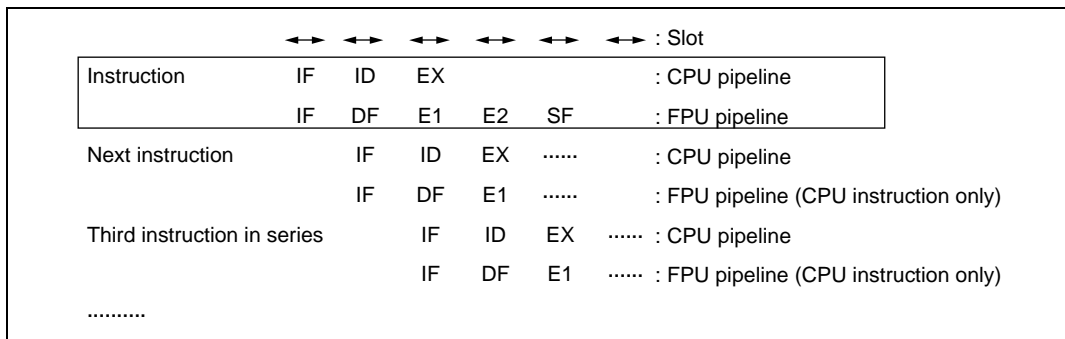


Figure 8.91 Floating-point Register Transfer Instruction Pipeline

The CPU pipeline has three stages, IF, ID, and EX (figure 8.91) ; and the FPU pipeline has five stages, IF, DF, E1, E2, and SF. Contention occurs if an instruction that reads from the destination of this instruction follows immediately after it.

Floating-point Register Immediate Instructions: Include the following instruction types:

- FLDI0 FR_n
- FMDI1 FR_n

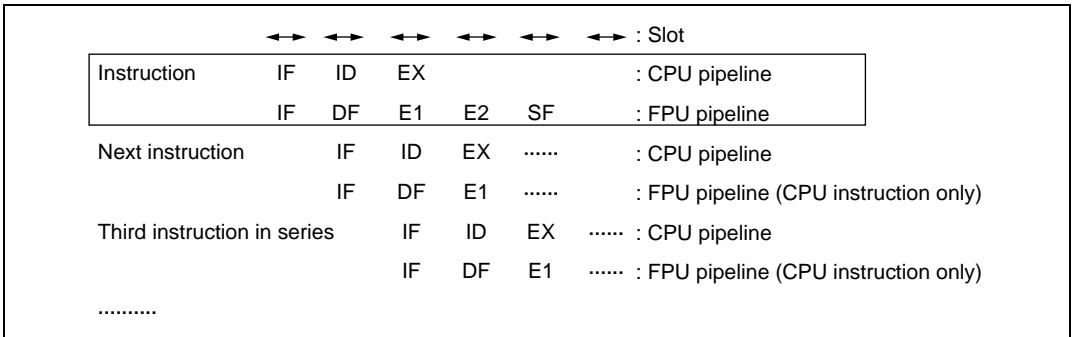


Figure 8.92 Floating-point Register Immediate Instructions

The CPU pipeline has three stages, IF, ID, and EX (figure 8.92) ; and the FPU pipeline has five stages, IF, DF, E1, E2, and SF. Contention occurs if an instruction that reads from the destination of this instruction follows immediately after it.

Floating-point Register Load Instructions: Include the following instruction types:

- FMOV.S @Rm,FRn
- FMOV.S @Rm+,FRn
- FMOV.S @(R0,Rm),FRn

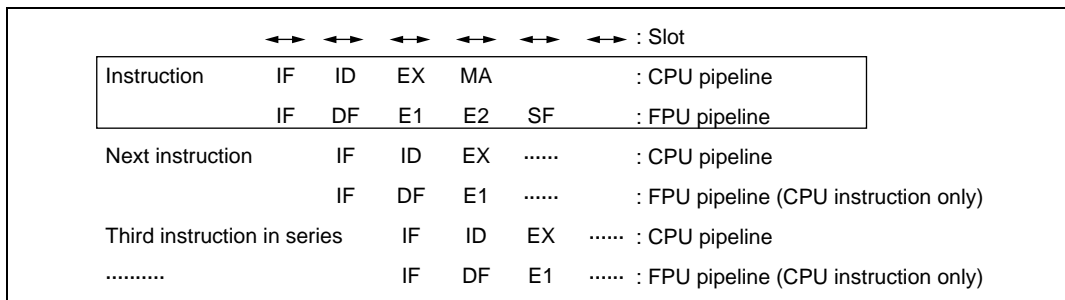


Figure 8.93 Floating-point Register Load Instruction Pipeline

The CPU pipeline has four stages, IF, ID, EX and MA (figure 8.93) ; and the FPU pipeline has five stages, IF, DF, E1, E2, and SF. The CPU MA stage contends with IF. Contention will also result if an instruction that reads from the destination of this instruction follows immediately after it.

Floating-point Register Store Instructions: Include the following instruction types:

- FMOV.S FRm,@Rn
- FMOV.S FRm,@-Rn
- FMOV.S FRm,@(R0,Rn)

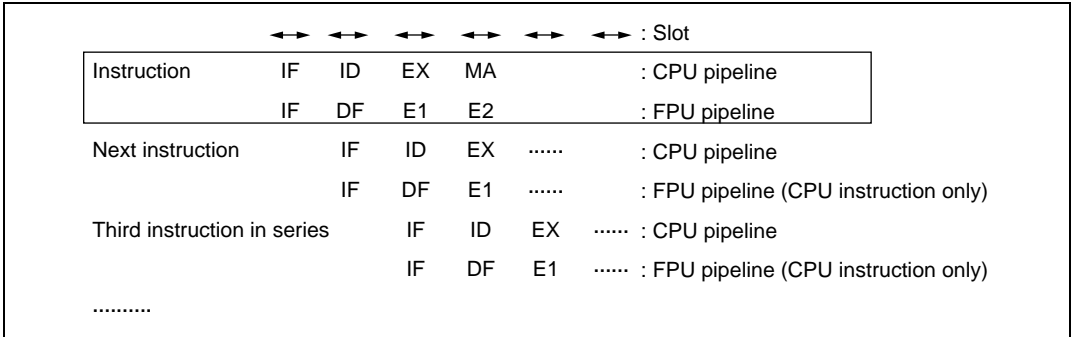


Figure 8.94 Floating-point Register Store Instruction Pipeline

The CPU pipeline has four stages, IF, ID, EX and MA (figure 8.94) ; and the FPU pipeline has four stages, IF, DF, E1, and E2. The CPU MA stage contends with IF.

Floating-point Operation Instructions (Excluding FDIV): Include the following instruction types:

- FABS FRn
- FADD FRm,FRn
- FLOAT FPUL,FRn
- FMAC FR0,FRm,FRn
- FMUL FRm,FRn
- FNEG FRn
- FSUB FRm,FRn
- FTRC FRm,FPUL

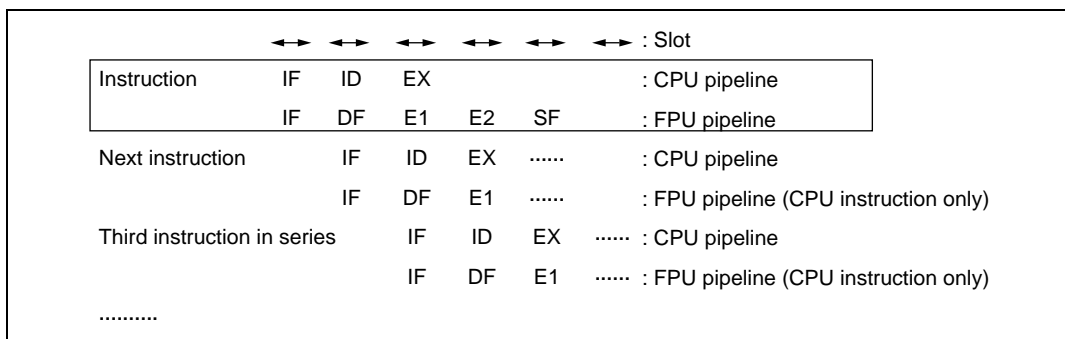


Figure 8.95 Floating-point Operation Instructions (Excluding FDIV) Pipeline

The CPU pipeline has three stages, IF, ID, and EX (figure 8.95) ; and the FPU pipeline has five stages, IF, DF, E1, E2, and SF. Contention occurs if an instruction that reads from the destination of this instruction follows immediately after it.

Floating-point Compare Instructions: Include the following instruction types:

- FCMP/EQ FRm,FRn
- FCMP/GT FRm,FRn

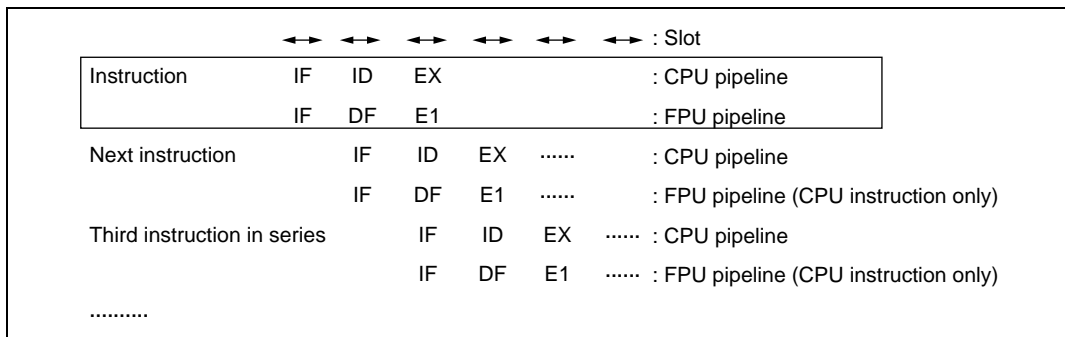


Figure 8.97 Floating-point Compare Instruction Pipeline

The CPU pipeline has three stages, IF, ID, and EX (figure 8.97) ; and the FPU pipeline has three stages, IF, DF, and E1.

Appendix A Instruction Code

A.1 Instruction Set by Addressing Mode

Table A.1 Instruction Set by Addressing Mode

Addressing Mode	Category	Sample Instruction	Types
No operand	—	NOP	8
Direct register addressing	Destination operand only	MOV.T Rn	22
	Source and destination operand	ADD Rm, Rn	42
	Load and store with control register or system register	LDC Rm, SR STS MACH, Rn	18
Indirect register addressing	Source operand only	JMP @Rm	2
	Destination operand only	TAS.B @Rn	1
	Data transfer direct from register	MOV.L Rm, @Rn	8
Post-increment indirect register addressing	Multiply/accumulate operation	MAC.W @Rm+, @Rn+	2
	Data transfer direct from register	MOV.L @Rm+, Rn	4
	Load to control register or system register	LDC.L @Rm+, SR	8
Pre-decrement indirect register addressing	Data transfer direct from register	MOV.L Rm, @-Rn	4
	Store from control register or system register	STC.L SR, @-Rn	8
Indirect register addressing with displacement	Data transfer direct to register	MOV.L Rm, @(disp, Rn)	6
Indirect indexed register addressing	Data transfer direct to register	MOV.L Rm, @(R0, Rn)	8
Indirect GBR addressing with displacement	Data transfer direct to register	MOV.L R0, @(disp, GBR)	6
Indirect indexed GBR addressing	Immediate data transfer	AND.B #imm, @(R0, GBR)	4
PC relative addressing with displacement	Data transfer direct to register	MOV.L @(disp, PC), Rn	3

Addressing Mode	Category	Sample Instruction		Types
PC relative addressing with Rn	Branch instruction	BRAF	Rn	2
PC relative addressing	Branch instruction	BRA	label	6
Immediate addressing	Load to register	FLDI0	FRn	2
	Arithmetic logical operations direct with register	ADD	#imm, Rn	7
	Specify exception processing vector	TRAPA	#imm	1
				Total: 172

Note: Figures not in parentheses () indicate the number of instructions for the SH-3E and figures in parentheses () indicate the number of instructions for the SH-3.

A.1.1 No Operand

Table A.2 No Operand

Instruction	Operation	Code	Cycles	T Bit
CLRT	0 → T	0000000000001000	1	0
CLRMAC	0 → MACH, MACL	000000000101000	1	—
DIV0U	0 → M/Q/T	000000000011001	1	0
NOP	No operation	000000000001001	1	—
RTE	Delayed branching, Stack area → PC/SR	000000000101011	4	—
RTS	Delayed branching, PR → PC	000000000001011	2	—
SETT	1 → T	000000000011000	1	1
SLEEP	Sleep	000000000011011	3	—

A.1.2 Direct Register Addressing

Table A.3 Destination Operand Only

Instruction	Operation	Code	Cycles	T Bit
CMP/PL Rn	$Rn > 0, 1 \rightarrow T$	0100nnnn00010101	1	Comparison result
CMP/PZ Rn	$Rn \geq 0, 1 \rightarrow T$	0100nnnn00010001	1	Comparison result
DT Rn	$Rn - 1 \rightarrow Rn$, when Rn is 0, $1 \rightarrow T$. When Rn is nonzero, $0 \rightarrow T$	0100nnnn00010000	1	Comparison result
FABS FRn	abs (FRn \rightarrow FRn)	1111nnnn01011101	1	—
FLOAT FPUL, FRn	(float) FPUL \rightarrow FRn	1111nnnn001011101	1	—
FNEG FRn	$-1.0 \times FRn \rightarrow FRn$	1111nnnn01001101	1	—
FTRC FRm, FPUL	(int) FRm \rightarrow FPUL	1111mmmm001111101	1	—
MOVT Rn	$T \rightarrow Rn$	0000nnnn00101001	1	—
ROTL Rn	$T \leftarrow Rn \leftarrow$ MSB	0100nnnn00000100	1	MSB
ROTR Rn	LSB $\rightarrow Rn \rightarrow T$	0100nnnn00000101	1	LSB
ROTCL Rn	$T \leftarrow Rn \leftarrow T$	0100nnnn00100100	1	MSB
ROTCR Rn	$T \rightarrow Rn \rightarrow T$	0100nnnn00100101	1	LSB
SHAL Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00100000	1	MSB
SHAR Rn	MSB $\rightarrow Rn \rightarrow T$	0100nnnn00100001	1	LSB
SHLL Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00000000	1	MSB
SHLR Rn	$0 \rightarrow Rn \rightarrow T$	0100nnnn00000001	1	LSB
SHLL2 Rn	$Rn \ll 2 \rightarrow Rn$	0100nnnn00001000	1	—
SHLR2 Rn	$Rn \gg 2 \rightarrow Rn$	0100nnnn00001001	1	—
SHLL8 Rn	$Rn \ll 8 \rightarrow Rn$	0100nnnn00011000	1	—
SHLR8 Rn	$Rn \gg 8 \rightarrow Rn$	0100nnnn00011001	1	—
SHLL16 Rn	$Rn \ll 16 \rightarrow Rn$	0100nnnn00101000	1	—
SHLR16 Rn	$Rn \gg 16 \rightarrow Rn$	0100nnnn00101001	1	—

Table A.4 Source and Destination Operand

Instruction		Operation	Code	Cycles	T Bit
ADD	Rm, Rn	$Rn + Rm \rightarrow Rn$	0011nnnnmmmm1100	1	—
ADDC	Rm, Rn	$Rn + Rm + T \rightarrow Rn$, carry $\rightarrow T$	0011nnnnmmmm1110	1	Carry
ADDV	Rm, Rn	$Rn + Rm \rightarrow Rn$, overflow $\rightarrow T$	0011nnnnmmmm1111	1	Overflow
AND	Rm, Rn	$Rn \& Rm \rightarrow Rn$	0010nnnnmmmm1001	1	—
CMP/EQ	Rm, Rn	When $Rn = Rm$, $1 \rightarrow T$	0011nnnnmmmm0000	1	Comparison result
CMP/HS	Rm, Rn	When unsigned and $Rn \geq Rm$, $1 \rightarrow T$	0011nnnnmmmm0010	1	Comparison result
CMP/GE	Rm, Rn	When signed and $Rn \geq Rm$, $1 \rightarrow T$	0011nnnnmmmm0011	1	Comparison result
CMP/HI	Rm, Rn	When unsigned and $Rn > Rm$, $1 \rightarrow T$	0011nnnnmmmm0110	1	Comparison result
CMP/GT	Rm, Rn	When signed and $Rn > Rm$, $1 \rightarrow T$	0011nnnnmmmm0111	1	Comparison result
CMP/STR	Rm, Rn	When a byte in Rn equals a bytes in Rm, $1 \rightarrow T$	0010nnnnmmmm1100	1	Comparison result
DIV1	Rm, Rn	1 step division ($Rn \div Rm$)	0011nnnnmmmm0100	1	Calculation result
DIV0S	Rm, Rn	MSB of Rn $\rightarrow Q$, MSB of Rm $\rightarrow M$, $M \wedge Q \rightarrow T$	0010nnnnmmmm0111	1	Calculation result
DMULS.L	Rm, Rn	Signed operation of $Rn \times Rm \rightarrow MACH, MACL$	0011nnnnmmmm1101	2 to 4*	—
DMULU.L	Rm, Rn	Unsigned operation of $Rn \times Rm \rightarrow MACH, MACL$	0011nnnnmmmm0101	2 to 4*	—
EXTS.B	Rm, Rn	Sign – extend Rm from byte $\rightarrow Rn$	0110nnnnmmmm1110	1	—
EXTS.W	Rm, Rn	Sign – extend Rm from word $\rightarrow Rn$	0110nnnnmmmm1111	1	—
EXTU.B	Rm, Rn	Zero – extend Rm from byte $\rightarrow Rn$	0110nnnnmmmm1100	1	—
EXTU.W	Rm, Rn	Zero – extend Rm from word $\rightarrow Rn$	0110nnnnmmmm1101	1	—
FADD	FRm, FRn	$FRm + FRn \rightarrow FRn$	1111nnnnmmmm0000	1	—

Instruction	Operation	Code	Cycles	T Bit
FCMP/EQ FRm, FRn	(FRn == FRm)? 1:0 → T	1111nnnnnnmmmm0100	1	Comparison result
FCMP/GT FRm, FRn	(FRn > FRm)? 1:0 → T	1111nnnnnnmmmm0101	1	Comparison result
FDIV FRm, FRn	FRn/FRm → FRn	1111nnnnnnmmmm0011	13	—
FMAC FR0, FRm FRn	(FR0 × FRm) + FRn → FRn	1111nnnnnnmmmm1110	1	—
FMOV FRm, FRn	FRm → FRn	1111nnnnnnmmmm1100	1	—
FMUL FRm, FRn	FRn × FRm → FRn	1111nnnnnnmmmm0010	1	—
FSUB FRm, FRn	FRn – FRm → FRn	1111nnnnnnmmmm0001	1	—
MOV Rm, Rn	Rm → Rn	0110nnnnnnmmmm0011	1	—
MUL.L Rm, Rn	Rn × Rm → MAC	0000nnnnnnmmmm0111	2 to 4*	—
MULS.W Rm, Rn	With sign, Rn × Rm → MAC	0010nnnnnnmmmm1111	1 to 3*	—
MULU.W Rm, Rn	Unsigned, Rn × Rm → MAC	0010nnnnnnmmmm1110	1 to 3*	—
NEG Rm, Rn	0 – Rm → Rn	0110nnnnnnmmmm1011	1	—
NEGC Rm, Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnnnmmmm1010	1	Borrow
NOT Rm, Rn	~Rm → Rn	0110nnnnnnmmmm0111	1	—
OR Rm, Rn	Rn Rm → Rn	0010nnnnnnmmmm1011	1	—
SUB Rm, Rn	Rn – Rm → Rn	0011nnnnnnmmmm1000	1	—
SUBC Rm, Rn	Rn – Rm – T → Rn, Borrow → T	0011nnnnnnmmmm1010	1	Borrow
SUBV Rm, Rn	Rn – Rm → Rn, Underflow → T	0011nnnnnnmmmm1011	1	Underflow
SWAP.B Rm, Rn	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnnnmmmm1000	1	—
SWAP.W Rm, Rn	Rm → Swap upper and lower word → Rn	0110nnnnnnmmmm1001	1	—
TST Rm, Rn	Rn & Rm, when result is 0, 1 → T	0010nnnnnnmmmm1000	1	Test results
XOR Rm, Rn	Rn ^ Rm → Rn	0010nnnnnnmmmm1010	1	—
XTRCT Rm, Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnnnmmmm1101	1	—

Note: * The normal minimum number of execution states.

Table A.5 Load and Store with Control Register or System Register

Instruction		Operation	Code	Cycles	T Bit
FLDS	FRm, FPUL	FRm → FPUL	1111mmmm00011101	1	—
FSTS	FPUL, FRn	FPUL → FRn	1111nnnn00001101	1	—
LDC	Rm, SR	Rm → SR	0100mmmm00001110	1	LSB
LDC	Rm, GBR	Rm → GBR	0100mmmm00011110	1	—
LDC	Rm, VBR	Rm → VBR	0100mmmm00101110	1	—
LDS	Rm, FPSCR	Rm → FPSCR	0100mmmm01101010	1	—
LDS	Rm, FPUL	Rm → FPUL	0100mmmm01011010	1	—
LDS	Rm, MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm, MACL	Rm → MACL	0100mmmm00011010	1	—
LDS	Rm, PR	Rm → PR	0100mmmm00101010	1	—
STC	SR, Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR, Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR, Rn	VBR → Rn	0000nnnn00100010	1	—
STS	FPSCR, Rn	FPSCR → Rn	1111nnnn01101010	1	—
STS	FPUL, Rn	FPUL → Rn	1111nnnn01011010	1	—
STS	MACH, Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL, Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR, Rn	PR → Rn	0000nnnn00101010	1	—

A.1.3 Indirect Register Addressing

Table A.6 Source Operand Only

Instruction	Operation	Code	Cycles	T Bit
JMP @Rm	Delayed branching, Rm → PC	0100nnnn00101011	2	—
JSR @Rm	Delayed branching, PC → PR, Rm → PC	0100nnnn00001011	2	—

Table A.7 Destination Operand Only

Instruction	Operation	Code	Cycles	T Bit
TAS.B @Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	4	Test results

Table A.8 Data Transfer Direct to Register

Instruction	Operation	Code	Cycles	T Bit
FMOV.S FRm, @Rn	FRm → (FRn)	1111nnnnmmmm1010	1	—
FMOV.S @Rm, FRn	(Rm) → FRn	1111nnnnmmmm1000	1	—
MOV.B Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.W Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—
MOV.L Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0010	1	—
MOV.B @Rm, Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—
MOV.W @Rm, Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.L @Rm, Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—

A.1.4 Post-Increment Indirect Register Addressing

Table A.9 Multiply/Accumulate Operation

Instruction	Operation	Code	Cycles	T Bit
MAC.L @Rm+, @Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnnnmmmm1111	3/(2 to 4)*	—
MAC.W @Rm+, @Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnnnmmmm1111	3/(2)*	—

Note: * Normal minimum number of execution states (the number in parenthesis is the number of states when there is contention with preceding/following instructions).

Table A.10 Data Transfer Direct from Register

Instruction	Operation	Code	Cycles	T Bit
FMOV.S @Rm+, FRn	(Rm) → FRn, Rm + 4 → Rm	1111nnnnnnmmmm1001	1	—
MOV.B @Rm+, Rn	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnnnmmmm0100	1	—
MOV.W @Rm+, Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnnnmmmm0101	1	—
MOV.L @Rm+, Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnnnmmmm0110	1	—

Table A.11 Load to Control Register or System Register

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+, SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	3	LSB
LDC.L @Rm+, GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	3	—
LDC.L @Rm+, VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	3	—
LDS.L @Rm+, FPSCR	(Rm) → FPSCR, Rm + 4 → Rm	0100mmmm01100110	1	—
LDS.L @Rm+, FPUL	(Rm) → FPUL, Rm + 4 → Rm	0100mmmm01010110	1	—
LDS.L @Rm+, MACH	(Rm) → MACH, @Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L @Rm+, MACL	(Rm) → MACL, @Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L @Rm+, PR	(Rm) → PR, @Rm + 4 → Rm	0100mmmm00100110	1	—

A.1.5 Pre-Decrement Indirect Register Addressing

Table A.12 Data Transfer Direct from Register

Instruction	Operation	Code	Cycles	T Bit
FMOV.S FRm, @-Rn	Rn - 4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	1	—
MOV.B Rm, @-Rn	Rn - 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
MOV.W Rm, @-Rn	Rn - 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
MOV.L Rm, @-Rn	Rn - 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—

Table A.13 Store from Control Register or System Register

Instruction	Operation	Code	Cycles	T Bit
STC.L SR, @-Rn	Rn - 4 → Rn, SR → (Rn)	0100nnnn00000011	2	—
STC.L GBR, @-Rn	Rn - 4 → Rn, GBR → (Rn)	0100nnnn00010011	2	—
STC.L VBR, @-Rn	Rn - 4 → Rn, VBR → (Rn)	0100nnnn00100011	2	—
STS.L FPSCR, @-Rn	Rn - 4 → Rn, FPSCR → (Rn)	0100nnnn01100010	1	—
STS.L FPUL, @-Rn	Rn - 4 → Rn, FPUL → (Rn)	0100nnnn01010010	1	—
STS.L MACH, @-Rn	Rn - 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L MACL, @-Rn	Rn - 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L PR, @-Rn	Rn - 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—

A.1.6 Indirect Register Addressing with Displacement

Table A.14 Indirect Register Addressing with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOV.B R0, @(disp, Rn)	R0 → (disp + Rn)	10000000nnnndddd	1	—
MOV.W R0, @(disp, Rn)	R0 → (disp + Rn)	10000001nnnndddd	1	—
MOV.L Rm, @(disp, Rn)	Rm → (disp + Rn)	0001nnnnmmmmdddd	1	—
MOV.B @(disp, Rm), R0	(disp + Rm) → sign extension → R0	10000100mmmmdddd	1	—
MOV.W @(disp, Rm), R0	(disp + Rm) → sign extension → R0	10000101mmmmdddd	1	—
MOV.L @(disp, Rm), Rn	(disp + Rm) → Rn	0101nnnnmmmmdddd	1	—

A.1.7 Indirect Indexed Register Addressing

Table A.15 Indirect Indexed Register Addressing

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.L Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
FMOV.S FRm,@(R0,Rn)	FRm → (R0 + Rn)	1111nnnnmmmm0111	1	—
MOV.B @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—
MOV.W @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	1	—
MOV.L @(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—
FMOV.S @(R0,FRm),FRm	(R0 + Rn) → FRn	1111nnnnmmmm0110	1	—

A.1.8 Indirect GBR Addressing with Displacement

Table A.16 Indirect GBR Addressing with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOV.B R0,@(disp,GBR)	R0 → (disp + GBR)	11000000ddddddd	1	—
MOV.W R0,@(disp,GBR)	R0 → (disp + GBR)	11000001ddddddd	1	—
MOV.L R0,@(disp,GBR)	R0 → (disp + GBR)	11000010ddddddd	1	—
MOV.B @(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100ddddddd	1	—
MOV.W @(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000101ddddddd	1	—
MOV.L @(disp,GBR),R0	(disp + GBR) → R0	11000110ddddddd	1	—

A.1.9 Indirect Indexed GBR Addressing

Table A.17 Indirect Indexed GBR Addressing

Instruction	Operation	Code	Cycles	T Bit
AND.B #imm,@(R0,GBR)	$(R0 + GBR) \& imm \rightarrow (R0 + GBR)$	11001101iiiiiii	3	—
OR.B #imm,@(R0,GBR)	$(R0 + GBR) imm \rightarrow (R0 + GBR)$	11001111iiiiiii	3	—
TST.B #imm,@(R0,GBR)	$(R0 + GBR) \& imm$, when result is 0, 1 \rightarrow T	11001100iiiiiii	3	Test results
XOR.B #imm,@(R0,GBR)	$(R0 + GBR) \wedge imm \rightarrow (R0 + GBR)$	11001110iiiiiii	3	—

A.1.10 PC Relative Addressing with Displacement

Table A.18 PC Relative Addressing with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOV.W @(disp,PC),Rn	$(disp + PC) \rightarrow$ sign extension $\rightarrow Rn$	1001nnnnddddddd	1	—
MOV.L @(disp,PC),Rn	$(disp + PC) \rightarrow Rn$	1101nnnnddddddd	1	—
MOVA @(disp,PC),R0	$disp + PC \rightarrow R0$	11000111ddddddd	1	—

A.1.11 PC Relative Addressing

Table A.19 PC Relative Addressing with Rn

Instruction	Operation	Code	Cycles	T Bit
BRAF Rm	Delayed branch, Rm + PC → PC	0000nnnn00100011	2	—
BSRF Rm	Delayed branch, PC → PR, Rm + PC → PC	0000nnnn00000011	2	—

Table A.20 PC Relative Addressing

Instruction	Operation	Code	Cycles	T Bit
BF label	When T = 0, disp + PC → PC; when T = 1, nop	10001011dddddddd	3/1*	—
BF/S label	If T = 0, disp + PC → PC; if T = 1, nop	10001111dddddddd	2/1*	—
BT label	When T = 1, disp + PC → PC; when T = 1, nop	10001001dddddddd	3/1*	—
BT/S label	If T = 1, disp + PC → PC; if T = 0, nop	10001101dddddddd	2/1*	—
BRA label	Delayed branching, disp + PC → PC	1010dddddddddddd	2	—
BSR label	Delayed branching, PC → PR, disp + PC → PC	1011dddddddddddd	2	—

Note: * One state when it does not branch.

A.1.12 Immediate

Table A.21 Load to Register

Instruction	Operation	Code	Cycles	T Bit
FLDI0 FRn	0x00000000 → FRn	1111n11110001101	1	—
FLDI1 FRn	0x3F800000 → FRn	1111n11110011101	1	—

Table A.22 Arithmetic Logical Operations Direct with Register

Instruction	Operation	Code	Cycles	T Bit
ADD #imm,Rn	Rn + imm → Rn	0111n111111111111	1	—
AND #imm,R0	R0 & imm → R0	1100100111111111	1	—
CMP/EQ #imm,R0	When R0 = imm, 1 → T	1000100011111111	1	Comparison result
MOV #imm,Rn	imm → sign extension → Rn	1110n111111111111	1	—
OR #imm,R0	R0 imm → R0	1100101111111111	1	—
TST #imm,R0	R0 & imm, when result is 0, 1 → T	1100100011111111	1	Test results
XOR #imm,R0	R0 ^ imm → R0	1100101011111111	1	—

Table A.23 Specify Exception Processing Vector

Instruction	Operation	Code	Cycles	T Bit
TRAPA #imm	Stack area → PC/SR (imm × 4 + VBR) → PC	1100001111111111	8	—

A.2 Instruction Sets by Instruction Format

Tables A.24 to A.54 list instruction codes and execution cycles by instruction formats.

Table A.24 Instruction Sets by Format

Format	Category	Sample Instruction	Types
0	—	NOF	8
n	Direct register addressing	MOV.T Rn	18
	Direct register addressing (store with control or system registers)	STS MACH, Rn	8
	Indirect register addressing	TAS.B @Rn	1
	Pre-decrement indirect register addressing	STC.L SR, @-Rn	8
	Floating-point instruction	FABS FRn	6
m	Direct register addressing (load with control or system registers)	LDC Rm, SR	8
	PC relative addressing with Rm	BRAF Rm	2
	Indirect register addressing	JMP @Rm	2
	Post-increment indirect register addressing	LDC.L @Rm+, SR	8
	Floating-point instruction	FLDS FRm, FPUL	2
nm	Direct register addressing	ADD Rm, Rn	34
	Indirect register addressing	MOV.L Rm, @Rn	6
	Post-increment indirect register addressing (multiply/accumulate operation)	MAC.W @Rm+, @Rn+	2
	Post-increment indirect register addressing	MOV.L @Rm+, Rn	3
	Pre-decrement indirect register addressing	MOV.L Rm, @-Rn	3
	Indirect indexed register addressing	MOV.L Rm, @(R0, Rn)	6
	Floating-point instruction	FADD FRm, FRn	14
md	Indirect register addressing with displacement	MOV.B @(disp, Rm), R0	2
nd4	Indirect register addressing with displacement	MOV.B R0, @(disp, Rn)	2
nmd	Indirect register addressing with displacement	MOV.L Rm, @(disp, Rn)	2

Format	Category	Sample Instruction	Types
d	Indirect GBR addressing with displacement	MOV.L R0,@(disp,GBR)	6
	Indirect PC addressing with displacement	MOVA @(disp,PC),R0	1
	PC relative addressing	BF disp	4
d12	PC relative addressing	BRA disp	2
nd8	PC relative addressing with displacement	MOV.L @(disp,PC),Rn	2
i	Indirect indexed GBR addressing	AND.B #imm,@(R0,GBR)	4
	Immediate addressing (arithmetic and logical operations direct with register)	AND #imm,R0	5
	Immediate addressing (specify exception processing vector)	TRAPA #imm	1
ni	Immediate addressing (direct register arithmetic operations and data transfers)	ADD #imm,Rn	2
			Total: 172

A.2.1 0 Format

Table A.25 0 Format

Instruction	Operation	Code	Cycles	T Bit
CLRT	0 → T	0000000000001000	1	0
CLRMAC	0 → MACH, MACL	000000000101000	1	—
DIV0U	0 → M/Q/T	000000000011001	1	0
NOP	No operation	000000000001001	1	—
RTE	Delayed branch, Stack area → PC/SR	000000000101011	4	LSB
RTS	Delayed branching, PR → PC	000000000001011	2	—
SETT	1 → T	000000000011000	1	1
SLEEP	Sleep	000000000011011	3*	—

Note: * The number of exception cycles before the chip enters sleep mode.

A.2.2 n Format

Table A.26 Direct Register

Instruction	Operation	Code	Cycles	T Bit
CMP/PL Rn	$Rn > 0, 1 \rightarrow T$	0100nnnn00010101	1	Comparison result
CMP/PZ Rn	$Rn \geq 0, 1 \rightarrow T$	0100nnnn00010001	1	Comparison result
DT Rn	$Rn - 1 \rightarrow Rn$, when Rn is 0, $1 \rightarrow T$. When Rn is nonzero, $0 \rightarrow T$	0100nnnn00010000	1	Comparison result
MOVT Rn	$T \rightarrow Rn$	0000nnnn00101001	1	—
ROTL Rn	$T \leftarrow Rn \leftarrow \text{MSB}$	0100nnnn00000100	1	MSB
ROTR Rn	$\text{LSB} \rightarrow Rn \rightarrow T$	0100nnnn00000101	1	LSB
ROTCL Rn	$T \leftarrow Rn \leftarrow T$	0100nnnn00100100	1	MSB
ROTCR Rn	$T \rightarrow Rn \rightarrow T$	0100nnnn00100101	1	LSB
SHAL Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00100000	1	MSB
SHAR Rn	$\text{MSB} \rightarrow Rn \rightarrow T$	0100nnnn00100001	1	LSB
SHLL Rn	$T \leftarrow Rn \leftarrow 0$	0100nnnn00000000	1	MSB
SHLR Rn	$0 \rightarrow Rn \rightarrow T$	0100nnnn00000001	1	LSB
SHLL2 Rn	$Rn \ll 2 \rightarrow Rn$	0100nnnn00001000	1	—
SHLR2 Rn	$Rn \gg 2 \rightarrow Rn$	0100nnnn00001001	1	—
SHLL8 Rn	$Rn \ll 8 \rightarrow Rn$	0100nnnn00011000	1	—
SHLR8 Rn	$Rn \gg 8 \rightarrow Rn$	0100nnnn00011001	1	—
SHLL16 Rn	$Rn \ll 16 \rightarrow Rn$	0100nnnn00101000	1	—
SHLR16 Rn	$Rn \gg 16 \rightarrow Rn$	0100nnnn00101001	1	—

Table A.27 Direct Register (Store with Control and System Registers)

Instruction		Operation	Code	Cycles	T Bit
STC	SR, Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR, Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR, Rn	VBR → Rn	0000nnnn00100010	1	—
STS	FPSCR, Rn	FPSCR → Rn	0000nnnn01101010	1	—
STS	FPUL, Rn	FPUL → Rn	0000nnnn01011010	1	—
STS	MACH, Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL, Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR, Rn	PR → Rn	0000nnnn00101010	1	—

Table A.28 Indirect Register

Instruction		Operation	Code	Cycles	T Bit
TAS.B	@Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	4	Test results

Table A.29 Indirect Pre-Decrement Register

Instruction		Operation	Code	Cycles	T Bit
STC.L	SR, @-Rn	Rn - 4 → Rn, SR → (Rn)	0100nnnn00000011	1	—
STC.L	GBR, @-Rn	Rn - 4 → Rn, GBR → (Rn)	0100nnnn00010011	1	—
STC.L	VBR, @-Rn	Rn - 4 → Rn, VBR → (Rn)	0100nnnn00100011	1	—
STS.L	FRSCR, @-Rn	Rn - 4 → Rn, FPSCR → Rn	0100nnnn01100010	1	—
STS.L	FPUL, @-Rn	Rn - 4 → Rn, FPUL → Rn	0100nnnn01010010	1	—
STS.L	MACH, @-Rn	Rn - 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—
STS.L	MACL, @-Rn	Rn - 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STS.L	PR, @-Rn	Rn - 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—

Note: SH-3E instructions.

Table A.30 Floating-Point Instruction

Instruction	Operation	Code	Cycles	T Bit
FABS FRn	FRn → FRn	1111nnnn01011101	1	—
FLDI0 FRn	H'00000000 → FRn	1111nnnn10001101	1	—
FLDI1 FRn	H'3F800000 → FRn	1111nnnn10011101	1	—
FLOAT FPUL, FRn	(float)FPUL → FRn	1111nnnn00101101	1	—
FNEG FRn	-FRn → FRn	1111nnnn01001101	1	—
FSTS FPUL, FRn	FPUL → FRn	1111nnnn00001101	1	—

A.2.3 m Format**Table A.31 Direct Register (Load from Control and System Registers)**

Instruction	Operation	Code	Cycles	T Bit
LDC Rm, SR	Rm → SR	0100mmmm00001110	1	LSB
LDC Rm, GBR	Rm → GBR	0100mmmm00011110	1	—
LDC Rm, VBR	Rm → VBR	0100mmmm00101110	1	—
LDS Rm, FPSCR	Rm → FPSCR	0100nnnn01101010	1	—
LDS Rm, FPUL	Rm → FPUL	0100nnnn01011010	1	—
LDS Rm, MACH	Rm → MACH	0100mmmm00001010	1	—
LDS Rm, MACL	Rm → MACL	0100mmmm00011010	1	—
LDS Rm, PR	Rm → PR	0100mmmm00101010	1	—

Table A.32 Indirect Register

Instruction	Operation	Code	Cycles	T Bit
JMP @Rm	Delayed branch, Rm → PC	0100mmmm00101011	2	—
JSR @Rm	Delayed branch, PC → PR, Rm → PC	0100mmmm00001011	2	—

Table A.33 Indirect Post-Increment Register

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+, SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	3	LSB
LDC.L @Rm+, GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	3	—
LDC.L @Rm+, VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	3	—
LDS.L @Rm+, FPSCR	@Rm → FPSCR, Rm + 4 → Rm	0100nnnn01100110	1	—
LDS.L @Rm+, FPUL	@Rm → FPUL, Rm + 4 → Rm	0100nnnn01010110	1	—
LDS.L @Rm+, MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDS.L @Rm+, MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDS.L @Rm+, PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—

Table A.34 PC Relative Addressing with Rn

Instruction	Operation	Code	Cycles	T Bit
BRAF Rn	Delayed branch, Rn + PC → PC	0000nnnn00100011	2	—
BSRF Rn	Delayed branch, PC → PR, Rn + PC → PC	0000nnnn00000011	2	—

Table A.35 Floating-Point Instructions

Instruction	Operation	Code	Cycles	T Bit
FLDS FRm, FPUL	FRm → FPUL	1111nnnn00011101	1	—
FTRC FRm, FPUL	(long)FRm → FPUL	1111nnnn00111101	1	—

A.2.4 nm Format

Table A.36 Direct Register

Instruction	Operation	Code	Cycles	T Bit
ADD Rm, Rn	$Rm + Rn \rightarrow Rn$	0011nnnnmmmm1100	1	—
ADDC Rm, Rn	$Rn + Rm + T \rightarrow Rn$, carry $\rightarrow T$	0011nnnnmmmm1110	1	Carry
ADDV Rm, Rn	$Rn + Rm \rightarrow Rn$, overflow $\rightarrow T$	0011nnnnmmmm1111	1	Overflow
AND Rm, Rn	$Rn \& Rm \rightarrow Rn$	0010nnnnmmmm1001	1	—
CMP/EQ Rm, Rn	When $Rn = Rm$, $1 \rightarrow T$	0011nnnnmmmm0000	1	Comparison result
CMP/HS Rm, Rn	When unsigned and $Rn \geq Rm$, $1 \rightarrow T$	0011nnnnmmmm0010	1	Comparison result
CMP/GE Rm, Rn	When signed and $Rn \geq Rm$, $1 \rightarrow T$	0011nnnnmmmm0011	1	Comparison result
CMP/HI Rm, Rn	When unsigned and $Rn > Rm$, $1 \rightarrow T$	0011nnnnmmmm0110	1	Comparison result
CMP/GT Rm, Rn	When signed and $Rn > Rm$, $1 \rightarrow T$	0011nnnnmmmm0111	1	Comparison result
CMP/STR Rm, Rn	When a byte in Rn equals a byte in Rm , $1 \rightarrow T$	0010nnnnmmmm1100	1	Comparison result
DIV1 Rm, Rn	1 step division ($Rn \div Rm$)	0011nnnnmmmm0100	1	Calculation result
DIV0S Rm, Rn	MSB of $Rn \rightarrow Q$, MSB of $Rm \rightarrow M$, $M \wedge Q \rightarrow T$	0010nnnnmmmm0111	1	Calculation result
DMULS.L Rm, Rn	Signed operation of $Rn \times Rm \rightarrow MACH, MACL$	0011nnnnmmmm1101	2 to 4*	—
DMULU.L Rm, Rn	Unsigned operation of $Rn \times Rm \rightarrow MACH, MACL$	0011nnnnmmmm0101	2 to 4*	—
EXTS.B Rm, Rn	Sign-extend Rm from byte $\rightarrow Rn$	0110nnnnmmmm1110	1	—
EXTS.W Rm, Rn	Sign-extend Rm from word $\rightarrow Rn$	0110nnnnmmmm1111	1	—
EXTU.B Rm, Rn	Zero-extend Rm from byte $\rightarrow Rn$	0110nnnnmmmm1100	1	—

Instruction	Operation	Code	Cycles	T Bit
EXTU.W Rm, Rn	Zero-extend Rm from word → Rn	0110nnnnnnmmmm1101	1	—
MOV Rm, Rn	Rm → Rn	0110nnnnnnmmmm0011	1	—
MUL.L Rm, Rn	Rn × Rm → MAC	0000nnnnnnmmmm0111	2 to 4*	—
MULS.W Rm, Rn	With sign, Rn × Rm → MAC	0010nnnnnnmmmm1111	1 to 3*	—
MULU.W Rm, Rn	Unsigned, Rn × Rm → MAC	0010nnnnnnmmmm1110	1 to 3*	—
NEG Rm, Rn	0 – Rm → Rn	0110nnnnnnmmmm1011	1	—
NEGC Rm, Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnnnmmmm1010	1	Borrow
NOT Rm, Rn	~Rm → Rn	0110nnnnnnmmmm0111	1	—
OR Rm, Rn	Rn Rm → Rn	0010nnnnnnmmmm1011	1	—
SUB Rm, Rn	Rn – Rm → Rn	0011nnnnnnmmmm1000	1	—
SUBC Rm, Rn	Rn – Rm – T → Rn, Borrow → T	0011nnnnnnmmmm1010	1	Borrow
SUBV Rm, Rn	Rn – Rm → Rn, Underflow → T	0011nnnnnnmmmm1011	1	Under-flow
SWAP.B Rm, Rn	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnnnmmmm1000	1	—
SWAP.W Rm, Rn	Rm → Swap upper and lower word → Rn	0110nnnnnnmmmm1001	1	—
TST Rm, Rn	Rn & Rm, when result is 0, 1 → T	0010nnnnnnmmmm1000	1	Test results
XOR Rm, Rn	Rn ^ Rm → Rn	0010nnnnnnmmmm1010	1	—
XTRCT Rm, Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnnnmmmm1101	1	—

Note: The normal minimum number of execution states.

Table A.37 Indirect Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.W Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—
MOV.L Rm, @Rn	Rm → (Rn)	0010nnnnmmmm0010	1	—
MOV.B @Rm, Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—
MOV.W @Rm, Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.L @Rm, Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—

Table A.38 Indirect Post-Increment Register (Multiply/Accumulate Operation)

Instruction	Operation	Code	Cycles	T Bit
MAC.L @Rm+, @Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	3/(2 to 4)*	—
MAC.W @Rm+, @Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	3/(2)*	—

Note: * Normal minimum number of execution states (the number in parentheses is the number of states when there is contention with preceding/following instructions).

Table A.39 Indirect Post-Increment Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B @Rm+, Rn	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
MOV.W @Rm+, Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
MOV.L @Rm+, Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—

Table A.40 Indirect Pre-Decrement Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm, @-Rn	Rn - 1 → Rn, Rm → (Rn)	0010nnnnmmmm0100	1	—
MOV.W Rm, @-Rn	Rn - 2 → Rn, Rm → (Rn)	0010nnnnmmmm0101	1	—
MOV.L Rm, @-Rn	Rn - 4 → Rn, Rm → (Rn)	0010nnnnmmmm0110	1	—

Table A.41 Indirect Indexed Register

Instruction	Operation	Code	Cycles	T Bit
MOV.B Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.L Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
MOV.B @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—
MOV.W @(R0,Rm),Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1101	1	—
MOV.L @(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—

Table A.42 Floating Point Instructions

Instruction	Operation	Code	Cycles	T Bit
FADD FRm,FRn	FRn + FRm → FRn	1111nnnnmmmm0000	1	—
FCMP/EQ FRm,FRn	(FRn = FRm)? 1:0 → T	1111nnnnmmmm0100	1	Comparison result
FCMP/GT FRm,FRn	(FRn > FRm)? 1:0 → T	1111nnnnmmmm0101	1	Comparison result
FDIV FRm,FRn	FRn/FRm → FRn	1111nnnnmmmm0011	13	—
FMAC FR0,FRm,FRn	FR0 × FRm + FRn → FRn	1111nnnnmmmm1110	1	—
FMOV FRm,FRn	FRm → FRn	1111nnnnmmmm1100	1	—
FMOV.S @(R0,Rm),FRn	(R0 + Rm) → FRn	1111nnnnmmmm0110	1	—
FMOV.S @Rm+,FRn	(Rm) → FRn, Rm + 4 → Rm	1111nnnnmmmm1001	1	—
FMOV.S @Rm,FRn	(Rm) → FRn	1111nnnnmmmm1000	1	—
FMOV.S FRm,@(R0,Rn)	FRm → (R0 + Rn)	1111nnnnmmmm0111	1	—
FMOV.S FRm,@-Rn	Rn-4 → Rn, FRm → (Rn)	1111nnnnmmmm1011	1	—
FMOV.S FRm,@Rn	FRm → (Rn)	1111nnnnmmmm1010	1	—
FMUL FRm,FRn	FRn × FRm → FRn	1111nnnnmmmm0010	1	—
FSUB FRm,FRn	FRn - FRm → FRn	1111nnnnmmmm0001	1	—

A.2.5 md Format

Table A.43 md Format

Instruction	Operation	Code	Cycles	T Bit
MOV.B @(disp,Rm),R0	(disp + Rm) → sign extension → R0	10000100mmmmddddd	1	—
MOV.W @(disp,Rm),R0	(disp × 2 + Rm) → sign extension → R0	10000101mmmmddddd	1	—

A.2.6 nd4 Format

Table A.44 nd4 Format

Instruction	Operation	Code	Cycles	T Bit
MOV.B R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnnddddd	1	—
MOV.W R0,@(disp,Rn)	R0 → (disp × 2 + Rn)	10000001nnnnddddd	1	—

A.2.7 nmd Format

Table A.45 nmd Format

Instruction	Operation	Code	Cycles	T Bit
MOV.L Rm,@(disp,Rn)	Rm → (disp + Rn)	0001nnnnmmmmddddd	1	—
MOV.L @(disp,Rm),Rn	(disp × 4 + Rm) → Rn	0101nnnnmmmmddddd	1	—

A.2.8 d Format

Table A.46 Indirect GBR with Displacement

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

Table A.47 PC Relative with Displacement

Instruction	Operation	Code	Cycles	T Bit
MOVA @(disp,PC),R0	disp × 4 + PC → R0	11000111dddddddd	1	—

Table A.48 PC Relative

Instruction	Operation	Code	Cycles	T Bit
BF label	When T = 0, disp × 2 + PC → PC; when T = 1, nop	10001011dddddddd	3/1*	—
BF/S label	If T = 0, disp × 2 + PC → PC; if T = 1, nop	10001111dddddddd	2/1*	—
BT label	When T = 1, disp × 2 + PC → PC; when T = 0, nop	10001001dddddddd	3/1*	—
BT/S label	If T = 1, disp × 2 + PC → PC; if T = 0, nop	10001101dddddddd	2/1*	—

Note: * One state when it does not branch.

A.2.9 d12 Format

Table A.49 d12 Format

Instruction	Operation	Code	Cycles	T Bit
BRA label	Delayed branching, $\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$	1010dddddddddddd	2	—
BSR label	Delayed branching, $\text{PC} \rightarrow \text{PR}$, $\text{disp} \times 2 + \text{PC} \rightarrow \text{PC}$	1011dddddddddddd	2	—

A.2.10 nd8 Format

Table A.50 nd8 Format

Instruction	Operation	Code	Cycles	T Bit
MOV.W @(disp,PC),Rn	$(\text{disp} \times 2 + \text{PC}) \rightarrow \text{sign extension} \rightarrow \text{Rn}$	1001nnnndddddddd	1	—
MOV.L @(disp,PC),Rn	$(\text{disp} \times 4 + \text{PC}) \rightarrow \text{Rn}$	1101nnnndddddddd	1	—

A.2.11 i Format

Table A.51 Indirect Indexed GBR

Instruction	Operation	Code	Cycles	T Bit
AND.B #imm,@(R0,GBR)	$(\text{R0} + \text{GBR}) \& \text{imm} \rightarrow (\text{R0} + \text{GBR})$	11001101iiiiiiii	3	—
OR.B #imm,@(R0,GBR)	$(\text{R0} + \text{GBR}) \text{imm} \rightarrow (\text{R0} + \text{GBR})$	11001111iiiiiiii	3	—
TST.B #imm,@(R0,GBR)	$(\text{R0} + \text{GBR}) \& \text{imm}$, when result is 0, 1 \rightarrow T	11001100iiiiiiii	3	Test results
XOR.B #imm,@(R0,GBR)	$(\text{R0} + \text{GBR}) \wedge \text{imm} \rightarrow (\text{R0} + \text{GBR})$	11001110iiiiiiii	3	—

Table A.52 Immediate (Arithmetic Logical Operation with Direct Register)

Instruction	Operation	Code	Cycles	T Bit
AND #imm,R0	$R0 \& imm \rightarrow R0$	11001001iiiiiiii	1	—
CMP/EQ #imm,R0	When $R0 = imm$, $1 \rightarrow T$	10001000iiiiiiii	1	Comparison results
OR #imm,R0	$R0 imm \rightarrow R0$	11001011iiiiiiii	1	—
TST #imm,R0	$R0 \& imm$, when result is 0, $1 \rightarrow T$	11001000iiiiiiii	1	Test results
XOR #imm,R0	$R0 \wedge imm \rightarrow R0$	11001010iiiiiiii	1	—

Table A.53 Immediate (Specify Exception Processing Vector)

Instruction	Operation	Code	Cycles	T Bit
TRAPA #imm	Stack area \rightarrow PC/SR ($imm \times 4 + VBR$) \rightarrow PC	11000011iiiiiiii	8	—

A.2.12 ni Format

Table A.54 ni Format

Instruction	Operation	Code	Cycles	T Bit
ADD #imm,Rn	$Rn + imm \rightarrow Rn$	0111nnnniiiiiiii	1	—
MOV #imm,Rn	$imm \rightarrow$ sign extension $\rightarrow Rn$	1110nnnniiiiiiii	1	—

A.3 Instruction Set by Instruction Code

Table A.55 lists instruction codes and execution cycles by instruction code.

Table A.55 Instruction Set by Instruction Code

Instruction		Operation	Code	Cycles	T Bit
CLRT		0 → T	0000000000001000	1	0
NOP		No operation	0000000000001001	1	—
RTS		Delayed branching, PR → PC	0000000000001011	2	—
SETT		1 → T	0000000000011000	1	1
DIVOU		0 → M/Q/T	0000000000011001	1	0
SLEEP		Sleep	0000000000011011	3	—
CLRMACH		0 → MACH, MACL	000000000101000	1	—
RTE		Delayed branch, SSR/SPC → SR/PC	000000000101011	4	—
STC	SR, Rn	SR → Rn	0000nnnn00000010	1	—
BSRF	Rn	Delayed branch, PC → PR, Rn + PC → PC	0000nnnn00000011	2	—
STS	MACH, Rn	MACH → Rn	0000nnnn00001010	1	—
STC	GBR, Rn	GBR → Rn	0000nnnn00010010	1	—
STS	MACL, Rn	MACL → Rn	0000nnnn00011010	1	—
STC	VBR, Rn	VBR → Rn	0000nnnn00100010	1	—
BRAF	Rm	Delayed branch, Rn + PC → PC	0000nnnn00100011	2	—
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
STS	PR, Rn	PR → Rn	0000nnnn00101010	1	—
STS	FPUL, Rn	FPUL → Rn	0000nnnn01011010	1	—
STS	FPSCR, Rn	FPSCR → Rn	0000nnnn01101010	1	—
MOV.B	Rm, @(R0, Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W	Rm, @(R0, Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—
MOV.L	Rm, @(R0, Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0110	1	—
MUL.L	Rm, Rn	Rn × Rm → MACL	0000nnnnmmmm0111	2 to 4*	—
MOV.B	@(R0, Rm), Rn	(R0 + Rm) → sign extension → Rn	0000nnnnmmmm1100	1	—

Instruction	Operation	Code	Cycles	T Bit
MOV.W @ (R0, Rm), Rn	$(R0 + Rm) \rightarrow$ sign extension $\rightarrow Rn$	0000nnnnmmmm1101	1	—
MOV.L @ (R0, Rm), Rn	$(R0 + Rm) \rightarrow Rn$	0000nnnnmmmm1110	1	—
MAC.L @Rm+, @Rn+	Signed operation of $(Rn) \times (Rm) + MAC \rightarrow MAC$	0000nnnnmmmm1111	3/(2 to 4)*	—
MOV.L Rm, @(disp, Rn)	$Rm \rightarrow (disp \times 4 + Rn)$	0001nnnnmmmmdddd	1	—
MOV.B Rm, @Rn	$Rm \rightarrow (Rn)$	0010nnnnmmmm0000	1	—
MOV.W Rm, @Rn	$Rm \rightarrow (Rn)$	0010nnnnmmmm0001	1	—
MOV.L Rm, @Rn	$Rm \rightarrow (Rn)$	0010nnnnmmmm0010	1	—
MOV.B Rm, @-Rn	$Rn - 1 \rightarrow Rn, Rm \rightarrow (Rn)$	0010nnnnmmmm0100	1	—
MOV.W Rm, @-Rn	$Rn - 2 \rightarrow Rn, Rm \rightarrow (Rn)$	0010nnnnmmmm0101	1	—
MOV.L Rm, @-Rn	$Rn - 4 \rightarrow Rn, Rm \rightarrow (Rn)$	0010nnnnmmmm0110	1	—
DIV0S Rm, Rn	MSB of Rn $\rightarrow Q$, MSB of Rm $\rightarrow M, M \wedge Q \rightarrow T$	0010nnnnmmmm0111	1	Calculation result
TST Rm, Rn	$Rn \& Rm$, when result is 0, 1 $\rightarrow T$	0010nnnnmmmm1000	1	Test results
AND Rm, Rn	$Rn \& Rm \rightarrow Rn$	0010nnnnmmmm1001	1	—
XOR Rm, Rn	$Rn \wedge Rm \rightarrow Rn$	0010nnnnmmmm1010	1	—
OR Rm, Rn	$Rn Rm \rightarrow Rn$	0010nnnnmmmm1011	1	—
CMP/STR Rm, Rn	When a byte in Rn equals a byte in Rm, 1 $\rightarrow T$	0010nnnnmmmm1100	1	Comparison result
XTRCT Rm, Rn	Rm: Center 32 bits of Rn $\rightarrow Rn$	0010nnnnmmmm1101	1	—
MULU.W Rm, Rn	Unsigned, $Rn \times Rm \rightarrow MAC$	0010nnnnmmmm1110	1 to 3*	—
MULS.W Rm, Rn	Signed, $Rn \times Rm \rightarrow MAC$	0010nnnnmmmm1111	1 to 3*	—

Instruction	Operation	Code	Cycles	T Bit
CMP/EQ Rm, Rn	When Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/HS Rm, Rn	When unsigned and Rn ≥ Rm, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/GE Rm, Rn	When signed and Rn ≥ Rm, 1 → T	0011nnnnmmmm0011	1	Comparison result
DIV1 Rm, Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	1	Calculation result
DMULU.L Rm, Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 to 4*	—
CMP/HI Rm, Rn	When unsigned and Rn > Rm, 1 → T	0011nnnnmmmm0110	1	Comparison result
CMP/GT Rm, Rn	When signed and Rn > Rm, 1 → T	0011nnnnmmmm0111	1	Comparison result
SUB Rm, Rn	Rn − Rm → Rn	0011nnnnmmmm1000	1	—
SUBC Rm, Rn	Rn − Rm − T → Rn, Borrow → T	0011nnnnmmmm1010	1	Borrow
SUBV Rm, Rn	Rn − Rm → Rn, underflow → T	0011nnnnmmmm1011	1	Underflow
ADD Rm, Rn	Rm + Rn → Rn	0011nnnnmmmm1100	1	—
DMULS.L Rm, Rn	Signed operation of Rn × Rm → MACH, MACL	0011nnnnmmmm1101	2 to 4*	—
ADDC Rm, Rn	Rn + Rm + T → Rn, carry → T	0011nnnnmmmm1110	1	Carry
ADDV Rm, Rn	Rn + Rm → Rn, overflow → T	0011nnnnmmmm1111	1	Overflow
SHLL Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLR Rn	0 → Rn → T	0100nnnn00000001	1	LSB
STS.L MACH, @-Rn	Rn − 4 → Rn, MACH → (Rn)	0100nnnn00000010	1	—

Instruction		Operation	Code	Cycles	T Bit
STC.L	SR, @-Rn	Rn - 4 → Rn, SR → (Rn)	0100nnnn00000011	2	—
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
LDS.L	@Rm+, MACH	(Rm) → MACH, Rm + 4 → Rm	0100mmmm00000110	1	—
LDC.L	@Rm+, SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	3	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
LDS	Rm, MACH	Rm → MACH	0100mmmm00001010	1	—
JSR	@Rm	Delayed branching, PC → Rn, Rn → PC	0100nnnn00001011	2	—
LDC	Rm, SR	Rm → SR	0100mmmm00001110	1	LSB
DT	Rn	Rn - 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Com- parison result
CMP/PZ	Rn	Rn ≥ 0, 1 → T	0100nnnn00010001	1	Com- parison result
STS.L	MACL, @-Rn	Rn - 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STC.L	GBR, @-Rn	Rn - 4 → Rn, GBR → (Rn)	0100nnnn00010011	2	—
CMP/PL	Rn	Rn > 0, 1 → T	0100nnnn00010101	1	Com- parison result
LDS.L	@Rm+, MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDC.L	@Rm+, GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	3	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
LDS	Rm, MACL	Rm → MACL	0100mmmm00011010	1	—
TAS.B	@Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	4	Test results

Instruction		Operation	Code	Cycles	T Bit
LDC	Rm, GBR	Rm → GBR	0100mmmm00011110	1	—
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
STS.L	PR, @-Rn	Rn - 4 → Rn, PR → (Rn)	0100nnnn00100010	1	—
STC.L	VBR, @-Rn	Rn - 4 → Rn, VBR → (Rn)	0100nnnn00100011	2	—
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
LDS.L	@Rm+, PR	(Rm) → PR, Rm + 4 → Rm	0100mmmm00100110	1	—
LDC.L	@Rm+, VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	3	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—
LDS	Rm, PR	Rm → PR	0100mmmm00101010	1	—
JMP	@Rm	Delayed branching, Rm → PC	0100nnnn00101011	2	—
LDC	Rm, VBR	Rm → VBR	0100mmmm00101110	1	—
STS.L	FPUL, @-Rn	Rn-4 → Rn, FPUL → (Rn)	0100nnnn01010010	1	—
LDS.L	@Rm+, FPUL	(Rm) → FPUL, Rm+4 → Rm	0100nnnn01010110	1	—
LDS	Rm, FPUL	Rm → FPUL	0100mmmm01011010	1	—
STS.L	FPSCR, @-Rn	Rn-4 → Rn, FPSCR → (Rn)	0100nnnn01100010	1	—
LDS.L	@Rm, FPSCR	(Rm) → FPSCR, Rm+4 → Rm	0100mmmm01100110	1	—
LDS	Rm, FPSCR	Rm → FPSCR	0100nnnn01101010	1	—
MAC.W	@Rm+, @Rn+	With sign, (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	3/(2)*	—
MOV.L	@(disp, Rm), Rn	(disp + Rm) → Rn	0101nnnnmmmmdddd	1	—
MOV.B	@Rm, Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0000	1	—

Instruction	Operation	Code	Cycles	T Bit
MOV.W @Rm, Rn	(Rm) → sign extension → Rn	0110nnnnmmmm0001	1	—
MOV.L @Rm, Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—
MOV Rm, Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MOV.B @Rm+, Rn	(Rm) → sign extension → Rn, Rm + 1 → Rm	0110nnnnmmmm0100	1	—
MOV.W @Rm+, Rn	(Rm) → sign extension → Rn, Rm + 2 → Rm	0110nnnnmmmm0101	1	—
MOV.L @Rm+, Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—
NOT Rm, Rn	~Rm → Rn	0110nnnnmmmm0111	1	—
SWAP.B Rm, Rn	Rm → Swap upper and lower halves of lower 2 bytes → Rn	0110nnnnmmmm1000	1	—
SWAP.W Rm, Rn	Rm → Swap upper and lower word → Rn	0110nnnnmmmm1001	1	—
NEGC Rm, Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
NEG Rm, Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—
EXTU.B Rm, Rn	Zero-extend Rm from byte → Rn	0110nnnnmmmm1100	1	—
EXTU.W Rm, Rn	Zero-extend Rm from word → Rn	0110nnnnmmmm1101	1	—
EXTS.B Rm, Rn	Sign-extend Rm from byte → Rn	0110nnnnmmmm1110	1	—
EXTS.W Rm, Rn	Sign-extend Rm from word → Rn	0110nnnnmmmm1111	1	—
ADD #imm, Rn	Rn + #imm → Rn	0111nnnniiiiiii	1	—
MOV.B R0, @(disp, Rn)	R0 → (disp + Rn)	10000000nnnndddd	1	—
MOV.W R0, @(disp, Rn)	R0 → (disp + Rn)	10000001nnnndddd	1	—
MOV.B @(disp, Rm), R0	(disp + Rm) → sign extension → R0	10000100mmmmddd	1	—
MOV.W @(disp, Rm), R0	(disp + Rm) → sign extension → R0	10000101mmmmddd	1	—

Instruction	Operation	Code	Cycles	T Bit
CMP/EQ #imm,R0	When R0 = imm, 1 → T	10001000iiiiiii	1	Comparison result
BT label	When T = 1, disp + PC → PC; when T = 1, nop.	10001001ddddddd	3/1*2	—
BF label	When T = 0, disp + PC → PC; when T = 1, nop	10001011ddddddd	3/1*2	—
BT/S label	If T = 1, disp + PC → PC; if T = 0, nop	10001101ddddddd	2/1*2	—
BF/S label	If T = 0, disp + PC → PC; if T = 1, nop	10001111ddddddd	2/1*2	—
MOV.W @(disp,PC),Rn	(disp + PC) → sign extension → Rn	1001nnnnddddddd	1	—
BRA label	Delayed branching, disp + PC → PC	1010ddddddddddd	2	—
BSR label	Delayed branching, PC → PR, disp + PC → PC	1011ddddddddddd	2	—
MOV.B R0,@(disp,GBR)	R0 → (disp + GBR)	11000000ddddddd	1	—
MOV.W R0,@(disp,GBR)	R0 → (disp × 2 + GBR)	11000001ddddddd	1	—
MOV.L R0,@(disp,GBR)	R0 → (disp × 4 + GBR)	11000010ddddddd	1	—
TRAPA #imm	Stack area → PC/SR (imm × 4 + VBR) → PC	11000011iiiiiii	8	—
MOV.B @(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100ddddddd	1	—
MOV.W @(disp,GBR),R0	(disp × 2 + GBR) → sign extension → R0	11000101ddddddd	1	—
MOV.L @(disp,GBR),R0	(disp × 4 + GBR) → R0	11000110ddddddd	1	—
MOVA @(disp,PC),R0	disp × 4 + PC → R0	11000111ddddddd	1	—
TST #imm,R0	R0 & imm, when result is 0, 1 → T	11001000iiiiiii	1	Test results
AND #imm,R0	R0 & imm → R0	11001001iiiiiii	1	—
XOR #imm,R0	R0 ^ imm → R0	11001010iiiiiii	1	—
OR #imm,R0	R0 imm → R0	11001011iiiiiii	1	—

Instruction	Operation	Code	Cycles	T Bit
TST.B	#imm,@(R0,GBR) (R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiii	3	Test results
AND.B	#imm,@(R0,GBR) (R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiii	3	—
XOR.B	#imm,@(R0,GBR) (R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiii	3	—
OR.B	#imm,@(R0,GBR) (R0 + GBR) imm → (R0 + GBR)	11001111iiiiiii	3	—
MOV.L	@(disp,PC),Rn (disp × 4 + PC) → Rn	1101nnnnddddddd	1	—
MOV	#imm,Rn #imm → sign extension → Rn	1110nnnniiiiiii	1	—
FSTS	FPUL,FRn FPUL → FRn	1111nnnn00001101	1	—
FLDS	FRm,FPUL FRm → FPUL	1111nnnn00011101	1	—
FLOAT	FPUL,FRn (float) FPUL → FRn	1111nnnn00101101	1	—
FTRC	FRm,FPUL (long) FRm → FPUL	1111nnnn00111101	1	—
FNEG	FRn -FRn → FRn	1111nnnn01001101	1	—
FABS	FRn FRn → FRn	1111nnnn01011101	1	—
FLDI0	FRn H'00000000 → FRn	1111nnnn10001101	1	—
FLDI1	FRn H'3F800000 → FRn	1111nnnn10011101	1	—
FADD	FRm,FRn FRn + FRm → FRn	1111nnnnmmmm0000	1	—
FSUB	FRm,FRn FRn - FRm → FRn	1111nnnnmmmm0001	1	—
FMUL	FRm,FRn FRn × FRm → FRn	1111nnnnmmmm0010	1	—
FDIV	FRm,FRn FRn/FRm → FRn	1111nnnnmmmm0011	13	—
FCMP/EQ	FRm,FRn (FRn = FRm)?1:0 → T	1111nnnnmmmm0100	1	Comparison result
FCMP/GT	FRm,FRn (FRn > FRm)?1:0 → T	1111nnnnmmmm0101	1	Comparison result
FMOV.S	@(R0,Rm),FRn (R0 + Rm) → FRn	1111nnnnmmmm0110	1	—
FMOV.S	FRm,@(R0,Rn) (FRm) → (R0 + Rn)	1111nnnnmmmm0111	1	—
FMOV.S	@Rm,FRn (Rm) → FRn	1111nnnnmmmm1000	1	—
FMOV.S	@Rm+,FRn (Rm) → FRn, Rm + 4 → Rm	1111nnnnmmmm1001	1	—

Instruction		Operation	Code	Cycles	T Bit
FMOV.S	FRm, @Rn	FRm → (Rn)	1111nnnnnnmmmm1010	1	—
FMOV.S	FRm, @-Rn	Rn - 4 → Rn, FRm → (Rn)	1111nnnnnnmmmm1011	1	—
FMOV	FRm, FRn	FRm → FRn	1111nnnnnnmmmm1100	1	—
FMAC	FR0, FRm, FRn	FR0 × FRm + FRn → FRn	1111nnnnnnmmmm1110	1	—

- Notes:
1. Normal minimum number of execution states (the number in parenthesis is the number of states when there is contention with preceding/following instructions).
 2. One state when it does not branch.

A.4 Operation Code Map

Table A.56 shows operation code map.

Table A.56 Operation Code Map

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011-1111
MSB		LSB		MD: 00	MD: 01	MD: 10	MD: 11
0000	Rn	Fx	0000				
0000	Rn	Fx	0001				
0000	Rn	Fx	0010	STC SR, Rn	STC GBR, Rn	STC VBR, Rn	
0000	Rn	Fx	0011	BSRF Rm		BRAF Rm	
0000	Rn	Rm	01MD	MOV.B Rm, @(R0, Rn)	MOV.W Rm, @(R0, Rn)	MOV.L Rm, @(R0, Rn)	MUL.L Rm, Rn
0000	0000	Fx	1000	CLRT	SETT	CLRMAC	
0000	0000	Fx	1001	NOP	DIV0U		
0000	0000	Fx	1010				
0000	0000	Fx	1011	RTS	SLEEP	RTE	
0000	Rn	Fx	1000				
0000	Rn	Fx	1001			MOVT Rn	
0000	Rn	Fx	1010	STS MACH, Rn	STS MACL, Rn	STS PR, Rn	STS FPUL, Rn/ STS FPSCR, Rn
0000	Rn	Fx	1011				
0000	Rn	RM	11MD	MOV.B @(R0, Rm), Rn	MOV.W @(R0, Rm), Rn	MOV.L @(R0, Rm), Rn	MAC.L @Rm+, @Rn+
0001	Rn	Rm	disp	MOV.L Rm, @(disp:4, Rn)			
0010	Rn	Rm	00MD	MOV.B Rm, @Rn	MOV.W Rm, @Rn	MOV.L Rm, @Rn	
0010	Rn	Rm	01MD	MOV.B Rm, @-Rn	MOV.W Rm, @-Rn	MOV.L Rm, @-Rn	DIV0S Rm, Rn
0010	Rn	Rm	10MD	TST Rm, Rn	AND Rm, Rn	XOR Rm, Rn	OR Rm, Rn
0010	Rn	Rm	11MD	CMP/STR Rm, Rn	XTRCT Rm, Rn	MULU.W Rm, Rn	MULS.W Rm, Rn
0011	Rn	Rm	00MD	CMP/EQ Rm, Rn		CMP/HS Rm, Rn	CMP/GE Rm, Rn
0011	Rn	Rm	01MD	DIV1 Rm, Rn	DMULU.L Rm, Rn	CMP/HI Rm, Rn	CMP/GT Rm, Rn
0011	Rn	Rm	10MD	SUB Rm, Rn		SUBC Rm, Rn	SUBV Rm, Rn
0011	Rn	Rm	11MD	ADD Rm, Rn	DMULS.L Rm, Rn	ADDC Rm, Rn	ADDV Rm, Rn
0100	Rn	Fx	0000	SHLL Rn	DT Rn	SHAL Rn	
0100	Rn	Fx	0001	SHLR Rn	CMP/PZ Rn	SHAR Rn	

Appendix A Instruction Code

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011-1111
MSB		LSB		MD: 00	MD: 01	MD: 10	MD: 11
0100	Rn	Fx	0010	STC.L MACH, @-Rn	STC.L MACL, @-Rn	STC.L PR, @-Rn	STC.L FPSCR, @-Rn STC.L FPUL, @-Rn
0100	Rn	00MD	0011	STC.L SR, @-Rn	STC.L GBR, @-Rn	STC.L VBR, @-Rn	
0100	Rn	Fx	0100	ROTL Rn		ROTCL Rn	
0100	Rn	Fx	0101	ROTR Rn	CMP/PL Rn	ROTCR Rn	
0100	Rm	Fx	0110	LDS.L @Rm+, MACH	LDS.L @Rm+, MACL	LDS.L @Rm+, PR	LDS.L @Rm+, FPSCR LDS.L @Rm+, FPUL
0100	Rm	Fx	0111	LDC.L @Rm+, SR	LDC.L @Rm+, GBR	LDC.L @Rm+, VBR	
0100	Rn	Fx	1000	SHLL2 Rn	SHLL8 Rn	SHLL16 Rn	
0100	Rn	Fx	1001	SHLR2 Rn	SHLR8 Rn	SHLR16 Rn	
0100	Rm	Fx	1010	LDS Rm, MACH	LDS Rm, MACL	LDS Rm, PR	LDS Rm, FPSCR LDS Rm, FPUL
0100	Rm/ Rn	Fx	1011	JSR @Rm	TAS.B @Rm	JMP @Rm	
0100	Rm	Fx	1100				
0100	Rm	Fx	1101				
0100	Rm	Fx	1110	LDC Rm, SR	LDC Rm, GBR	LDC Rm, VBR	LDC Rm, SSR
0100	Rn	Rm	1111	MAC.W @Rm+, @Rn+			
0101	Rn	Rm	disp	MOV.L @(disp:4, Rm), Rn			
0110	Rn	Rm	00MD	MOV.B @Rm, Rn	MOV.W @Rm, Rn	MOV.L @Rm, Rn	MOV Rm, Rn
0110	Rn	Rm	01MD	MOV.B @Rm+, Rn	MOV.W @Rm+, Rn	MOV.L @Rm+, Rn	NOT Rm, Rn
0110	Rn	Rm	10MD	SWAP.B @Rm, Rn	SWAP.W @Rm, Rn	NEGC Rm, Rn	NEG Rm, Rn
0110	Rn	Rm	11MD	EXTU.B Rm, Rn	EXTU.W Rm, Rn	EXTS.B Rm, Rn	EXTS.W Rm, Rn
0111	Rn		imm	ADD #imm:8, Rn			
1000	00MD	Rn	disp	MOV.B R0, @(disp:4, Rn)	MOV.W R0, @(disp:4, Rn)		
1000	01MD	Rm	disp	MOV.B @(disp:4, Rm), R0	MOV.W @(disp:4, Rm), R0		
1000	10MD		imm/disp	CMP/EQ #imm:8, R0	BT disp:8		BF disp:8
1000	10MD		imm/disp		BT/S disp:8		BF/S disp:8

Instruction Code			Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011-1111
MSB	LSB		MD: 00	MD: 01	MD: 10	MD: 11
1001	Rn	disp	MOV.W @(disp:8,PC),Rn			
1010		disp	BRA disp:12			
1011		disp	BSR disp:12			
1100	00MD	imm/disp	MOV.B R0,@(disp:8,GBR)	MOV.W R0,@(disp:8,GBR)	MOV.L R0,@(disp:8,GBR)	TRAPA #imm:8
1100	01MD	disp	MOV.B @(disp:8,GBR),R0	MOV.W @(disp:8,GBR),R0	MOV.L @(disp:8,GBR),R0	MOVA @(disp:8,PC),R0
1100	10MD	imm	TST #imm:8,R0	AND #imm:8,R0	XOR #imm:8,R0	OR #imm:8,R0
1100	11MD	imm	TST.B #imm:8,@(R0,GBR)	AND.B #imm:8,@(R0,GBR)	XOR.B #imm:8,@(R0,GBR)	OR.B #imm:8,@(R0,GBR)
1101	Rn	disp	MOV.L @(disp:8,PC),R0			
1110	Rn	imm	MOV #imm:8,Rn			
1111		—	Floating-point instruction			

Appendix B Pipeline Operation and Contention

The SH-2E is designed so that basic instructions are executed in one cycle. Two or more cycles are required for instructions when, for example, the branch destination address is changed by a branch instruction or when the number of cycles is increased by contention between MA and IF. Table B.1 gives the number of execution cycles and stages for different types of contention and their instructions. Instructions without contention and instructions that require 2 or more cycles even without contention are also shown.

Instructions contend in the following ways:

CPU instructions

- Operations and transfers between registers are executed in one cycle with no contention.
- No contention occurs, but the instruction still requires 2 or more cycles.
- Contention occurs, increasing the number of execution cycles. Contention combinations are:
 - MA contends with IF
 - MA contends with IF and sometimes with memory loads as well
 - MA contends with IF and sometimes with the multiplier as well
 - MA contends with IF and sometimes with memory loads and sometimes with the multiplier

Floating-point instructions or FPU-related CPU instructions

- No contention occurs with the FCMP instruction.
- MA contends with IF in the case of store instructions involving FR0 to FR15 and FRUL.
- For floating-point operation instructions other than FDIV, floating-point register transfer instructions, and floating-point register immediate instructions, contention occurs if an instruction that reads from the destination of the instruction follows immediately after it.
- MA contends with IF in the case of load instructions involving FR0 to FR15 and FRUL. Also, contention occurs if an instruction that reads from the destination of the instruction follows immediately after it.
- Contention occurs if an instruction that uses Rn follows the STS FPUL,Rn or STS FPSCR,Rn instruction.
- In the case of FPSCR load instructions, contention occurs as shown in Figure 8.11.
- In the case of FPSCR store instructions, contention occurs as shown in Figure 8.12, and MA contends with IF.
- In the case of the FDIV instruction, contention occurs as shown in Figure 8.13.

Table B.1 Instructions and Their Contention Patterns

Contention	Cycles	Stages	Instructions
None	1	3	<ul style="list-style-type: none"> • Transfers between registers • Operations between registers (except when a multiplier is involved) • Logical operations between registers • Shift instructions • System control ALU instructions
	2	3	Unconditional branches
	3/1	3	Conditional branches
	3	3	SLEEP instruction
	4	5	RTE instruction
	8	9	TRAP instruction
	MA contends with IF	1	4
2		4	STC.L instruction
3		6	Memory logic operations
4		6	TAS instruction
MA contends with IF and sometimes with memory loads as well.	1	5	<ul style="list-style-type: none"> • Memory load instructions • LDS.L instruction (PR)
	3	5	LDC.L instruction
MA contends with IF and sometimes with the multiplier as well.	1	4	<ul style="list-style-type: none"> • Register to MAC transfer instructions • Memory to MAC transfer instructions • MAC to memory transfer instructions
	1 to 3*	6	Multiplication instructions
	3/(2)*	7	Multiply/accumulate instructions
	3/(2 to 4)*	9	Double length multiply/accumulate instructions (SH-2 CPU only)
	2 to 4*	9	Double length multiplication instructions (SH-2 CPU only)
MA contends with IF and sometimes with memory loads and sometimes with the multiplier.	1	5	MAC to register transfer instructions

Note: * The normal minimum number of execution states. (The number in parentheses is the number in contention with the preceding/following instructions.)

Table B.2 Types of Contention and Instruction Behavior (Floating-point Instructions or FPU-related CPU Instructions)

Contention	Cycles	Stages	Instructions
None	1	3 (FPU pipeline) 3 (CPU pipeline)	FCMP/EQ FRm, FRn FCMP/GT FRm, FRn
• MA in CPU pipeline contends with IF	1	4 (FPU pipeline) 4 (CPU pipeline)	STS.L FPUL, @-Rn FMOV.S FRm, @Rn FMOV.S FRm, @-Rn FMOV.S FRm, @ (R0, Rn)
• Contention occurs if next instruction reads destination register	1	5 (FPU pipeline) 3 (CPU pipeline)	FLDS FRm, FPUL FMOV FRm, FRn FSTS FPUL, FRn FLDI0 FRn FLDI1 FRn FABS FRn FADD FRm, FRn FLOAT FPUL, FRn FMAC FR0, FRm, FRn FMUL FRm, FRn FNEG FRn FSUB FRm, FRn FTRC FRm, FPUL
• Contention occurs if next instruction reads destination register	1	5 (FPU pipeline) 4 (CPU pipeline)	LDS Rm, FPUL LDS.L @Rm+, FPUL FMOV.S @Rm, FRn FMOV.S @Rm+, FRn FMOV.S @ (R0, Rm), FRn
• MA in CPU pipeline contends with IF	1	4 (FPU pipeline) 5 (CPU pipeline)	STS FPUL, Rn
• Contention occurs as shown in Figure 8.11	1	5 (FPU pipeline) 4 (CPU pipeline)	LDS Rm, FPSCR LDS.L @Rm+, FPSCR
• Contention occurs as shown in Figure 8.12	1	4 (FPU pipeline) 5 (CPU pipeline)	STS FPSCR, Rn
• Contention occurs if next instruction uses Rn			
• MA in CPU pipeline contends with IF			

Contention	Cycles	Stages	Instructions	
<ul style="list-style-type: none">• Contention occurs as shown in Figure 8.12• MA in CPU pipeline contends with IF	1	4 (FPU pipeline) 4 (CPU pipeline)	STS.L	FPSCR, @-Rn
<ul style="list-style-type: none">• Contention occurs as shown in Figure 8.13	13	17 (FPU pipeline) 3 (CPU pipeline)	FDIV	FRm, FRn

**Renesas 32-Bit RISC Microcomputer
Software Manual
SH-2E**

Publication Date: 1st Edition, March 1999
Rev.2.00, May 31, 2006

Published by: Sales Strategic Planning Div.
Renesas Technology Corp.

Edited by: Customer Support Department
Global Strategic Communication Div.
Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan



RENESAS SALES OFFICES

<http://www.renesas.com>

Refer to "<http://www.renesas.com/en/network>" for the latest and detailed information.

Renesas Technology America, Inc.

450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited

Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.

Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7898

Renesas Technology Hong Kong Ltd.

7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.

10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology Singapore Pte. Ltd.

1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.

Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> (2) 796-3115, Fax: <82> (2) 796-2145

Renesas Technology Malaysia Sdn. Bhd

Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jalan Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

SH-2E Software Manual



Renesas Electronics Corporation

1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

REJ09B0316-0200