Hitachi
Single-Chip RISC Microcomputers

SH7700 Series Programming Manual

Draft

Hitachi Micro Systems, Inc.
8/8/95
Joe Brennan

Notice

When using this document, keep the following in mind:

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

Introduction

The SH7700 Series 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.

These CPUs have 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.

This programming manual describes in detail the instructions for the SH7700 Series and is intended as a reference on instruction operation and architecture. It also covers the pipeline operation, which is a feature of the SH7700 Series. For information on the hardware, please refer to the hardware manual for the product in question.

Organization of This Manual

Table 1 describes how this manual is organized. Table 2 show the relationships between the items listed and lists the sections within this manual that cover those items.

Table 1 Manual Organization

Category	Section Title	Contents
Introduction	1. Features	CPU features
Architecture (1)	2. Programming model	Types and structure of general registers, control registers and system registers
	3. Data Formats	Data formats for registers and memory
Introduction to instructions	4. Instruction Features	Instruction features, addressing modes, and instruction formats
	5. Instruction Sets	Summary of instructions by category and list in alphabetic order
Detailed information on instructions	6. Description of Each Instruction	Operation of each instruction in alphabetical order
Architecture (2)	7. Processing States	Power-down and other processing states

Table 2 Subjects and Corresponding Sections

Category	Topic	Section Title
Introduction and features	CPU features	1. Features
	Instruction features	4.1 RISC-Type Instruction Set
	Pipelines	8.1 Basic Configuration of Pipelines 8.2 Slot and Pipeline Flow
Architecture	Organization of registers	2. Programming model
	Data formats	3. Data Formats
	Processing states, reset state, exception processing state, bus release state, program execution state, power-down state, sleep mode and standby mode	7. Processing States
	Pipeline operation	8. Pipeline Operation
Introduction to instructions	Instruction features	4. Instruction Features
	Addressing modes	4.2 Addressing Modes
	Instruction formats	4.3 Instruction Formats
List of instructions	Instruction sets	5.1 Instruction Set by Classification 5.2 Instruction Set in Alphabetical Order
Detailed information on instructions	Detailed information of instruction operation	6. Instruction Description 8.7 Instruction Pipelines
	Number of instruction execution states	8.3 Number of Instruction Execution Cycles

Contents

Section 1 Features

The SH7700 Series 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 SH7700 Series CPU features.

Table 1.1 SH7700 Series CPU Features

Feature	Description
Architecture	• Original Hitachi architecture • 32-bit internal data paths
General-register machine	• Sixteen 32-bit general registers (eight banked registers) • Five 32-bit control registers • Four 32-bit system registers
Instruction set	• 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	• One instruction/cycle for basic instructions (16.7 ns/instruction at 60-MHz operation)
Address space	• Architecture makes 4 Gbytes available
On-chip multiplier	• Multiplication operations (32 bits $\times$ 32 bits $\rightarrow$ 64 bits) executed in 2 to 5 cycles, and multiplication/accumulation operations (32 bits $\times$ 32 bits + 64 bits $\rightarrow$ 64 bits) executed in 2 to 5 cycles
Pipeline	• Five-stage pipeline
Processing states	• Reset state • Exception processing state • Program execution state • Power-down state • Bus release state
Power-down states	• Sleep mode • Standby mode • Module stop mode

Section 2 Programming Model

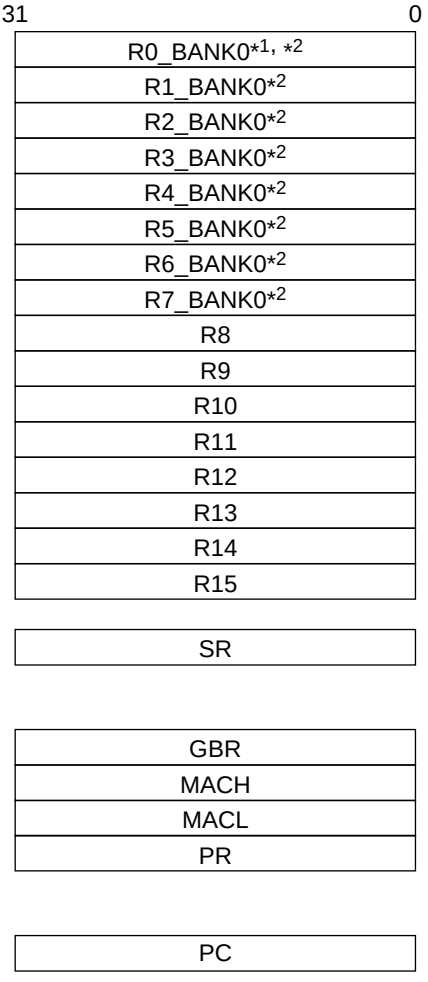
The SH7700 Series operates in user mode under normal conditions and enters privileged mode in response to an exception. Processor mode is specified by the mode (MD) bit in the status register (SR). The registers accessible to the programmer differ depending on the processor mode. General-purpose registers R0 to R7 are banked registers that are switched by a processor mode change.

In privileged mode (MD = 1), the register bank (RB) bit in the SR defines which banked register set is accessed as general-purpose, and which set is accessed only through the load control register (LDC) and store control register (STC) instructions. When the RB bit is logic one, bank 1 general-purpose registers R0–R7_BANK1 and nonbanked general-purpose registers R8–R15 function as the general-purpose register set, with bank 0 general-purpose registers R0–R7_BANK0 accessed only by the LDC/STC instructions.

When the RB bit is logic zero, bank 0 general-purpose registers R0–R7_BANK0 and nonbanked general-purpose registers R8–R15 function as the general-purpose register set, with bank 1 general-purpose registers R0–R7_BANK1 accessed only by the LDC/STC instructions.

In user mode (MD = 0), bank 0 general-purpose registers R0–R7_BANK0 and nonbanked general-purpose registers R8–R15 function as the general-purpose register set regardless of the SR.RB.

The programming model for user mode is shown in figure 2.1. Figure 2.2 shows the programming model for privileged mode. The registers are briefly defined in figures 2.3 and 2.4.



User Mode Programming Model

Notes: 1. R0 functions as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode. In some instructions, only R0 can be used as the source or destination register.

2. R0–R7 are banked registers. In user mode, BANK0 is used. In privileged mode, SR.RB specifies BANK. (SR.RB = 0: BANK0 is used. SR.RB = 1: BANK1 is used.)

Figure 2.1 User Mode Programming Model

31 … 0	31 … 0
R0_BANK1*1, *2	R0_BANK0*1, *2
R1_BANK1*2	R1_BANK0*2
R2_BANK1*2	R2_BANK0*2
R3_BANK1*2	R3_BANK0*2
R4_BANK1*2	R4_BANK0*2
R5_BANK1*2	R5_BANK0*2
R6_BANK1*2	R6_BANK0*2
R7_BANK1*2	R7_BANK0*2
R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13	R13
R14	R14
R15	R15
SR	SR
SSR	SSR
GBR	GBR
MACH	MACH
MACL	MACL
PR	PR
VBR	VBR
PC	PC
SPC	SPC
R0_BANK0*1, *3	R0_BANK1*1, *3
R1_BANK0*3	R1_BANK1*3
R2_BANK0*3	R2_BANK1*3
R3_BANK0*3	R3_BANK1*3
R4_BANK0*3	R4_BANK1*3
R5_BANK0*3	R5_BANK1*3
R6_BANK0*3	R6_BANK1*3
R7_BANK0*3	R7_BANK1*3
a. Privileged mode programming model (RB = 1)	b. Privileged mode programming model (RB = 0)

Notes:

1. R0 functions as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode. In some instructions, only R0 can be used as the source or destination register.

2. R0–R7 are banked registers. In user mode, BANK0 is used. In privileged mode, SR.RB specifies BANK.
   SR.RB = 0: BANK0 is used.
   SR.RB = 1: BANK1 is used.

3. These registes are only accessed by LDC/STC instructions. SR.RB specifies BANK.
   SR.RB = 0: BANK1 is used.
   SR.RB = 1: BANK0 is used.

Figure 2.2 Privileged Mode Programming Model

General Purpose Registers

31	0
R0*1, *2	
R1*2	
R2*2	
R3*2	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

Notes:
1. R0 functions as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode. In some instructions, only R0 can be used as the source or destination register.
2. R0–R7 are banked registers.
   In user mode, R0_BANK0–R7_BANK0 are used.
   In privileged mode:
   SR.RB = 0: R0_BANK0–R7_BANK0 are used.
   SR.RB = 1: R0_BANK1–R7_BANK1 are used.

System Registers

31	0
MACH	
MACL	

Multiply and Accumulate High and Low Registers (MACH/MACL): Store the results of multiply and accumulate operations.

31	0
PR	

Procedure Register (PR): Stores the return address for exiting subroutines.

31	0
PC	

Program Counter (PC): Indicates starting address of the current instruction incremented by 4.

Figure 2.3 Register Set Overview, GPRs, and System Registers

31	0
SSR	

Saved Status Register (SSR): Stores current SR value at the time of exception to indicate processor status for the return to instruction stream from the exception handler.

31	0
SPC	

Saved Program Counter (SPC): Stores current PC value at the time of exception to indicate the return address at completion of exception processing.

31	0
GBR	

Global Base Register (GBR): Stores the base address of the GBR-indirect addressing mode. This mode transfers data to the register areas of the resident peripheral modules, and is used for logic operations.

31	0
VBR	

Vector Base Register (VBR): Stores the base address of the exception processing vector area.

31	30	29	28	27		10	9	8	7		3		1	0	
0	MD	RB	BL	0	———	0	M	Q	IMASK		0	0	S	T	Status register

T bit: The MOVT, CMP/cond, TAS, TST, BT, BF, SETT, CLRT, and DT instructions use the T bit to indicate true (logical 1) or false (logical 0). The ADDV/C, SUBV/C, DIV0U/S, DIV1, NEGC, SHAR/L, SHLR/L, ROTR/L, and ROTCR/L instructions also use the T bit to indicate a carry, borrow, overflow, or underflow.

S bit: Used by the MAC instruction.

Zero bits: Always read as 0, and should always be written as 0.

IMASK: 4-bit field indicating the interrupt request mask level.

M, Q bits: Used by the DIV0U/S and DIV1 instructions.

BL bit: Block bit, used to mask exceptions in privileged mode.
BL = 1: interrupts are masked (not accepted).
User break trap exception is neglected. Other exceptions cause the reset exception.
In sleep or standby mode, interrupts are accepted.
BL = 0: exceptions and interrupts are accepted.

RB bit: Register bank bit; used to define the general purpose registers.
RB = 1: R0_BANK1–R7_BANK1 are accessed as general purpose registers.
R0_BANK0–R7–BANK0 are accessed by LDC/STC instructions.
RB = 0: R0_BANK0–R7_BANK0 are accessed as general purpose registers.
R0_BANK1–BANK–R7_BANK1 are accessed by LDC/STC instructions.

MD: Processor operation mode field, indicates the processor mode.
MD = 1: privileged mode
MD = 0: user mode

Only the M, Q, S, and T bits are read or written from user mode. All other bits are read or written from privileged mode.

Figure 2.4 Register Set Overview, Control Registers

2.1 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 register	R0–R15	Undefined
Control register	SR	Bits I3–I0 are 1111 (H'F), reserved bits are 0, and other bits are undefined
	GBR	Undefined
	VBR	H'00000000
	SSR, SPC	Undefined
System register	MACH, MACL, PR	Undefined
	PC	H'A0000000

Section 3 Data Formats

3.1 Data Format in Registers

Register operands are always longwords (32 bits) (figure 3.1). When the memory operand is only a byte (8 bits) or a word (16 bits), it is sign-extended into a longword when loaded into a register.

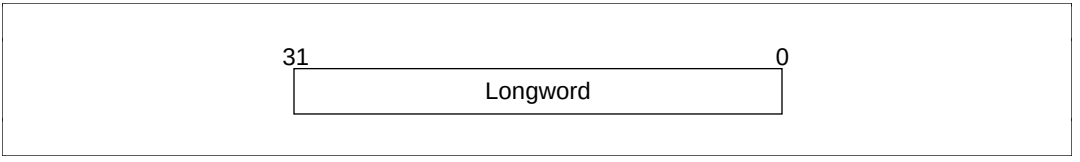


Figure 3.1 Longword Operand

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 (figure 3.2). See the *SH7700 Series Hardware Manual* for more information on address errors.

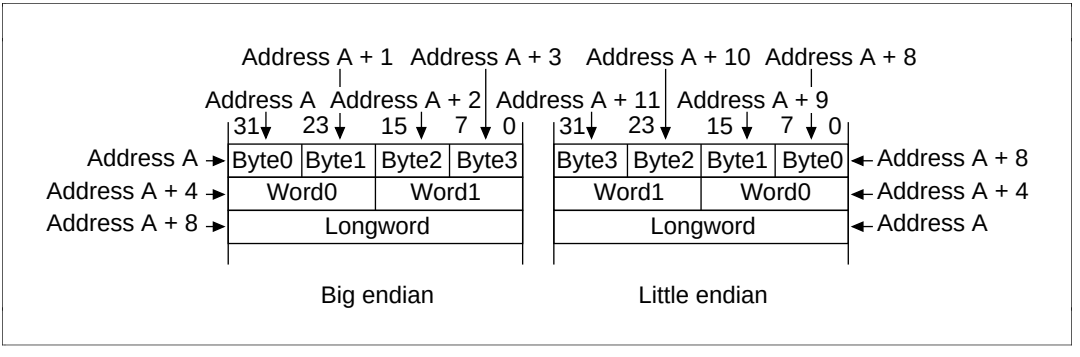


Figure 3.2 Byte, Word, and Longword Alignment

Address can be configured in either big endian or little endian byte order, according to the MD5 pin at reset. When MD5 is low at reset, the processor operates in big endian. When MD5 is high at reset, the processor operates in little endian. In little endian mode, data written in byte (word) size must be read in byte (word) size.

Section 4 Instruction Features

4.1 RISC-Type Instruction Set

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

4.1.1 16-Bit Fixed Length

All instructions are 16 bits long, increasing program coding efficiency.

4.1.2 One Instruction/Cycle

Basic instructions can be executed in one cycle using the pipeline system. Instructions are executed in 16.7 ns at 60 MHz.

4.1.3 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 handled as longword data (table 4.1). Immediate data is sign-extended for arithmetic operations or zero-extended for logic operations. It also is handled as longword data.

Table 4.1 Sign Extension of Word Data

SH7000 Series CPU	Description	Example for Conventional CPU

`MOV.W @(disp,PC),R1`
`ADD R1,R0`
`.....`
`.DATA.W H'1234`
	Data is sign-extended to 32 bits, and R1 becomes H'00001234. It is next operated upon by an ADD instruction.	`ADD.W #H'1234,R0`

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

4.1.4 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.

4.1.5 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 4.2).

Table 4.2 Delayed Branch Instructions

SH7000 Series CPU		Description	Example for Conventional CPU	
BRA	TRGET	Executes an ADD before branching to TRGET.	ADD.W	R1,R0
ADD	R1,R0		BRA	TRGET

4.1.6 Multiplication/Accumulation Operation

The five-stage pipeline system and on-chip multiplier enable 32-bit × 32-bit → 64-bit multiplication operations to be executed in two to five cycles. 32-bit × 32-bit + 64-bit → 64 bit multiplication/accumulation operations are executed in two to five cycles.

4.1.7 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 (table 4.3). The number of instructions after T bit in the status register is kept to a minimum to improve the processing speed.

Table 4.3 T Bit

SH7000 Series CPU		Description	Example for Conventional CPU	
CMP/GE	R1,R0	T bit is set when R0 ≥ R1. The program branches to TRGET0 when R0 ≥ R1 and to TRGET1 when R0 < R1.	CMP.W	R1,R0
BT	TRGET0		BGE	TRGET0
BF	TRGET1		BLT	TRGET1
ADD	#–1,R0	T bit is not changed by ADD. T bit is set when R0 = 0. The program branches if R0 = 0.	SUB.W	#1,R0
CMP/EQ	#0,R0		BEQ	TRGET
BT	TRGET			

4.1.8 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 4.4).

Table 4.4 Immediate Data Accessing

Classification	SH7000 Series CPU	Example for Conventional CPU
8-bit immediate	`MOV #H'12,R0`	`MOV.B #H'12,R0`

16-bit immediate	`MOV.W @(disp,PC),R0`
	`.....`
	`.DATA.W H'1234`
		`MOV.W #H'1234,R0`

32-bit immediate	`MOV.L @(disp,PC),R0`
	`.....`
	`.DATA.L H'12345678`
		`MOV.L #H'12345678,R0`

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

4.1.9 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 4.5 Absolute Address

Classification	SH7000 Series CPU	Example for Conventional CPU

Absolute address	`MOV.L @(disp,PC),R1`
	`MOV. B @R1,R0`
	`.....`
	`.DATA.L H'12345678`
		`MOV.B @H'12345678,R0`

4.1.10 16-Bit/32-Bit Displacement

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

Table 4.6 16-Bit/32-Bit Displacement

Classification	SH7000 Series CPU	Example for Conventional CPU

16-bit displacement	`MOV.W @(disp,PC),R0`
	`MOV.W @(R0,R1),R2`
	`.....`
	`.DATA.W H'1234`
		`MOV.W @(H'1234,R1),R2`

4.1.11 Privileged Instructions

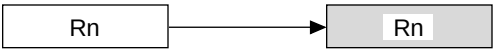
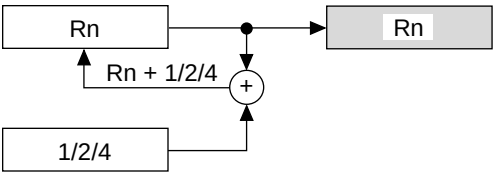
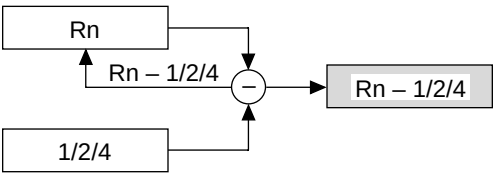
The processor has two operation modes (user/privileged). If these instructions are used in user mode, an illegal instruction exception is detected. Privileged instructions are:

- LDC
- STC
- RTE
- LDTLB
- SLEEP

4.2 Addressing Modes

Addressing modes and effective address calculation are described in table 4.7.

Table 4.7 Addressing Modes and Effective Addresses

Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
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, and 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, and 4 for a longword operation. 	Byte: Rn – 1 → Rn Word: Rn – 2 → Rn Longword: Rn – 4 → Rn (Instruction executed with Rn after calculation)

: Effective address

Table 4.7 Addressing Modes and Effective Addresses (cont)

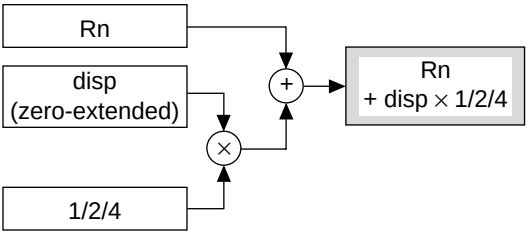
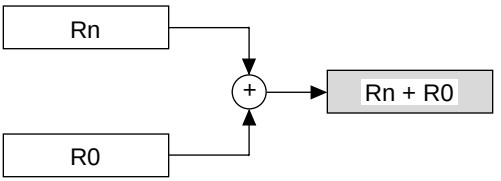
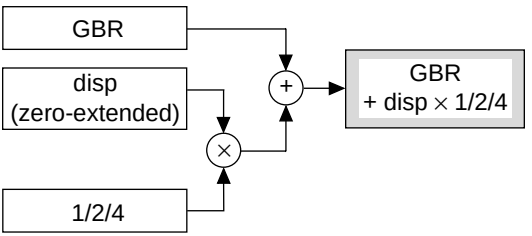
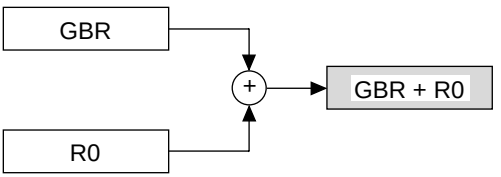
Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
Indirect register addressing with displace-ment	@(disp:4, Rn)	The effective address is Rn plus a 4-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, and is quadrupled for a longword operation. 	Byte: Rn + disp Word: Rn + disp × 2 Longword: Rn + disp × 4
Indirect indexed register addressing	@(R0, Rn)	The effective address is the Rn value plus R0. 	Rn + R0
Indirect GBR addressing with displace-ment	@(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, and is quadrupled for a longword operation. 	Byte: GBR + disp Word: GBR + disp × 2 Longword: GBR + disp × 4
Indirect indexed GBR addressing	@(R0, GBR)	The effective address is the GBR value plus the R0. 	GBR + R0

Table 4.7 Addressing Modes and Effective Addresses (cont)

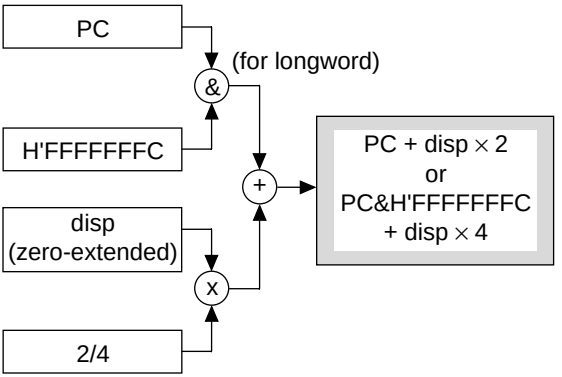
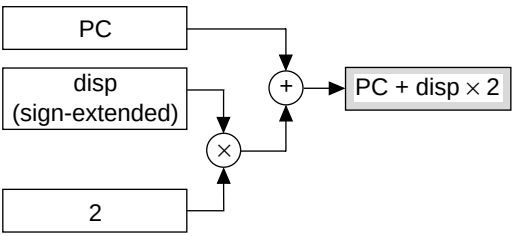
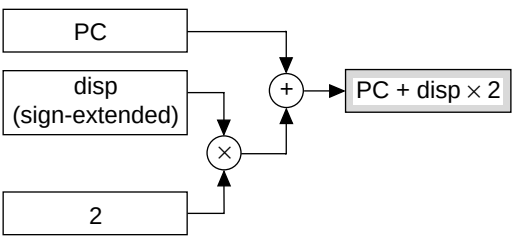
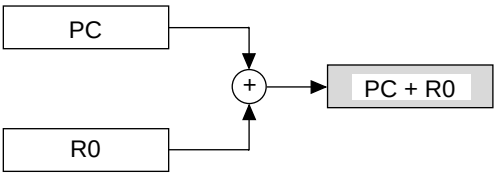
Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
Indirect PC addressing with displace-ment	@(disp:8, PC)	The effective address is the PC value plus an 8-bit displacement (disp). The value of disp is zero-extended, and remains the same for a byte operation, is doubled for a word operation, and is quadrupled for a longword operation. For a longword operation, the lowest two bits of the PC are masked. 	Word: PC + disp × 2 Longword: PC & H'FFFFFFFC + disp × 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 × 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 × 2

Table 4.7 Addressing Modes and Effective Addresses (cont)

Addressing Mode	Instruction Format	Effective Addresses Calculation	Equation
PC relative addressing (cont)	Rn	The effective address is the register PC plus R0. 	PC + R0
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.	—

4.3 Instruction Format

The instruction format table, table 4.8, refers to the source operand and the destination operand. The meaning of the operand depends on the instruction code. The symbols are used as follows:

- xxxx: Instruction code
- mmmm: Source register
- nnnn: Destination register
- iiii: Immediate data
- dddd: Displacement

Table 4.8 Instruction Formats

Instruction Formats	Source Operand	Destination Operand	Example
0 format 15 … 0 xxxx xxxx xxxx xxxx	—	—	NOP
n format 15 … 0 xxxx nnnn xxxx xxxx	—	nnnn: Direct register	MOVT Rn
	Control register or system register	nnnn: Direct register	STS MACH, Rn

Table 4.8 Instruction Formats (cont)

Instruction Formats	Source Operand	Destination Operand	Example
n format (cont)	—	nnnn: Direct register	`JMP @Rn`
	Control register or system register	nnnn: Indirect pre-decrement register	`STC.L SR,@-Rn`
	—	nnnn: PC relative using Rn	`BRAF Rn`
m format 15 0 xxxx mmmm xxxx xxxx	mmmm: Direct register	Control register or system register	`LDC Rm,SR`
	mmmm: Indirect post-increment register	Control register or system register	`LDC.L @Rm+,SR`
nm format 15 0 xxxx nnnn mmmm xxxx	mmmm: Direct register	nnnn: Direct register	`ADD Rm,Rn`
	mmmm: Direct register	nnnn: Direct register	`MOV.L Rm,@Rn`
	mmmm: Indirect post-increment register (multiply/ accumulate) nnnn: Indirect post-increment register (multiply/ accumulate)*	MACH, MACL	`MAC.W @Rm+,@Rn+`
	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 15 0 xxxx xxxx mmmm dddd	mmmmdddd: indirect register with displacement	R0 (Direct register)	`MOV.B @(disp,Rm),R0`
nd4 format 15 0 xxxx xxxx nnnn dddd	R0 (Direct register)	nnnndddd: Indirect register with displacement	`MOV.B R0,@(disp,Rn)`

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

Table 4.8 Instruction Formats (cont)

Instruction Formats	Source Operand	Destination Operand	Example
nmd format 15 0 xxxx nnnn mmmm dddd	mmmm: Direct register	nnnnnddd: Indirect register with displacement	`MOV.L Rm,@(disp,Rn)`
	mmmmdddd: Indirect register with displacement	nnnn: Direct register	`MOV.L @(disp,Rm),Rn`
d format 15 0 xxxx xxxx dddd dddd	dddddddd: Indirect GBR with displacement	R0 (Direct register)	`MOV.L @(disp,GBR),R0`
	R0(Direct register)	dddddddd: Indirect GBR with displacement	`MOV.L R0,@(disp,GBR)`
	dddddddd: PC relative with displacement	R0 (Direct register)	`MOVA @(disp,PC),R0`
	—	dddddddd: PC relative	`BF label`
d12 format 15 0 xxxx dddd dddd dddd	—	ddddddddddddd: PC relative	`BRA label` `(label = disp + PC)`
nd8 format 15 0 xxxx nnnn dddd dddd	dddddddd: PC relative with displacement	nnnn: Direct register	`MOV.L @(disp,PC),Rn`
i format 15 0 xxxx xxxx iiii iiii	iiiiiiii: Immediate	Indirect indexed GBR	`AND.B #imm,@(R0,GBR)`
	iiiiiiii: Immediate	R0 (Direct register)	`AND #imm,R0`
	iiiiiiii: Immediate	—	`TRAPA #imm`
ni format 15 0 xxxx nnnn iiii iiii	iiiiiiii: Immediate	nnnn: Direct register	`ADD #imm,Rn`

Section 5 Instruction Set

5.1 Instruction Set by Classification

The SH7700 Series instruction set includes 66 basic instruction types, divided into six functional classifications, as shown in table 5.1. Tables 5.3 to 5.9 summarize instruction notation, machine mode, execution time, and function.

Table 5.1 Classification of Instructions

Classification	Types	Operation Code	Function	No. of Instructions
Data transfer	6	MOV	Data transfer Immediate data transfer Peripheral module data transfer Structure data transfer	40
		MOVA	Effective address transfer	
		MOVT	T bit transfer	
		SWAP	Swap of upper and lower bytes	
		XTRCT	Extraction of the middle of registers connected	
		PREF	Prefetching data to cache	
Arithmetic operations	21	ADD	Binary addition	34
		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/accumulate, double-length multiply/accumulate operation	
		MUL	Double-length multiplication (32 × 32 bits)	
		MULS	Signed multiplication (16 × 16 bits)	
		MULU	Unsigned multiplication (16 × 16 bits)	
		NEG	Negation	
		NEGC	Negation with borrow	
		SUB	Binary subtraction	
		SUBC	Binary subtraction with carry	
		SUBV	Binary subtraction with underflow check	

Table 5.1 Classification of Instructions (cont)

Classification	Types	Operation Code	Function	No. of Instructions
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	12	ROTL	One-bit left rotation	16
		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	
		SHAD	Dynamic arithmetic shift	
		SHLD	Dynamic logical shift	
Branch	9	BF	Conditional branch, conditional branch with delay (T = 0)	11
		BT	Conditional branch, conditional branch with delay (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	

Table 5.1 Classification of Instructions (cont)

Classification	Types	Operation Code	Function	No. of Instructions
System control	14	CLRT	T bit clear	74
		CLRMAC	MAC register clear	
		CLRS	S bit clear	
		LDC	Load to control register	
		LDS	Load to system register	
		LDTLB	Load PTE to TLB	
		NOP	No operation	
		RTE	Return from exception processing	
		SETS	S bit set	
		SETT	T bit set	
		SLEEP	Shift into power-down mode	
		STC	Storing control register data	
		STS	Storing system register data	
		TRAPA	Trap exception handling	
Total:	66			189

Instruction codes, operation, and execution states are listed as shown in table 5.2 in order by classification.

Tables 5.3 to 5.8 list the minimum number of clock cycles required for execution. In practice, the number of execution cycles increases when the instruction fetch is in contention with data access or when the destination register of a load instruction (memory → register) is the same as the register used by the next instruction.

Table 5.2 Instruction Code Format

Item	Format	Explanation
Instruction mnemonic	`OP.Sz SRC,DEST`	OP: Operation code Sz: Size SRC: Source DEST: Destination Rm: Source register Rn: Destination register imm: Immediate data disp: Displacement
Instruction code	MSB ↔ LSB	mmmm: Source register nnnn: Destination register 0000: R0 0001: R1 ..... 1111: R15 iiii: Immediate data dddd: Displacement
Operation summary	→, ← (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 shift
Execution cycle		Value when no wait states are inserted
Instruction execution cycles		The execution cycles shown in the table are minimums. The actual number of cycles may be increased: 1. When 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.
T bit		Value of T bit after instruction is executed
—		No change

Note: Scaling (x1, x2, x4) is performed according to the instruction operand size. See "6. Instruction Descriptions" for details.

5.1.1 Data Transfer Instructions

Table 5.3 Data Transfer Instructions

Instruction		Operation	Code	Cycles	T Bit
MOV	#imm,Rn	#imm → Sign extension → Rn	1110nnnniiiiiiii	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	—
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	R0,@(disp,Rn)	R0 → (disp + Rn)	10000000nnnnddddd	1	—
MOV.W	R0,@(disp,Rn)	R0 → (disp × 2 + Rn)	10000001nnnnddddd	1	—
MOV.L	Rm,@(disp,Rn)	Rm → (disp × 4 + Rn)	0001nnnnmmmmddddd	1	—
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	—
MOV.L	@(disp,Rm),Rn	(disp × 4 + Rm) → Rn	0101nnnnmmmmddddd	1	—
MOV.B	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0100	1	—
MOV.W	Rm,@(R0,Rn)	Rm → (R0 + Rn)	0000nnnnmmmm0101	1	—

Table 5.3 Data Transfer Instructions (cont)

Instruction		Operation	Code	Cycles	T Bit
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	—
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	—
MOVA	@(disp,PC),R0	disp × 4 + PC → R0	11000111dddddddd	1	—
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—
SWAP.B	Rm,Rn	Rm → Swap the bottom two bytes → REG	0110nnnnmmmm1000	1	—
SWAP.W	Rm,Rn	Rm → Swap two consecutive words → Rn	0110nnnnmmmm1001	1	—
XTRCT	Rm,Rn	Rm: Middle 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

5.1.2 Arithmetic Instructions

Table 5.4 Arithmetic Instructions

Instruction		Operation	Code	Cycles	T Bit
ADD	Rm,Rn	Rn + Rm → Rn	0011nnnnmmmm1100	1	—
ADD	#imm,Rn	Rn + imm → Rn	0111nnnniiiiiiii	1	—
ADDC	Rm,Rn	Rn + Rm + T → Rn, Carry → T	0011nnnnmmmm1110	1	Carry
ADDV	Rm,Rn	Rn + Rm → Rn, Overflow → T	0011nnnnmmmm1111	1	Overflow
CMP/EQ	#imm,R0	If R0 = imm, 1 → T	10001000iiiiiiii	1	Comparison result
CMP/EQ	Rm,Rn	If Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/HS	Rm,Rn	If Rn≥Rm with unsigned data, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/GE	Rm,Rn	If Rn ≥ Rm with signed data, 1 → T	0011nnnnmmmm0011	1	Comparison result
CMP/HI	Rm,Rn	If Rn > Rm with unsigned data, 1 → T	0011nnnnmmmm0110	1	Comparison result
CMP/GT	Rm,Rn	If Rn > Rm with signed data, 1 → T	0011nnnnmmmm0111	1	Comparison result
CMP/PZ	Rn	If Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
CMP/PL	Rn	If Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/STR	Rm,Rn	If Rn and Rm have an equivalent byte, 1 → T	0010nnnnmmmm1100	1	Comparison result
DIV1	Rm,Rn	Single-step division (Rn/Rm)	0011nnnnmmmm0100	1	Calculation result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q →T	0010nnnnmmmm0111	1	Calculation result
DIV0U		0 → M/Q/T	0000000000011001	1	0

Table 5.4 Arithmetic Instructions (cont)

Instruction		Operation	Code	Cycles	T Bit
DMULS.L	Rm,Rn	Signed operation of Rn × Rm → MACH, MACL 32 × 32 → 64 bits	0011nnnnmmmm1101	2 (to 5)*1	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL 32 × 32 → 64 bits	0011nnnnmmmm0101	2 (to 5)*1	—
DT	Rn	Rn – 1 → Rn, if Rn = 0, 1 → T, else 0 → T	0100nnnn00010000	1	Comparison result
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	—
MAC.L	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	2 (to 5)*1	—
MAC.W	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC 16 × 16 + 64 → 64 bits	0100nnnnmmmm1111	2 (to 5)*1	—
MUL.L	Rm,Rn	Rn × Rm → MACL 32 × 32 → 32 bits	0000nnnnmmmm0111	2 (to 5)*1	—
MULS.W	Rm,Rn	Signed operation of Rn × Rm → MAC 16 × 16 → 32 bits	0010nnnnmmmm1111	1 (to 3)*2	—
MULU.W	Rm,Rn	Unsigned operation of Rn × Rm → MAC 16 × 16 → 32 bits	0010nnnnmmmm1110	1 (to 3)*2	—

Table 5.4 Arithmetic Instructions (cont)

Instruction		Operation	Code	Cycles	T Bit
NEG	Rm,Rn	0–Rm → Rn	0110nnnnmmmm1011	1	—
NEGC	Rm,Rn	0–Rm–T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
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

Notes:
1. The normal minimum number of execution cycles is 2, but 5 cycles are required when the results of an operation are read from the MAC register immediately after the instruction.
2. The normal minimum number of execution cycles is 1, but 3 cycles are required when the results of an operation are read from the MAC register immediately after a MUL instruction.

5.1.3 Logic Operation Instructions

Table 5.5 Logic Operation Instructions

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

5.1.4 Shift Instructions

Table 5.6 Shift Instructions

Instruction		Operation	Code	Cycles	T Bit
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (MSB→)Rn	0100nnnnmmmm1100	1	—
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (0→)Rn	0100nnnnmmmm1101	1	—
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLR	Rn	0 → Rn → T	0100nnnn00000001	1	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—

5.1.5 Branch Instructions

Table 5.7 Branch Instructions

Instruction		Operation	Code	Cycles	T Bit
BF	label	If T = 0, disp × 2 + PC → PC; if T = 1, nop	10001011dddddddd	3/1*	—
BF/S	label	Delayed branch, if T = 0, disp × 2 + PC → PC; if T = 1, nop	10001111dddddddd	2/1*	
BT	label	Delayed branch, if T = 1, disp × 2 + PC → PC; if T = 0, nop	10001001dddddddd	3/1*	—
BT/S	label	If T = 1, disp × 2 + PC → PC; if T = 0, nop	10001101dddddddd	2/1*	—
BRA	label	Delayed branch, disp × 2 + PC → PC	1010dddddddddddd	2	—
BRAF	Rn	Rn + PC → PC	0000nnnn00100011	2	—
BSR	label	Delayed branch, PC → PR, disp × 2 + PC → PC	1011dddddddddddd	2	—
BSRF	Rn	PC → PR, Rn + PC → PC	0000nnnn00000011	2	—
JMP	@Rn	Delayed branch, Rn → PC	0100nnnn00101011	2	—
JSR	@Rn	Delayed branch, PC → PR, Rn → PC	0100nnnn00001011	2	—
RTS		Delayed branch, PR → PC	0000000000001011	2	—

Note: One state when it does not branch.

5.1.6 System Control Instructions

Table 5.8 System Control Instructions

Instruction		Operation	Code	Cycles	T Bit
CLRMAC		0 → MACH, MACL	0000000000101000	1	—
CLRS		0 → S	0000000001001000	1	—
CLRT		0 → T	0000000000001000	1	0
LDC	Rm,SR	Rm → SR	0100mmmm00001110	5	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	1	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—
LDC.L	@Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	7	LSB
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
LDC.L	@Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC.L	@Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC.L	@Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC.L	@Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC.L	@Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC.L	@Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—

Table 5.8 System Control Instructions (cont)

Instruction		Operation	Code	Cycles	T Bit
LDC.L	@Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC.L	@Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC.L	@Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC.L	@Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	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+,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	—
LDTLB		PTEH/PTEL → TLB	0000000000111000	1	—
NOP		No operation	0000000000001001	1	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	4	—
SETS		1 → S	0000000001011000	1	—
SETT		1 → T	0000000000011000	1	1
SLEEP		Sleep	0000000000011011	4	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—

Table 5.8 System Control Instructions (cont)

Instruction	Operation	Code	Cycles	T Bit
STC R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
STC R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	1	—
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	—
STC.L SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
STC.L SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—
STC.L R0_BANK,@–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L R1_BANK,@–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L R2_BANK,@–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L R3_BANK,@–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L R4_BANK,@–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L R5_BANK,@–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L R6_BANK,@–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L R7_BANK,@–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—
STS MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS PR,Rn	PR → Rn	0000nnnn00101010	1	—

Table 5.8 System Control Instructions (cont)

Instruction		Operation	Code	Cycles	T Bit
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	—
TRAPA	#imm	PC/SR → SPC/SSR, #imm<<2 → TRA, 0x160 → EXPEVT VBR + H'0100 → PC	11000011iiiiiiii	6	—

Note: The number of execution states before the chip enters the sleep state. This table lists the minimum execution cycles. In practice, the number of execution cycles increases when the instruction fetch is in contention with data access or when the destination register of a load instruction (memory → register) is the same as the register used by the next instruction.

5.2 Instruction Set in Alphabetical Order

Table 5.9 alphabetically lists the instruction codes and number of execution cycles for each instruction.

Table 5.9 Instruction Set Listed Alphabetically

Instruction		Operation	Code	Cycles	T Bit
ADD	#imm,Rn	Rn + imm → Rn	0111nnnniiiiiiii	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	10001011dddddddd	3/1*2	—
BF/S	label	If T = 0, disp + PC → PC; if T = 1, nop	10001111dddddddd	2/1*2	—
BRA	label	Delayed branch, disp + PC → PC	1010dddddddddddd	2	—
BRAF	Rn	Delayed branch, Rn + PC → PC	0000nnnn00100011	2	—
BSR	label	Delayed branch, PC → PR, disp + PC → PC	1011dddddddddddd	2	—
BSRF	Rn	Delayed branch, PC → PR, Rn + PC → PC	0000nnnn00000011	2	—
BT	label	If T = 1, disp + PC → PC; if T = 0, nop	10001001dddddddd	3/1*2	—
BT/S	label	If T = 1, disp + PC → PC; if T = 0, nop	10001101dddddddd	2/1*2	—
CLRMAC		0 → MACH, MACL	0000000000101000	1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
CLRS		0 → S	0000000001001000	1	—
CLRT		0 → T	0000000000001000	1	0
CMP/EQ	#imm,R0	If R0 = imm, 1 → T	10001000iiiiiiii	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	0000000000011001	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 5)*1	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 (to 5)*1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
DT	Rn	Rn - 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result
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	—
JMP	@Rn	Delayed branch, Rn → PC	0100nnnn00101011	2	—
JSR	@Rn	Delayed branch, PC → PR, Rn → PC	0100nnnn00001011	2	—
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	1	—
LDC	Rm,SR	Rm → SR	0100mmmm00001110	5	LSB
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
LDC.L @Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	7	LSB
LDC.L @Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1	—
LDC.L @Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC.L @Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC.L @Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC.L @Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC.L @Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC.L @Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—
LDC.L @Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC.L @Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC.L @Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC.L @Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	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+,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 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
LDTLB		PTEH/PTEL → TLB	0000000000111000	1	—
MAC.L	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	2 (to 5)*1	—
MAC.W	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	2 (to 5)*1	—
MOV	#imm,Rn	#imm → Sign extension → Rn	1110nnnniiiiiiii	1	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MOV.B	@(disp,GBR),R0	(disp + GBR) → Sign extension → R0	11000100dddddddd	1	—
MOV.B	@(disp,Rm),R0	(disp + Rm) → Sign extension → R0	10000100mmmmdddd	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)	11000000dddddddd	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 + GBR) → R0	11000110dddddddd	1	—
MOV.L	@(disp,PC),Rn	(disp + PC) → Rn	1101nnnnddddddd	1	—
MOV.L	@(disp,Rm),Rn	(disp + Rm) → Rn	0101nnnnmmmmdddd	1	—
MOV.L	@(R0,Rm),Rn	(R0 + Rm) → Rn	0000nnnnmmmm1110	1	—
MOV.L	@Rm+,Rn	(Rm) → Rn, Rm + 4 → Rm	0110nnnnmmmm0110	1	—
MOV.L	@Rm,Rn	(Rm) → Rn	0110nnnnmmmm0010	1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
MOV.L	R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	1	—
MOV.L	Rm,@(disp,Rn)	Rm → (disp + Rn)	0001nnnnmmmmdddd	1	—
MOV.L	Rm,@(R0,Rn)	Rm → (R0 + 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 + GBR) → Sign extension → R0	11000101dddddddd	1	—
MOV.W	@(disp,PC),Rn	(disp + PC) → Sign extension → Rn	1001nnnnddddddd	1	—
MOV.W	@(disp,Rm),R0	(disp + Rm) → Sign extension → R0	10000101mmmmdddd	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 + GBR)	11000001dddddddd	1	—
MOV.W	R0,@(disp,Rn)	R0 → (disp + Rn)	10000001nnnnddd	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 + PC → R0	11000111dddddddd	1	—
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
MUL.L	Rm,Rn	Rn × Rm → MAC	0000nnnnmmmm0111	2 (to 5)*1	—
MULS.W	Rm,Rn	Signed operation of Rn × Rm → MAC	0010nnnnmmmm1111	1 (to 3)*1	—
MULU.W	Rm,Rn	Unsigned operation of Rn × Rm → MAC	0010nnnnmmmm1110	1 (to 3)*1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
NEG	Rm,Rn	0–Rm → Rn	0110nnnnmmmm1011	1	—
NEGC	Rm,Rn	0–Rm–T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
NOP		No operation	0000000000001001	1	—
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
OR.B	#imm, @(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	4	—
RTS		Delayed branch, PR → PC	0000000000001011	2	—
SETS		1 → S	0000000001011000	1	—
SETT		1 → T	0000000000011000	1	1
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (MSB→)Rn	0100nnnnmmmm1100	1	—
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (0→)Rn	0100nnnnmmmm1101	1	—
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR	Rn	0 → Rn → T	0100nnnn00000001	1	LSB

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
SHLR2	Rn	Rn>>2 → Rn	0100nnnn00001001	1	—
SHLR8	Rn	Rn>>8 → Rn	0100nnnn00011001	1	—
SHLR16	Rn	Rn>>16 → Rn	0100nnnn00101001	1	—
SLEEP		Sleep	0000000000011011	4	—
STC	GBR,Rn	GBR → Rn	0000nnnn00010010	1	—
STC	SR,Rn	SR → Rn	0000nnnn00000010	1	—
STC	VBR,Rn	VBR → Rn	0000nnnn00100010	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	1	—
STC.L	GBR,@–Rn	Rn–4 → Rn, GBR → (Rn)	0100nnnn00010011	1	—
STC.L	SR,@–Rn	Rn–4 → Rn, SR → (Rn)	0100nnnn00000011	1	—
STC.L	VBR,@–Rn	Rn–4 → Rn, VBR → (Rn)	0100nnnn00100011	1	—
STC.L	SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
STC.L	SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
STC.L	R0_BANK,@–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L	R1_BANK,@–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L	R2_BANK,@–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L	R3_BANK,@–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L	R4_BANK,@–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L	R5_BANK,@–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L	R6_BANK,@–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L	R7_BANK,@–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—
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	—
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	Under-flow
SWAP.B	Rm,Rn	Rm → Swap the two lowest-order bytes → Rn	0110nnnnmmmm1000	1	—
SWAP.W	Rm,Rn	Rm → Swap two consecutive words → Rn	0110nnnnmmmm1001	1	—

Table 5.9 Instruction Set Listed Alphabetically (cont)

Instruction		Operation	Code	Cycles	T Bit
TAS.B	@Rn	If (Rn) is 0, 1 → T; 1 → MSB of (Rn)	0100nnnn00011011	3	Test result
TRAPA	#imm	PC/SR → SPC/SSR, (#imm) <<2 → TRA VBR + H'0100 → PC	11000011iiiiiiii	6	—
TST	#imm,R0	R0 & imm; if the result is 0, 1 →T	11001000iiiiiiii	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	11001100iiiiiiii	3	Test result
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XOR.B	#imm, @(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—
XTRCT	Rm,Rn	Rm: Middle 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

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 it does not branch.

Section 6 Instruction Descriptions

This section describes instructions in alphabetical order using the format shown below in section 6.1. The actual descriptions begin at section 6.2.

6.1 Sample Description (Name): Classification

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. This part is just a reference to help understanding of an operation. 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. The following instructions are detected before execution as having illegal slots (they 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);
```

- 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 dummy0:22;
    unsigned long  M0:1;
    unsigned long  Q0:1;
    unsigned long  I0:4;
    unsigned long dummy1:2;
    unsigned long  S0:1;
    unsigned long  T0:1;
};
```

- Definition of bits in SR:

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

- Error display function:

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

The PC should point to the location four bytes (the second instruction) 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.*

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

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

6.2 ADD (Add Binary): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
ADD	Rm,Rn	Rm + Rn → Rn	0011nnnnmmmm1100	1	—
ADD	#imm,Rn	Rn + #imm → Rn	0111nnnniiiiiiii	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]+=(0xFFFFFF00 | (long)i);
    PC+=2;
}
```

Examples:

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

6.3 ADDC (Add with Carry): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
ADDC	Rm,Rn	Rn + Rm + T → Rn, carry → T	0011nnnnmmmm1110	1	Carry

Description: Adds general register Rm data and the T bit to 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'FFFFFFFF
		After execution	T = 1, R1 = H'0000000
ADDC	R2,R0	Before execution	T = 1, R0 = H'00000000, R2 = H'00000000
		After execution	T = 0, R0 = H'00000001

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

Format	Abstract	Code	Cycle	T Bit
ADDV Rm,Rn	Rn + Rm → Rn, overflow → T	0011nnnnmmmm1111	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

6.5 AND (AND Logical): Logic Operation Instruction

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

6.6 BF (Branch if False): Branch Instruction

Format	Abstract	Code	Cycle	T Bit
BF label	When T = 0, disp + PC → PC; When T = 1, nop	10001011dddddddd	3/1	—

Description: Reads the T bit, and conditionally branches. If T = 1, BF executes the next instruction. If T = 0, it branches. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch 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=(0xFFFFFF00 | (long)d);
    if (T==0) PC=PC+(disp<<1)+4;
    else PC+=2;
}
```

Example:

```
        CLRT                T is always cleared to 0
        BT      TRGET_T     Does not branch, because T = 0
        BF      TRGET_F     Branches to TRGET_F, because T = 0
        NOP
        NOP                 ← The PC location is used to calculate the branch destination
                            address of the BF instruction
TRGET_F:        ← Branch destination of the BF instruction
```

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

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
`BF label`	When T = 0, disp + PC → PC; When T = 1, nop	`10001111dddddddd`	2/1	—

Description: Reads the T bit, and if T = 1, BF executes the next instruction. If T = 0, it branches after executing the next instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after the branch 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: The BF/S instruction is a conditional delayed branch instruction:

1. **Taken case:** The instruction immediately following is executed before the branch. Between the time this instruction and the instruction immediately following are executed, no interrupts are accepted. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction.

2. **Not taken case:** This instruction operates as a nop instruction. Between the time this instruction and the instruction immediately following are executed, interrupts are accepted. When the instruction immediately following is a branch instruction, it is not recognized as an illegal slot instruction.

Operation:

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

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==0) {
        PC=PC+(disp<<1)+4;
        Delay_Slot(temp+2);
    }
    else PC+=2;
}
```

Examples:

	SETT	T is always 1
	BF/S TARGET_F	Does not branch, because T = 1
	NOP	
	BT/S TARGET_T	Branches to TARGET, because T = 1
	ADD R0,R1	Executed before branch.
	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

6.8 BRA (Branch): Branch Instruction

Class: Delayed branch instruction

Format		Abstract	Code	Cycle	T Bit
BRA	label	disp + PC → PC	1010ddddddddddd	2	—

Description: Branches unconditionally after executing the instruction following this BRA instruction. The branch destination is an address specified by PC + displacement. The PC points to the starting address of the second instruction after this BRA 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 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 & d);
    else disp=(0xFFFFF000 | d);
    temp=PC;
    PC=PC+(disp<<1)+4;
    Delay_Slot(temp+2);
}
```

Examples:

```
        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
```

6.9 BRAF (Branch Far): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
BRAF Rn	Rn + PC → PC	0000nnnn00100011	2	—

Description: Branches unconditionally. The branch destination is PC + the 32-bit contents of the general register Rn. PC is the start address of the second instruction 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 n)   /* BRAF Rn */
{
    unsigned long temp;

    temp=PC;
    PC+=R[n];
    Delay_Slot(temp+2);
}
```

Examples:

```
        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
```

6.10 BSR (Branch to Subroutine): Branch Instruction

Class: Delayed branch instruction

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

Description: Branches to the subroutine procedure at a specified address after executing the instruction following this BSR instruction. The PC value is stored in the PR, and the program branches to an address specified by PC + displacement. The PC points to the starting address of the second instruction after this BSR 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 for a subroutine procedure call.

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

Operation:

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

    if ((d&0x800)==0) disp=(0x00000FFF & d);
    else disp=(0xFFFFF000 | d);
    PR=PC;
    PC=PC+(disp<<1)+4;
    Delay_Slot(PR+2);
}
```

Examples:

```
        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
```

6.11 BSRF (Branch to Subroutine Far): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
BSRF Rn	PC → PR, Rn + PC → PC	0000nnnn00000011	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 Rn. PC is the start address of the second instruction after this instruction. Used as a subroutine call in combination with RTS.

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

Operation:

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

Examples:

```
        MOV.L   #(TARGET-BSRF_PC),R0     Sets displacement.
        BRSF    @R0                      Branches to TARGET
        MOV     R3,R4                    Executes the MOV instruction before
                                         branching
BSRF_PC:                                 ← The PC location is used to calculate the
                                         branch destination with BSRF.
        ADD     R0,R1
        .....
        .....
TARGET:                                  ←Procedure entrance
        MOV     R2,R3
        RTS                              Returns to the above ADD instruction
        MOV     #1,R0                    Executes MOV before branching
```

6.12 BT (Branch if True): Branch Instruction

Format		Abstract	Code	Cycle	T Bit
BT	label	When T = 1, disp + PC → PC; When T = 0, nop	10001001dddddddd	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. The PC points to the starting address of the second instruction after the branch 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=(0xFFFFFF00 | (long)d);
    if (T==1) PC=PC+(disp<<1)+4;
    else PC+=2;
}
```

Examples:

```
        SETT            T is always 1
        BF    TRGET_F   Does not branch, because T = 1
        BT    TRGET_T   Branches to TRGET_T, because T = 1
        NOP
        NOP             ← The PC location is used to calculate the branch destination
                          address of the BT instruction
TRGET_T:        ← Branch destination of the BT instruction
```

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

Format		Abstract	Code	Cycle	T Bit
BT/S	label	When T = 1, disp + PC → PC; When T = 0, nop	10001101dddddddd	2/1	—

Description: Reads the T bit, and 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. The PC points to the starting address of the second instruction after the branch 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: The BF/S instruction is a conditional delayed branch instruction:

1. **Taken case:** The instruction immediately following is executed before the branch. Between the time this instruction and the instruction immediately following are executed, no interrupts are accepted. When the instruction immediately following is a branch instruction, it is recognized as an illegal slot instruction.

2. **Not taken case:** This instruction operates as a nop instruction. Between the time this instruction and the instruction immediately following are executed, interrupts are accepted. When the instruction immediately following is a branch instruction, it is not recognized as an illegal slot instruction.

Operation:

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

    temp=PC;
    if ((d&0x80)==0) disp=(0x000000FF & (long)d);
    else disp=(0xFFFFFF00 | (long)d);
    if (T==1) {
        PC=PC+(disp<<1)+4;
        Delay_Slot(temp+2);
    }
    else PC+=2;
}
```

Examples:

```
        SETT              T is always 1
        BF/S  TARGET_F    Does not branch, because T = 1
        NOP
        BT/S  TARGET_T    Branches to TARGET, because T = 1
        ADD   R0,R1       Executes before branching.
        NOP               ← 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
```

6.14 CLRMAC (Clear MAC Register): System Control Instruction

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

Description: Clears the MACH and MACL registers.

Operation:

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

Examples:

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

6.15 CLRS (Clear S Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRS	0 → S	0000000001001000	1	—

Description: Clears the S bit.

Operation:

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

Examples:

CLRS	Before execution S=1
	After execution S=0

6.16 CLRT (Clear T Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
CLRT	0 →T	0000000000001000	1	0

Description: Clears the T bit.

Operation:

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

Examples:

CLRT	Before execution	T = 1
	After execution	T = 0

6.17 CMP/cond (Compare Conditionally): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
CMP/EQ	Rm,Rn	When Rn = Rm, 1 → T	0011nnnnmmmm0000	1	Comparison result
CMP/GE	Rm,Rn	When signed and Rn ≥ Rm, 1 → T	0011nnnnmmmm0011	1	Comparison result
CMP/GT	Rm,Rn	When signed and Rn > Rm, 1 → T	0011nnnnmmmm0111	1	Comparison result
CMP/HI	Rm,Rn	When unsigned and Rn > Rm, 1 → T	0011nnnnmmmm0110	1	Comparison result
CMP/HS	Rm,Rn	When unsigned and Rn ≥ Rm, 1 → T	0011nnnnmmmm0010	1	Comparison result
CMP/PL	Rn	When Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	When Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
CMP/STR	Rm,Rn	When a byte in Rn equals a byte in Rm, 1 → T	0010nnnnmmmm1100	1	Comparison result
CMP/EQ	#imm,R0	When R0 = imm, 1 → T	10001000iiiiiiii	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, and the Rn data does not change. The nine conditions in table 6.1 can be specified. Conditions PZ and PL are the results of comparisons between Rn and 0. Sign-extended 8-bit immediate data can also be compared with R0 by using condition EQ. Here, R0 data does not change. Table 6.1 shows the mnemonics for the conditions.

Table 6.1 CMP Mnemonics

Mnemonics		Condition
CMP/EQ	Rm,Rn	If Rn = Rm, T = 1
CMP/GE	Rm,Rn	If Rn ≥ 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 ≥ Rm with unsigned data, T = 1
CMP/PL	Rn	If Rn > 0, T = 1
CMP/PZ	Rn	If Rn ≥ 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&0xFF000000)>>12;
    HL=(temp&0x00FF0000)>>8;
    LH=(temp&0x0000FF00)>>4; LL=temp&0x000000FF;
    HH=HH&&HL&&LH&&LL;
    if (HH==0) T=1;
    else T=0;
    PC+=2;
}
```

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

    if ((i&0x80)==0) imm=(0x000000FF & (long i));
    else imm=(0xFFFFFF00 | (long i));
    if (R[0]==imm) T=1;
    else T=0;
    PC+=2;
}
```

Examples:

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

6.18 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;
}
```

Examples: See DIV1.

6.19 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.

6.20 DIV1 (Divide Step 1): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
DIV1	Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	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). Zero division, overflow detection, and remainder operation are not supported. Check for zero division and overflow division before dividing.

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

Operation:

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

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

```
    case 1:switch(M){
        case 0:tmp0=R[n];
            R[n]+=R[m];
            tmp1=(R[n]<tmp0);
            switch(Q){
            case 0:Q=tmp1;
                break;
            case 1:Q=(unsigned char)(tmp1==0);
                break;
            }
            break;
        case 1:tmp0=R[n];
            R[n]-=R[m];
            tmp1=(R[n]>tmp0);
            switch(Q){
            case 0:Q=(unsigned char)(tmp1==0);
                break;
        case 1:Q=tmp1;
                break;
            }
            break;
        }
        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,R2         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
`.arepeat`	`16`	
`DIV1`	`R0,R1`	Repeat 16 times
`.aendr`		
`EXTS.W`	`R1,R1`	
`ROTCL`	`R1`	R1 = quotient (ones complement)
`ADDC`	`R2,R1`	Increments and takes the twos 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 ones complement if the dividend is negative
`DIV0S`	`R0,R1`	Flag initialization
`.arepeat`	`32`	
`ROTCL`	`R2`	Repeat 32 times
`DIV1`	`R0,R1`	
`.aendr`		
`ROTCL`	`R2`	R2 = Quotient (one's complement)
`ADDC`	`R3,R2`	Increments and takes the two's complement if the MSB of the quotient is 1. R2 = Quotient (two's complement)

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

Format	Abstract	Code	Cycle	T Bit
`DMULS.L Rm,Rn`	With sign, Rn × Rm → MACH, MACL	`0011nnnnmmmm1101`	2 (to 5)	—

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;
}
```

Examples:

DMULS	R0,R1	Before execution	R0 = H'FFFFFFFE, R1 = H'00005555
		After execution	MACH = H'FFFFFFFF, MACL = H'FFFF5556
STS	MACH,R0	Operation result (top)	
STS	MACL,R0	Operation result (bottom)	

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

Format	Abstract	Code	Cycle	T Bit
`DMULU.L Rm,Rn`	Without sign, Rn × Rm → MACH, MACL	`0011nnnnmmmm0101`	2 (to 5)	—

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;
}
```

Examples:

DMULU	R0,R1	Before execution	R0 = H'FFFFFFFE, R1 = H'00005555
		After execution	MACH = H'FFFFFFFF, MACL = H'FFFF5556
STS	MACH,R0	Operation result (top)	
STS	MACL,R0	Operation result (bottom)	

6.23 DT (Decrement and Test): Arithmetic Instruction

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

Description: Decrements the contents of general register Rn by 1 and compares the results 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 is T=0. (In this example, loops 4 times.)
```

6.24 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]|=0xFFFFFF00;
    PC+=2;
}

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

Examples:

```
EXTS.B   R0,R1  Before execution      R0 = H'00000080
                After execution       R1 = H'FFFFFF80
EXTS.W   R0,R1  Before execution      R0 = H'00008000
                After execution       R1 = H'FFFF8000
```

6.25 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

6.26 JMP (Jump): Branch Instruction

Class: Delayed branch instruction

Format		Abstract	Code	Cycle	T Bit
JMP	@Rn	Rn → PC	0100nnnn00101011	2	—

Description: Branches unconditionally after executing the instruction following this JMP instruction. The branch destination is an address specified by the 32-bit data in general register Rn.

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

Operation:

```
JMP(long n)    /* JMP @Rn */
{
    unsigned long temp;

    temp=PC;
    PC=R[n]+4;
    Delay_Slot(temp+2);
}
```

Examples:

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

6.27 JSR (Jump to Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
JSR @Rn	PC → Rn, Rn → PC	0100nnnn00001011	2	—

Description: Branches to the subroutine procedure at a specified address after executing the instruction following this JSR instruction. The PC value is stored in the PR. The jump destination is an address specified by the 32-bit data in general register Rn. The PC points to the starting address of the second instruction after JSR. The JSR instruction and RTS instruction are used for subroutine procedure calls.

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

Operation:

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

Examples:

```
            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
```

6.28 LDC (Load to Control Register): System Control Instruction (Privileged Only)

Format		Abstract	Code	Cycle	T Bit
LDC	Rm,SR	Rm → SR	0100mmmm00001110	5	LSB
LDC	Rm,GBR	Rm → GBR	0100mmmm00011110	1	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—
LDC.L	@Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	7	LSB
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
LDC.L	@Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC.L	@Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC.L	@Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC.L	@Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC.L	@Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC.L	@Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—
LDC.L	@Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC.L	@Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC.L	@Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC.L	@Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	1	—

Description: Stores source operand into control registers SR, GBR, VBR, SSR, SPC, or R0_BANK to R7_BANK. LDC and LDC.L, except for LDC GBR, Rn and LDC.L GBR, @-Rn are privileged instructions and can be used in privileged mode only. If used in user mode, they cause illegal instruction exceptions. LDC GBR, Rn and LDC.L GBR, @-Rn can be used in user mode.
Rn_BANK is designated by the RB bit of the SR. When the RB = 1, Rn_BANK0 is accessed by LDC/LDC.L instructions. When the RB = 0, Rn_BANK1 is accessed by LDC/LDC.L instructions.

Operation:

```
LDCSR(long m)     /* LDC Rm,SR */
{
    SR=R[m]&0x700003F3;
    PC+=2;
}

LDCGBR(long m)    /* LDC Rm,GBR */
{
    GBR=R[m];
    PC+=2;
}

LDCVBR(long m)    /* LDC Rm,VBR */
{
    VBR=R[m];
    PC+=2;
}

LDCSSR(long m)        /* LDC Rm,SSR */
{
    SSR=R[m]&0x700003F3;
    PC+=2;
}

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

```
LDCRn_BANK(long m)    /* LDC Rm,Rn_BANK */
{                     /* n=0-7, */
    Rn_BANK=R[m];
    PC+=2;
}


LDCMSR(long m)    /* LDC.L @Rm+,SR */
{
    SR=Read_Long(R[m])&0x700003F3;
    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;
}

LDCMSSR(long m)   /* LDC.L @Rm+,SSR */
{
    SSR=Read_Long(R[m])&0x700003F3;
    R[m]+=4;
    PC+=2;
}
```

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

LDCMRn_BANK(long m)   /* LDC.L @Rm+,Rn_BANK */
                      /* n=0-7 */
{
    Rn_BANK=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'700003F3
`LDC.L`	`@R15+,GBR`	Before execution	R15 = H'10000000
		After execution	R15 = H'10000004, GBR = @H'10000000

6.29 LDS (Load to System Register): System Control 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: Stores the source operand into the system registers MACH, MACL, or PR.

Operation:

```
LDSMACH(long m)          /* LDS Rm,MACH */
{
    MACH=R[m];
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    PC+=2;
}

LDSMACL(long m)          /* LDS Rm,MACL */
{
    MACL=R[m];
    PC+=2;
}

LDSPR(long m)            /* LDS Rm,PR */
{
    PR=R[m];
    PC+=2;
}
```

```
LDSMMACH(long m)         /* LDS.L @Rm+,MACH */
{
    MACH=Read_Long(R[m]);
    if ((MACH&0x00000200)==0) MACH&=0x000003FF;
    else MACH|=0xFFFFFC00;
    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

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

Format	Abstract	Code	Cycle	T Bit
LDTLB	PTEH/PTEL → TLB	0000000000111000	1	—

Description: Loads PTEH/PTEL registers to the translation lookaside buffer (TLB). The TLB is indexed by the virtual address held in the PTEH register. The loaded set is designated by the MMUCR.RC (MMUCR is an MMU control register and RC is a two bit field for a counter). LDTLB is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: As LDTLB is for loading PTEH and PTEL to the TLB, the instruction should be issued when MMU is off (MMUCR.AT = 0) or should be placed in the P1 or P2 space with MMU enabled (see section 3, MMU, of the *SH7700 Series Hardware Manual*). If the instruction is issued in an exception handler, it should be at least two instructions prior to an RTE instruction that terminates the handler.

Operation:

```
LDTLB()    /*LDTLB*/
{
    TLB_tag=PTEH;
    TLB_data=PTEL;
    PC+=2;
}
```

Examples:

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

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

Format	Abstract	Code	Cycle	T Bit
`MAC.L @Rm+,@Rn+`	Signed operation, (Rn) × (Rm) + MAC → MAC	`0000nnnnmmmm1111`	2 (to 5)	—

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

When the S bit is 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 between H'FFFF800000000000 (minimum) and H'00007FFFFFFFFFFF (maximum).

Operation:

```
MACL(long m,long n)   /* MAC.L @Rm+,@Rn+*/
{
    unsigned long RnL,RnH,RmL,RmH,Res0,Res1,Res2;
    unsigned long temp0,templ,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;
}
```

Examples:

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

6.32 MAC (Multiply and Accumulate): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
`MAC.W @Rm+,@Rn+`	With sign, (Rn) × (Rm) + MAC → MAC	`0100nnnnmmmm1111`	2 (to 5)	—

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

When the S bit is cleared to 0, the 42-bit result is stored in the coupled MACH and MACL registers. Bit 9 data is copied to the upper 22 bits (bits 31 to 10) of the MACH register. Rm and Rn data are incremented by 2 after the operation.

When the bit S is set to 1, addition to the MAC register is a saturation operation. For the saturation operation, only the MACL register is enabled and the result is limited to between H'80000000 (minimum) and H'7FFFFFFF (maximum).

If an overflow occurs, the LSB of the MACH register is set to 1. The result is stored in the MACL register, and 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.

Note: The normal number of cycles for execution is 3; however, succeeding instructions can be executed in two cycles.

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+=tempm;
    if ((long)MACL>=0) ans=0;
    else ans=1;
    ans+=dest;
    if (S==1) {
        if (ans==1) {
            if (src==0 || src==2) MACH|=0x00000001;
            if (src==0) MACL=0x7FFFFFFF;
            if (src==2) MACL=0x80000000;
        }
    }
    else {
        MACH+=tempn;
        if (templ>MACL) MACH+=1;
        if ((MACH&0x00000200)==0) MACH&=0x000003FF;
        else MACH|=0xFFFFFC00;
    }
    PC+=2;
}
```

Examples:

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

6.33 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]|=0xFFFFFF00;
    PC+=2;
}

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

MOVLL(long m,long n)     /* MOV.L @Rm,Rn */
{
    R[n]=Read_Long(R[m]);
    PC+=2;
}
```

```
MOVBM(long m,long n)     /* MOV.B Rm,@-Rn */
{
    Write_Byte(R[n]-1,R[m]);
    R[n]-=1;
    PC+=2;
}

MOVWM(long m,long n)     /* MOV.W Rm,@-Rn */
{
    Write_Word(R[n]-2,R[m]);
    R[n]-=2;
    PC+=2;
}

MOVLM(long m,long n)     /* MOV.L Rm,@-Rn */
{
    Write_Long(R[n]-4,R[m]);
    R[n]-=4;
    PC+=2;
}

MOVBP(long m,long n)  /* MOV.B @Rm+,Rn */
{
    R[n]=(long)Read_Byte(R[m]);
    if ((R[n]&0x80)==0) R[n]&0x000000FF;
    else R[n]|=0xFFFFFF00;
    if (n!=m) R[m]+=1;
    PC+=2;
}

MOVWP(long m,long n)     /* MOV.W @Rm+,Rn */
{
    R[n]=(long)Read_Word(R[m]);
    if ((R[n]&0x8000)==0) R[n]&0x0000FFFF;
    else R[n]|=0xFFFF0000;
    if (n!=m) R[m]+=2;
    PC+=2;
}
```

```
MOVLP(long m,long n)     /* MOV.L @Rm+,Rn */
{
    R[n]=Read_Long(R[m]);
    if (n!=m) R[m]+=4;
    PC+=2;
}

MOVBS0(long m,long n)      /* MOV.B Rm,@(R0,Rn) */
{
    Write_Byte(R[n]+R[0],R[m]);
    PC+=2;
}

MOVWS0(long m,long n)      /* MOV.W Rm,@(R0,Rn) */
{
    Write_Word(R[n]+R[0],R[m]);
    PC+=2;
}

MOVLS0(long m,long n)    /* MOV.L Rm,@(R0,Rn) */
{
    Write_Long(R[n]+R[0],R[m]);
    PC+=2;
}

MOVBL0(long m,long n)    /* MOV.B @(R0,Rm),Rn */
{
    R[n]=(long)Read_Byte(R[m]+R[0]);
    if ((R[n]&0x80)==0) R[n]&0x000000FF;
    else R[n]|=0xFFFFFF00;
    PC+=2;
}

MOVWL0(long m,long n)    /* MOV.W @(R0,Rm),Rn */
{
    R[n]=(long)Read_Word(R[m]+R[0]);
    if ((R[n]&0x8000)==0) R[n]&0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}
```

```
MOVLL0(long m,long n)     /* MOV.L @(R0,Rm),Rn */
{
    R[n]=Read_Long(R[m]+R[0]);
    PC+=2;
}
```

Examples:

`MOV`	`R0,R1`	Before execution	R0 = H'FFFFFFFF, R1 = H'00000000
		After execution	R1 = H'FFFFFFFF
`MOV.W`	`R0,@R1`	Before execution	R0 = H'FFFF7F80
		After execution	@R1 = H'7F80
`MOV.B`	`@R0,R1`	Before execution	@R0 = H'80, R1 = H'00000000
		After execution	R1 = H'FFFFFF80
`MOV.W`	`R0,@-R1`	Before execution	R0 = H'AAAAAAAA, R1 = H'FFFF7F80
		After execution	R1 = H'FFFF7F7E, @R1 = H'AAAA
`MOV.L`	`@R0+,R1`	Before execution	R0 = H'12345670
		After execution	R0 = H'12345674, R1 = @H'12345670
`MOV.B`	`R1,@(R0,R2)`	Before execution	R2 = H'00000004, R0 = H'10000000
		After execution	R1 = @H'10000004
`MOV.W`	`@(R0,R2),R1`	Before execution	R2 = H'00000004, R0 = H'10000000
		After execution	R1 = @H'10000004

6.34 MOV (Move Immediate Data): Data Transfer Instruction

Format		Abstract	Code	Cycle	T Bit
MOV	#imm,Rn	#imm → sign extension → Rn	1110nnnniiiiiiii	1	—
MOV.W	@(disp,PC),Rn	(disp + PC) → sign extension → Rn	1001nnnnddddddd	1	—
MOV.L	@(disp,PC),Rn	(disp + 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 is 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 is 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 end address of the program area (module) or the second address after an unconditional branch instruction are suitable for the start address of the table. If suitable table assignment is impossible (for example, if there are no unconditional branch instructions within the area specified by PC + 510 bytes or PC + 1020 bytes), the BRA instruction must be used to jump past the table. When this MOV instruction is placed immediately after a delayed branch instruction, the PC points to an address specified by (the starting address of the branch destination) + 2.

Operation:

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

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

    disp=(0x000000FF & (long)d);
    R[n]=(long)Read_Word(PC+(disp<<1));
    if ((R[n]&0x8000)==0) R[n]&=0x0000FFFF;
    else R[n]|=0xFFFF0000;
    PC+=2;
}
```

```
MOVLI(long d,long n)     /* MOV.L @(disp,PC),Rn */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[n]=Read_Long((PC&0xFFFFFFFC)+(disp<<2));
    PC+=2;
}
```

Examples:

Address				
1000		MOV	#H'80,R1	R1 = H'FFFFFF80
1002		MOV.W	IMM,R2	R2 = H'FFFF9ABC, IMM means @(H'08,PC)
1004		ADD	#–1,R0	
1006		TST	R0,R0	← PC location used for address calculation for the MOV.W instruction
1008		MOVT	R13	
100A		BRA	NEXT	Delayed branch instruction
100C		MOV.L	@(4,PC),R3	R3 = H'12345678
100E	IMM	*.data.w*	H'9ABC	
1010		*.data.w*	H'1234	
1012	NEXT	JMP	@R3	Branch destination of the BRA instruction
1014		CMP/EQ	#0,R0	← PC location used for address calculation for the MOV.L instruction
		.align	4	
1018		*.data.l*	H'12345678	

6.35 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 + GBR) → sign extension → R0	11000101dddddddd	1	—
MOV.L	@(disp,GBR),R0	(disp + GBR) → R0	11000110dddddddd	1	—
MOV.B	R0,@(disp,GBR)	R0 → (disp + GBR)	11000000dddddddd	1	—
MOV.W	R0,@(disp,GBR)	R0 → (disp + GBR)	11000001dddddddd	1	—
MOV.L	R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	1	—

Description: Transfers the source operand to the destination. This instruction is suitable for accessing data in the peripheral module area. The data can be a byte, word, or longword, but only the R0 register can be used.

A peripheral module base address is set to the GBR. When the peripheral module data is a byte, the only change made is to zero-extend the 8-bit displacement. Consequently, an address within +255 bytes can be specified. When the peripheral module data is a word, the 8-bit displacement is zero-extended and doubled. Consequently, an address within +510 bytes can be specified. When the peripheral module data is a longword, the 8-bit displacement is zero-extended and is quadrupled. Consequently, an address within +1020 bytes can be specified. If the displacement is too short to reach the memory operand, the above @(R0,Rn) mode must be used after the GBR data is transferred to a general register. When the source operand is in memory, the loaded data is stored in the register after it is sign-extended to a longword.

Note: The destination register of a data load is always R0. R0 cannot be accessed by the next instruction until the load instruction is finished. The instruction order shown in figure 6.1 will give better results.

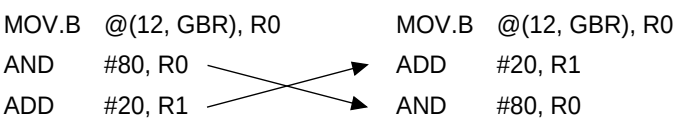


Figure 6.1 Using R0 after MOV

Operation:

```
MOVBLG(long d)    /* MOV.B @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Byte(GBR+disp);
    if ((R[0]&0x80)==0) R[0]&=0x000000FF;
    else R[0]|=0xFFFFFF00;
    PC+=2;
}

MOVWLG(long d)    /* MOV.W @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=(long)Read_Word(GBR+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}

MOVLLG(long d)    /* MOV.L @(disp,GBR),R0 */
{
    long disp;

    disp=(0x000000FF & (long)d);
    R[0]=Read_Long(GBR+(disp<<2));
    PC+=2;
}
```

```
MOVBSG(long d)    /* MOV.B R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Byte(GBR+disp,R[0]);
    PC+=2;
}
```

```
MOVWSG(long d)    /* MOV.W R0,@(disp,GBR) */
{
    long disp;

    disp=(0x000000FF & (long)d);
    Write_Word(GBR+(disp<<1),R[0]);
    PC+=2;
}
```

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

    disp=(0x000000FF & (long)d);
    Write_Long(GBR+(disp<<2),R[0]);
    PC+=2;
}
```

Examples:

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

6.36 MOV (Move Structure Data): Data Transfer Instruction

Format		Abstract	Code	Cycle	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	—

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

Note: When byte or word data is loaded, the destination register is always R0. R0 cannot be accessed by the next instruction until the load instruction is finished. The instruction order in figure 6.2 will give better results.

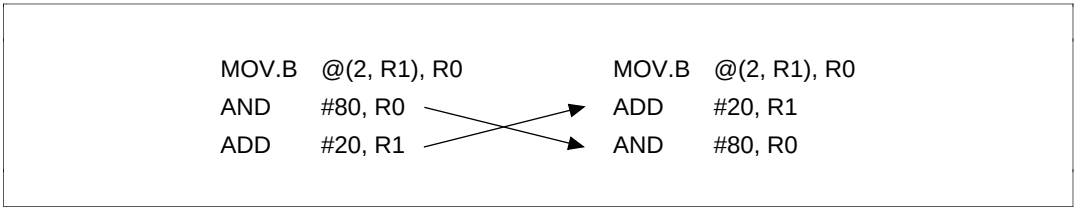


Figure 6.2 Using R0 after MOV

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]|=0xFFFFFF00;
    PC+=2;
}
```

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

    disp=(0x0000000F & (long)d);
    R[0]=Read_Word(R[m]+(disp<<1));
    if ((R[0]&0x8000)==0) R[0]&=0x0000FFFF;
    else R[0]|=0xFFFF0000;
    PC+=2;
}
```

```
MOVLL4(long m,long d,long n)
    /* MOV.L @(disp,Rm),Rn */
{
    long disp;

    disp=(0x0000000F & (long)d);
    R[n]=Read_Long(R[m]+(disp<<2));
    PC+=2;
}
```

Examples:

`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

6.37 MOVA (Move Effective Address): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
`MOVA @(disp,PC),R0`	disp + PC → R0	11000111dddddddd	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 points to the starting address of the second instruction after this MOVA 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&0xFFFFFFFC)+(disp<<2);
    PC+=2;
}
```

Examples:

```
Address          .org     H'1006
1006             MOVA     STR,R0        Address of STR → R0
1008             MOV.B    @R0,R1        R1 = "X"   ← PC location after correcting the lowest
                                        two bits
100A             ADD      R4,R5         ← Original PC location for address calculation for the
                                        MOVA instruction
                 .align   4
100C     STR:    .sdata   "XYZP12"
...............
2002             BRA      TRGET         Delayed branch instruction
2004             MOVA     @(0,PC),R0    Address of TRGET + 2 → R0
2006             NOP
```

6.38 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;
}
```

Examples:

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

6.39 MUL.L (Multiply Long): Arithmetic Instruction

Format	Abstract	Code	Cycle	T Bit
`MUL.L Rm,Rn`	Rn × Rm → MACL	`0000nnnnmmmm0111`	2 (to 5)	—

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

Examples:

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

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

Format		Abstract	Code	Cycle	T Bit
MULS.W	Rm,Rn	Signed operation, Rn × Rm → MACL	0010nnnnmmmm1111	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;
}
```

Examples:

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

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

Format		Abstract	Code	Cycle	T Bit
MULU.W	Rm,Rn	Unsigned, Rn × Rm → MAC	0010nnnnmmmm1110	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;
}
```

Examples:

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

6.42 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;
}
```

Examples:

NEG	R0,R1	Before execution	R0 = H'00000001
		After execution	R1 = H'FFFFFFFF

6.43 NEGC (Negate with Carry): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
NEGC	Rm,Rn	0 – Rm – T → Rn, Borrow → 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'FFFFFFFF, T = 1
NEGC	R0,R0	Before execution	R0 = H'00000000, T = 1
		After execution	R0 = H'FFFFFFFF, T = 1

6.44 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;
}
```

Examples:

```
NOP      Executes in one cycle
```

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

Format	Abstract	Code	Cycle	T Bit
NOT Rm,Rn	~Rm → 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;
}
```

Examples:

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

6.46 OR (OR Logical) Logic Operation Instruction

Format		Abstract	Code	Cycle	T Bit
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
OR.B	#imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	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

6.47 PREF (Prefetch Data to the Cache)

Format	Abstract	Code	Cycle	T Bit
PREF @Rn	(Rn &0xfffffff0) → Cache (Rn &0xfffffff0+4) → Cache (Rn &0xfffffff0+8) → Cache (Rn &0xfffffff0+C) → Cache	0000nnnn10000011	1	—

Description: Loads data to cache on software prefetching. 16-byte data containing the data pointed by Rn (Cache 1 line) is loaded to the cache. Address Rn should be on longword boundary.

No address related error is detected in this instruction. In case of an error, the instruction operates as NOP.

The destination is on-chip cache, therefore this instruction functions as an NOP instruction in effect, that is, it never changes registers or processor status.

Operation:

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

Examples:

```
MOV.L SOFT_PF,R1          Address of R1 is SOFT_PF
PREF @R1                  Load data from SOFT_PF to on-chip cache

.align 4

SOFT_PF:   .data.1   H'12345678
           .data.1   H'9ABCDEF0
           .data.1   H'AAAA5555
           .data.1   H'5555AAAA
```

6.48 ROTCL (Rotate with Carry Left): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTCL Rn	T ← Rn ← 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 6.3).

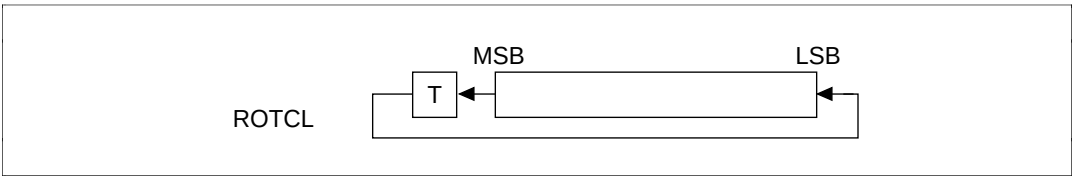


Figure 6.3 Rotate with Carry Left

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;
}
```

Examples:

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

6.49 ROTCR (Rotate with Carry Right): Shift Instruction

Format	Abstract	Code	Cycle	T Bit
ROTCR Rn	T → Rn → 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 6.4).

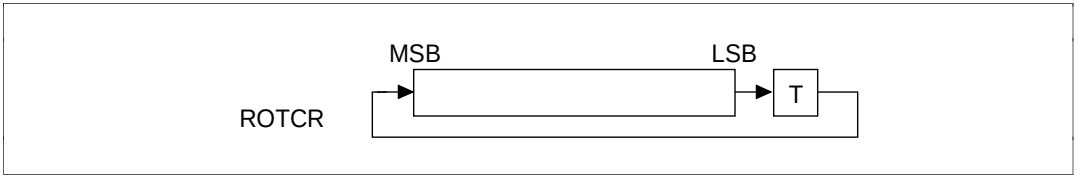


Figure 6.4 Rotate with Carry Right

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

6.50 ROTL (Rotate Left): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
ROTL	Rn	T ← Rn ← 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 6.5). The bit that is shifted out of the operand is transferred to the T bit.

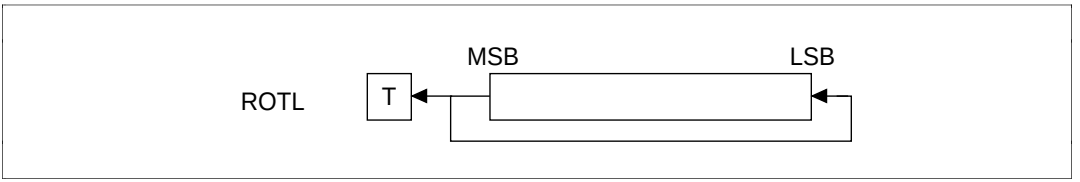


Figure 6.5 Rotate Left

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

6.51 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 6.6). The bit that is shifted out of the operand is transferred to the T bit.

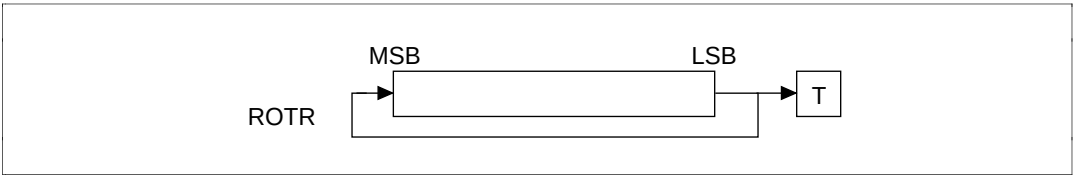


Figure 6.6 Rotate Right

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

6.52 RTE (Return from Exception): System Control Instruction (Privileged Only)

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
RTE	SSR → SR, SPC → PC	0000000000101011	4	—

Description: Returns from an exception routine. The PC and SR values are loaded from SPC and SSR. The program continues from the address specified by the loaded PC value. RTE is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: Since this is a delayed branch instruction, the instruction after this RTE is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. The SR accessed by an instruction in the delay slot of an RTE has been restored from the SSR by the RTE.

Operation:

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

    temp=PC;
    PC=SPC;
    SR=SSR;
    Delay_Slot(temp+2);
}
```

Examples:

RTE		Returns to the original routine
ADD	#8,R15	Executes ADD before branching

6.53 RTS (Return from Subroutine): Branch Instruction

Class: Delayed branch instruction

Format	Abstract	Code	Cycle	T Bit
RTS	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 or JSR instruction.

Note: Since this is a delayed branch instruction, the instruction after this RTS is executed before branching. No interrupts are accepted between this instruction and the next instruction. If the next instruction is a branch instruction, it is acknowledged as an illegal slot instruction. An instruction restoring the PR should be prior to an RTS instruction. That restoring instruction should not be the delay slot of the RTS.

Operation:

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

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

Examples:

```
        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
```

6.54 SETS (Set S Bit): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
SETS	1 → S	0000000001011000	1	—

Description: Sets the S bit to 1.

Operation:

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

Examples:

SETS	Before execution	S = 0
	After execution	S = 1

6.55 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;
}
```

Examples:

SETT	Before execution	T = 0
	After execution	T = 1

6.56 SHAD (Shift Arithmetic Dynamically): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHAD	Rm,Rn	Rn << Rm → Rn (Rm ≥ 0) Rn >> Rm → Rn (Rm < 0)	0100nnnnmmmm1100	2	—

Description: Arithmetically shifts the contents of general register Rn. General register Rm indicates the shift direction and shift count (figure 6.7).

- Shift direction: Rm ≥ 0, left
  Rm < 0, right
- Shift count: Rm (4–0) are used; if negative, two's complement is set to Rm.
  The maximum magnitude of the left shift count is 31 (0–31).
  The maximum magnitude of the right shift count is 32 (1–32).

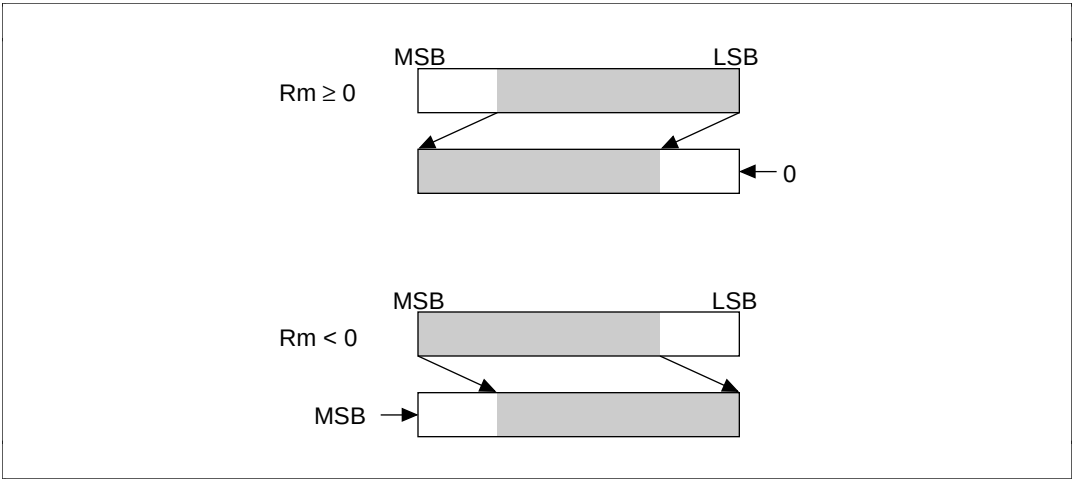


Figure 6.7 Shift Arithmetic Dynamically

Operation:

```
SHAD(long m,n)    /* SHAD Rm,Rn */
{
    long cont, sgn;
    sgn = R[m] &0x80000000;
    cnt = R[m] &0x0000001F;
    if (sgn==0)    R[n]<<=cnt;
    else           R[n]=(signed long)R[n]>>((~cnt+1) & 0x1F); /*shift
                   arithmetic right*/
    PC+=2;
}
```

Examples:

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

6.57 SHAL (Shift Arithmetic Left): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHAL	Rn	T ← Rn ← 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 6.8).

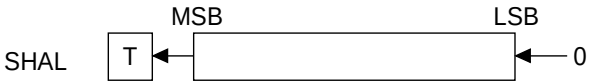


Figure 6.8 Shift Arithmetic Left

Operation:

```
SHAL(long n)  /* SHAL Rn  (Same as SHLL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

Examples:

SHAL	R0	Before execution	R0 = H'80000001, T = 0
		After execution	R0 = H'00000002, T = 1

6.58 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 6.9).

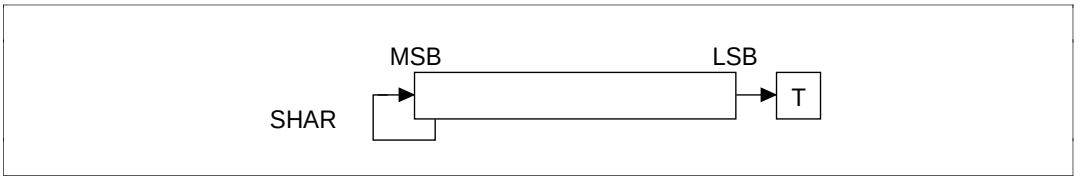


Figure 6.9 Shift Arithmetic Right

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;
}
```

Examples:

SHAR	R0	Before execution	R0 = H'80000001, T = 0
		After execution	R0 = H'C0000000, T = 1

6.59 SHLD (Shift Logical Dynamically): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHLD	Rm,Rn	Rn << Rm → Rn (Rm ≥ 0) Rn >> Rm → Rn (Rm < 0)	0100nnnnmmmm1101	2	—

Description: Arithmetically shifts the contents of general register Rn. General register Rm indicates the shift direction and shift count (figure 6.10). T bit is the last shifted bit of Rn.

- Shift direction: Rm ≥ 0, left
  Rm < 0, right
- Shift count: Rm (4–0) are used; if negative, two's complement is set to Rm.
  The maximum magnitude of the left shift count is 31 (0–31).
  The maximum magnitude of the right shift count is 32 (1–32).

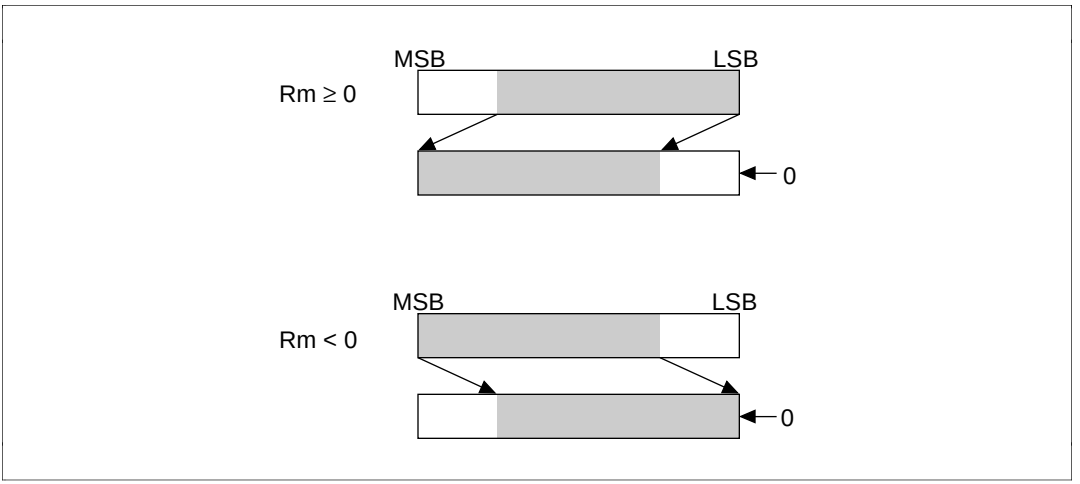


Figure 6.10 Shift Logical Dynamically

Operation:

```
    SHLD(long m,n)    /* SHLD Rm,Rn */
{
    long cont, sgn;

    sgn = R[m]&0x80000000;
    cnt = R[m]&0x0000001F);
    if (sgn==0)    R[n]<<=cnt;
    else           R[n]=R[n]>>((~cnt+1)&0x1F);
    PC+=2;
}
```

Examples:

SHLD	R1,R2	Before execution	R1 = H'FFFFFFEC, R2 = H'80180000
		After execution	R1 = H'FFFFFFEC, R2 = H'00000801

6.60 SHLL (Shift Logical Left): Shift Instruction

Format		Abstract	Code	Cycle	T Bit
SHLL	Rn	T ← Rn ← 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 6.11).

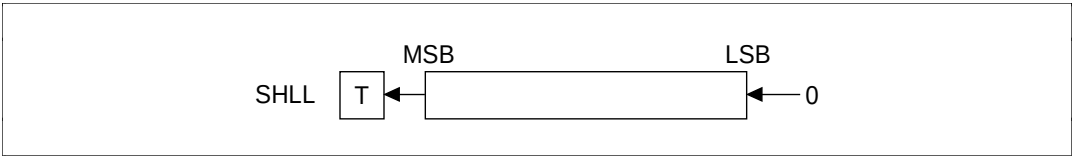


Figure 6.11 Shift Logical Left

Operation:

```
SHLL(long n)  /* SHLL Rn  (Same as SHAL) */
{
    if ((R[n]&0x80000000)==0) T=0;
    else T=1;
    R[n]<<=1;
    PC+=2;
}
```

Examples:

SHLL	R0	Before execution	R0 = H'80000001, T = 0
		After execution	R0 = H'00000002, T = 1

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

Format		Abstract	Code	Cycle	T Bit
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLL16	Rn	Rn << 16 → 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 6.12).

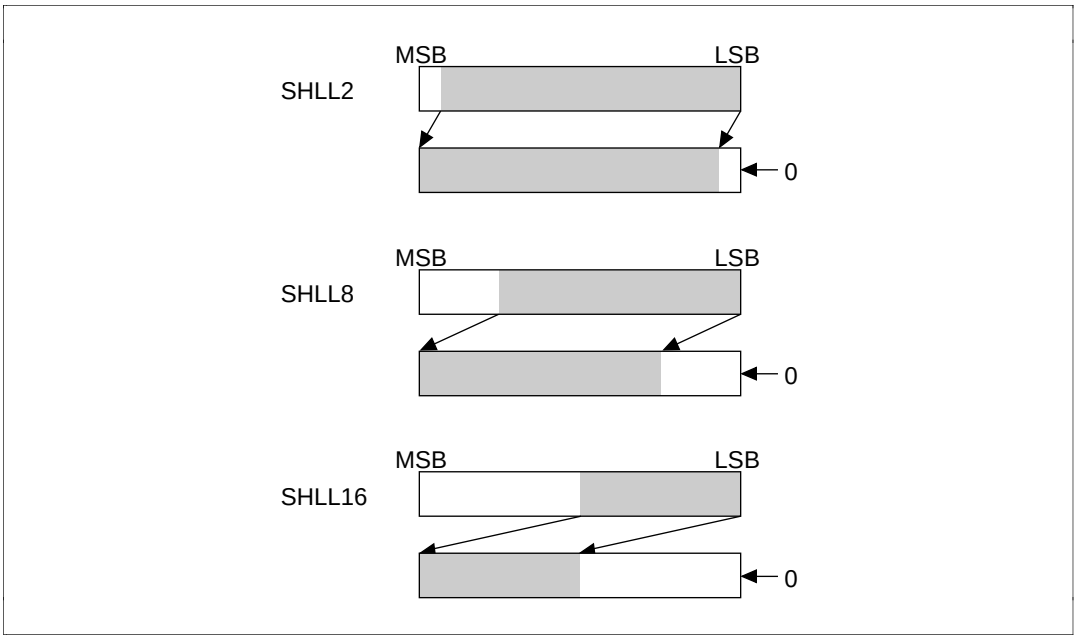


Figure 6.12 Shift Logical Left n Bits

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

6.62 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 6.13).

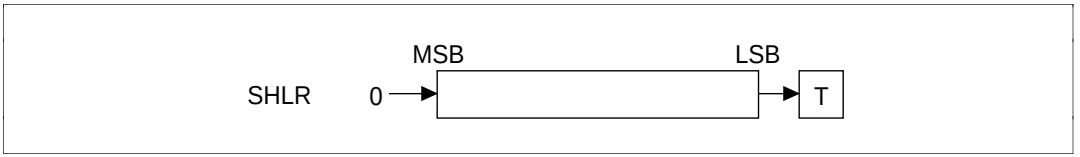


Figure 6.13 Shift Logical Right

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

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

Format		Abstract	Code	Cycle	T Bit
SHLR2	Rn	Rn>>2 → Rn	0100nnnn00001001	1	—
SHLR8	Rn	Rn>>8 → Rn	0100nnnn00011001	1	—
SHLR16	Rn	Rn>>16 → 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 6.14).

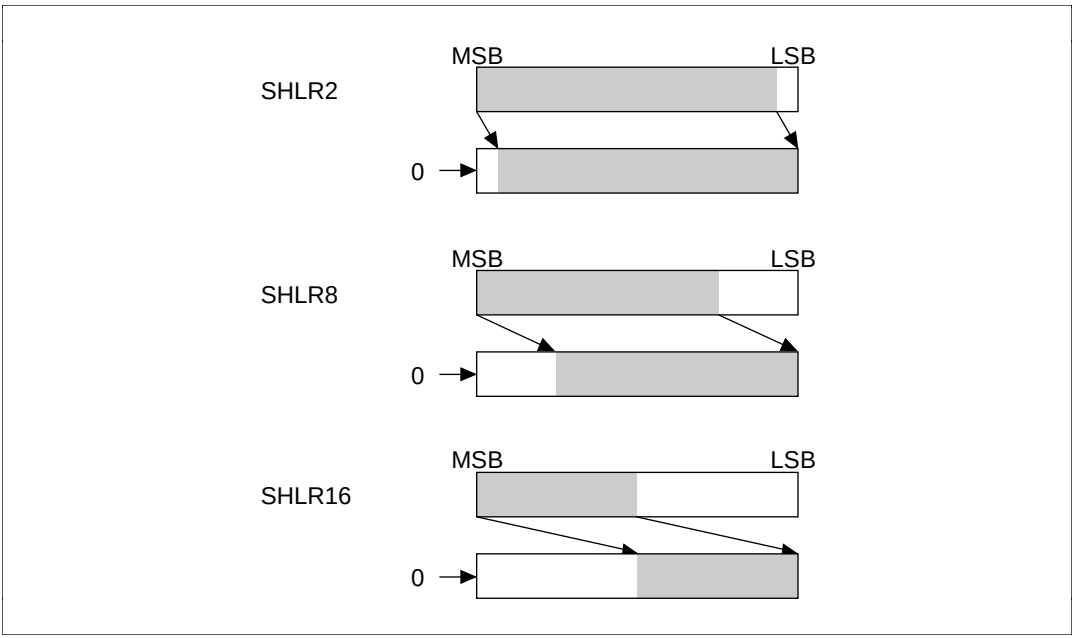


Figure 6.14 Shift Logical Right n Bits

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

6.64 SLEEP (Sleep): System Control Instruction (Privileged Only)

Format	Abstract	Code	Cycle	T Bit
SLEEP	Sleep	0000000000011011	4	—

Description: Sets the CPU into power-down mode. In power-down mode, instruction execution stops, but the CPU module status is maintained, and the CPU waits for an interrupt request. If an interrupt is requested, the CPU exits the power-down mode and begins exception processing.

SLEEP is a privileged instruction and can be used in privileged mode only. If used in user mode, it causes an illegal instruction exception.

Note: The number of cycles given is for the transition to sleep mode.

Operation:

```
SLEEP()    /* SLEEP */
{
    PC-=2;
    Error("Sleep Mode.");
}
```

Examples:

`SLEEP` Enters power-down mode

6.65 STC (Store Control Register): System Control Instruction (Privileged Only)

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	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	R0_BANK,Rn	R0_BANK → Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK → Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK → Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK → Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK → Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK → Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK → Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK → Rn	0000nnnn11110010	1	—
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	—
STC.L	SSR,@-Rn	Rn – 4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
STC.L	SPC,@-Rn	Rn – 4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—
STC.L	R0_BANK,@-Rn	Rn – 4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L	R1_BANK,@-Rn	Rn – 4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L	R2_BANK,@-Rn	Rn – 4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L	R3_BANK,@-Rn	Rn – 4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L	R4_BANK,@-Rn	Rn – 4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L	R5_BANK,@-Rn	Rn – 4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L	R6_BANK,@-Rn	Rn – 4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L	R7_BANK,@-Rn	Rn – 4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—

Description: Stores control registers SR, GBR, VBR, SSR, SPC, or R0–R7_BANK data into a specified destination. STC and STC.L, except for STC GBR, Rn and STC.L GBR, @-Rn are privileged instructions and can be used in privileged mode only. If used in user mode, they cause illegal instruction exceptions. STC GBR, Rn and STC.L GBR, @-Rn can be used in user mode.

Rn_BANK is designated by the RB bit of the SR. When the RB = 1, Rn_BANK0 is accessed by STC/STC.L instructions. When the RB = 0, Rn_BANK1 is accessed by STC/STC.L instructions.

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;
}

STCSSR(long n)     /* STC SSR,Rn */
{
    R[n]=SSR;
    PC+=2;
}

STCSPC(long n)     /* STC SPC,Rn */
{
    R[n]=SPC;
    PC+=2;
}

STCRn_BANK(long m)     /* STC Rn_BANK,Rm */
{                      /* n=0-7 */
    R[m]=Rn_BANK;
    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;
}
STCMSSR(long n)   /* STC.L SSR,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SSR);
    PC+=2;
}

STCMSPC(long n)   /* STC.L SPC,@-Rn */
{
    R[n]-=4;
    Write_Long(R[n],SPC);
    PC+=2;
}
```

```
STCMRn(long m)    /* STC.L Rn_BANK,@-Rnm */
                  /* n=0–7 */
{
    R[m]-=4;
    Write_Long(R[m],Rn_BANK);
    PC+=2;
}
```

Examples:

STC	SR,R0	Before execution	R0 = H'FFFFFFFF, SR = H'00000000
		After execution	R0 = H'00000000
STC.L	GBR,@-R15	Before execution	R15 = H'10000004
		After execution	R15 = H'10000000, @R15 = GBR

6.66 STS (Store System Register): System Control 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 system registers MACH, MACL and PR data into a specified destination.

Operation:

```
STSMACH(long n)   /* STS MACH,Rn */
{
    R[n]=MACH;
    if ((R[n]&0x00000200)==0)
    R[n]&=0x000003FF;
    else R[n]|=0xFFFFFC00;
    PC+=2;
}

STSMACL(long n)   /* STS MACL,Rn */
{
    R[n]=MACL;
    PC+=2;
}

STSPR(long n)     /* STS PR,Rn */
{
    R[n]=PR;
    PC+=2;
}
```

```
STSMMACH(long n)  /* STS.L MACH,@-Rn */
{
    R[n]-=4;
    if ((MACH&0x00000200)==0)
    Write_Long(R[n],MACH&0x000003FF);
    else Write_Long (R[n],MACH|0xFFFFFC00)
    PC+=2;
}

STSMMACL(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;
}
```

Examples:

`STS`	`MACH,R0`	Before execution	R0 = H'FFFFFFFF, MACH = H'00000000
		After execution	R0 = H'00000000
`STS.L`	`PR,@-R15`	Before execution	R15 = H'10000004
		After execution	R15 = H'10000000, @R15 = PR

6.67 SUB (Subtract Binary): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
SUB	Rm,Rn	Rn – Rm → Rn	0011nnnnmmmm1000	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;
}
```

Examples:

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

6.68 SUBC (Subtract with Carry): Arithmetic Instruction

Format		Abstract	Code	Cycle	T Bit
SUBC	Rm,Rn	Rn – Rm– T → Rn, Borrow → T	0011nnnnmmmm1010	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'FFFFFFFF
SUBC	R2,R0	Before execution	T = 1, R0 = H'00000000, R2 = H'00000000
		After execution	T = 1, R0 = H'FFFFFFFF

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

Format		Abstract	Code	Cycle	T Bit
SUBV	Rm,Rn	Rn – Rm → Rn, Underflow → T	0011nnnnmmmm1011	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

6.70 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]&0x0000ff00)>>8;
    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
```

6.71 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	3	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.

Note: The destination of the TAS instruction should be placed in a non-cacheable space when the cache is enabled.

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
```

6.72 TRAPA (Trap Always): System Control Instruction

Format	Abstract	Code	Cycle	T Bit
TRAPA #imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB 0x160 → EXPEVT VBR + H'00000100 → PC	11000011iiiiiiii	6	—

Description: Starts the trap exception processing. The PC and SR values are saved in SPC and SSR. Eight-bit immediate data is stored in the TRA registers (TRA9 to TRA2). The processor goes into privileged mode (SR.MD = 1) with SR.BL = 1 and SR.RB = 1, that is, blocking exceptions and masking interrupts, and selecting BANK1 registers (R0_BANK1 to R7_BANK1). Exception code 0x160 is stored in the EXPEVT register (EXPEVT11 to EXPEVT0). The program branches to an address (VBR+H'00000100).

Operation:

```
TRAPA(long i)  /* TRAPA #imm */
{
    long imm;
    imm=(0x000000FF & i);
    TRA=imm<<2;
    SSR=SR;
    SPC=PC;
    SR.MD=1
    SR.BL=1
    SR.RB=1
    EXPEVT=0x00000160;
    PC=VBR+H'00000100;
}
```

6.73 TST (Test Logical): Logic Operation Instruction

Format		Abstract	Code	Cycle	T Bit
TST	Rm,Rn	Rn & Rm, when result is 0, 1 →T	0010nnnnmmmm1000	1	Test results
TST	#imm,R0	R0 & imm, when result is 0, 1 →T	11001000iiiiiiii	1	Test results
TST.B	#imm,@(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	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

6.74 XOR (Exclusive OR Logical): Logic Operation Instruction

Format		Abstract	Code	Cycle	T Bit
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—
XOR.B	#imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiiii	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'FFFFFF0F
`XOR.B`	`#H'A5,@(R0,GBR)`	Before execution	@(R0,GBR) = H'A5
		After execution	@(R0,GBR) = H'00

6.75 XTRCT (Extract): Data Transfer Instruction

Format	Abstract	Code	Cycle	T Bit
XTRCT Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

Description: Extracts the middle 32 bits from the 64 bits of general registers Rm and Rn, and stores the 32 bits in Rn (figure 6.15).

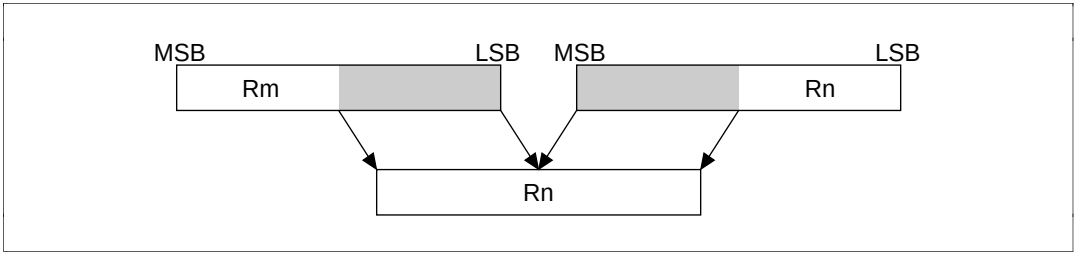


Figure 6.15 Extract

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

Section 7 Processing States

7.1 State Transitions

The CPU has five processing states: reset, exception processing, bus release, program execution and power-down. The transitions between the states are shown in figure 7.1. For more information, see the *SH7700 Series Hardware Manual*.

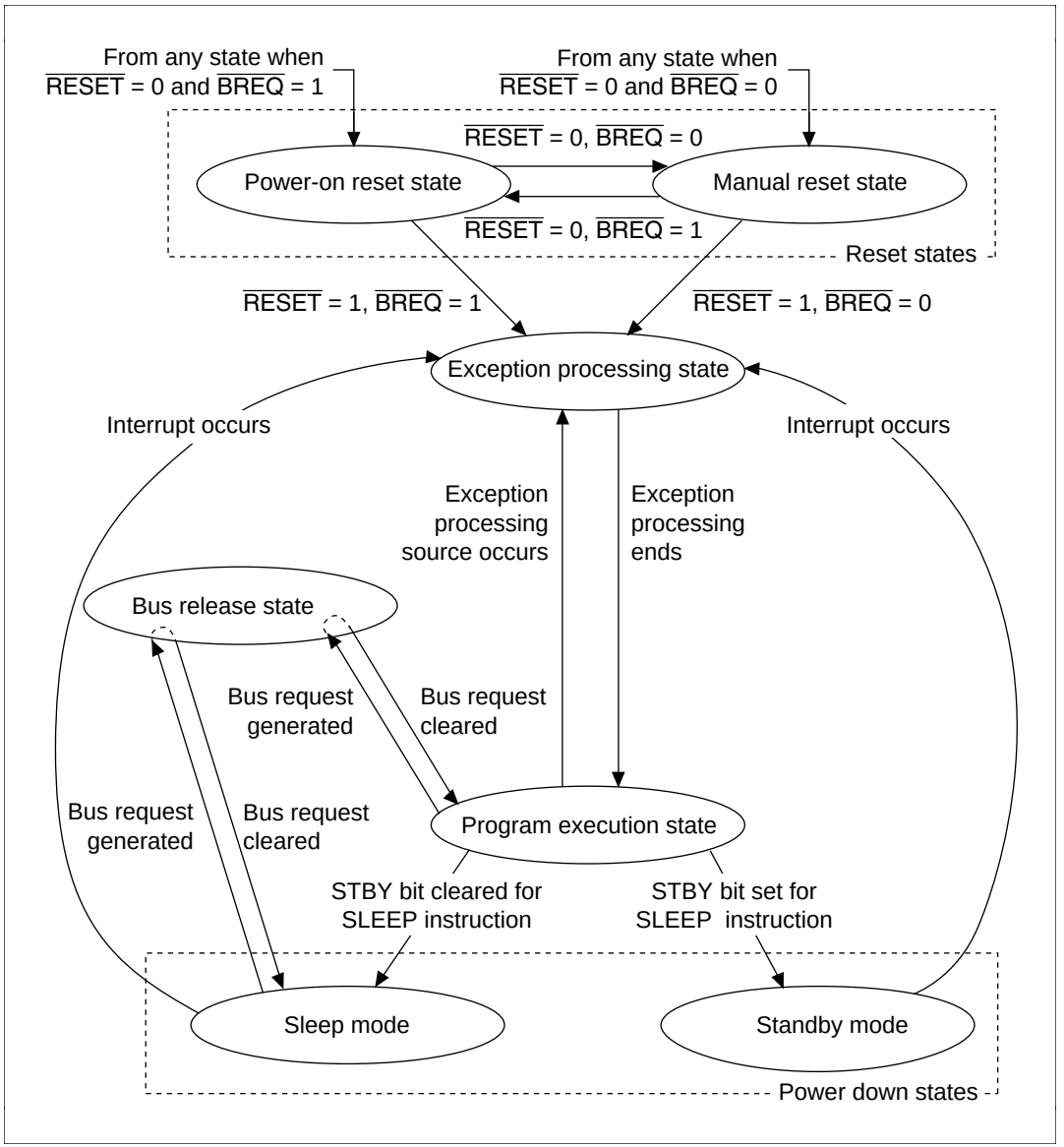


Figure 7.1 Transitions between Processing States

7.1.1 Reset State

In the reset state, the CPU is reset. This occurs when the $\overline{\text{RESET}}$ pin level goes low. When the $\overline{\text{BREQ}}$ pin is high, the result is a power-on reset; when it is low, a manual reset will occur.

7.1.2 Exception Processing State

The exception processing state is a transient state that occurs when the CPU's processing state flow is altered by exception processing sources such as resets, general exceptions, or interrupts.

For a reset, the CPU branches to H'A0000000 and starts executing the user-created exception process program.

For a general exception or interrupt, the program counter (PC) is saved in the save program counter (SPC), and the status register (SR) is saved in the save status register (SSR). The CPU then branches to the starting address of the user-created exception service routine by adding the content of the vector base address and the vector offset, thereby starting program execution state.

7.1.3 Program Execution State

In the program execution state, the CPU sequentially executes the program.

7.1.4 Power-Down State

In the power-down state, the CPU operation halts and power consumption declines. The SLEEP instruction places the CPU in the power-down state. This state has two modes: sleep mode and standby mode. See section 7.2 for more details.

7.1.5 Bus Release State

In the bus release state, the CPU releases access rights to the bus to the device that has requested them.

7.2 Power-Down State

In addition to the ordinary program execution states, the CPU also has a power-down state in which CPU operation halts and power consumption is lowered (table 7.1). There are three power-down state modes: sleep mode, standby mode, and module stop mode.

7.2.1 Sleep Mode

When standby bit STBY (in the standby control register STBCR) is cleared to 0 and a SLEEP instruction executed, the CPU moves from program execution state to sleep mode. In sleep mode, the CPU halts, and the contents of its internal registers and the data in on-chip cache and TLB data are maintained. The on-chip peripheral modules other than the CPU do not halt in the sleep mode.

To return from sleep mode, use a reset or any interrupt; the CPU returns to ordinary program execution state through the exception processing state.

7.2.2 Standby Mode

To enter the standby mode, set the standby bit STBY (in the standby control register STBCR) to 1 and execute a SLEEP instruction. In standby mode, all CPU, on-chip peripheral module and oscillator functions are halted. CPU internal register contents and on-chip cache and TLB data are held.

To return from standby mode, use a reset or an interrupt (NMI, IRQ, on-chip peripheral). For resets, the CPU returns to ordinary program execution state through the exception processing state when placed in a reset state after the oscillator stabilization time has elapsed. For interrupts, the CPU returns to ordinary program execution state through the exception processing state after the oscillator stabilization time has elapsed. In this mode, power consumption drops markedly, since the oscillator stops.

7.2.3 Module Stop Mode

The supply of the clock to on-chip peripheral modules can be halted by setting the corresponding module stop bits (MSTP) in the standby control register (STBCR) to 1. Using this function can reduce the power consumption in sleep mode.

The external pins of the on-chip peripheral modules in module standby are reset by the module stop mode. Module stop mode can be cleared by clearing the MSTP bits to 0.

Table 7.1 Power-Down Modes

Mode	Entering Procedure	State: CPG	State: CPU	State: CPU Register	State: On-Chip Memory	State: On-Chip Peripheral Modules	State: Pins	State: External Memory	Canceling Procedure
Sleep mode	Execute SLEEP instruction with STBY bit set to 0 in STBCR	Run	Halt	Held	Held	Run	Held	Refresh	1. Interrupt 2. Reset
Standby mode	Execute SLEEP instruction with STBY bit set to 1 in STBCR	Halt	Halt	Held	Held	Halt*	Held	Self-refresh	1. Interrupt 2. Reset
Module standby	Set MSTP bit of STBCR to 1	Run	Run	Held	Held	Specified module halts	Held	Refresh	1. Set MSTP bit to 0 2. Reset

Note: The RTC still runs if the START bit of RCR2 is set to a logic one (see section 12 of the *SH7700 Series Hardware Manual*, Realtime Clock (RTC)). TMU still runs when output of the RTC is used as input to its counter (see section 11 of the *SH7700 Series Hardware Manual*, Timer (TMU)).

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

Pipelines are composed of the following five stages:

- IF (Instruction fetch) — Fetches instruction from the memory stored in the program.
- 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.

As shown in figure 8.1, these stages flow with the execution of the instructions and thereby constitute a pipeline. At a given instant, five instructions are being executed simultaneously. All instructions have at least the three stages: IF, ID, and EX. Most, but not all, have stages MA and WB as well. The way the pipeline flows also varies with the type of instruction. The basic pipeline flow is as shown in figure 8.1; some pipelines differ, however, because of contention between IF and MA. In figure 8.1, the period in which a single stage is operating is called a slot.

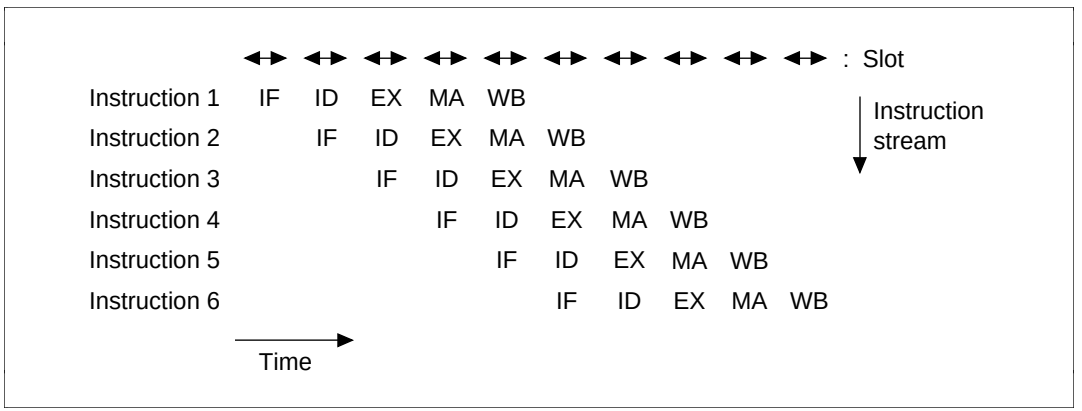


Figure 8.1 Basic Structure of Pipeline Flow

8.2 Slot and Pipeline Flow

The time period in which a single stage operates is called a slot. Slots must follow the rules described below.

8.2.1 Instruction Execution

Each stage (IF, ID, EX, MA, WB) of an instruction must be executed in one slot. Two or more stages cannot be executed within one slot (figure 8.2), with exception of WB and MA.

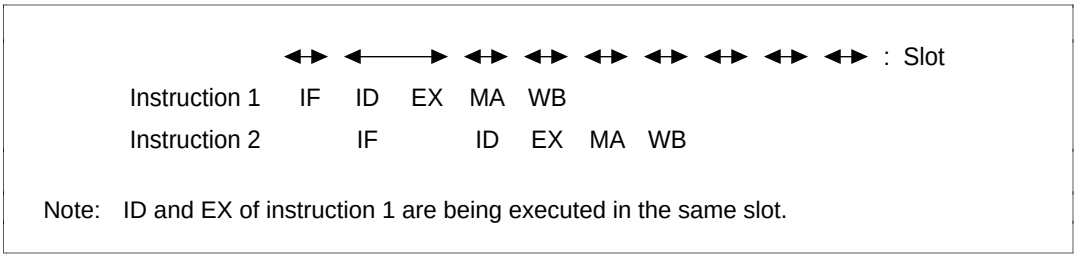


Figure 8.2 Impossible Pipeline Flow 1

8.2.2 Slot Sharing

A maximum of one stage from another instruction may be set per slot, and that stage must be different from the stage of the first instruction. Identical stages from two different instructions may never be executed within the same slot (figure 8.3).

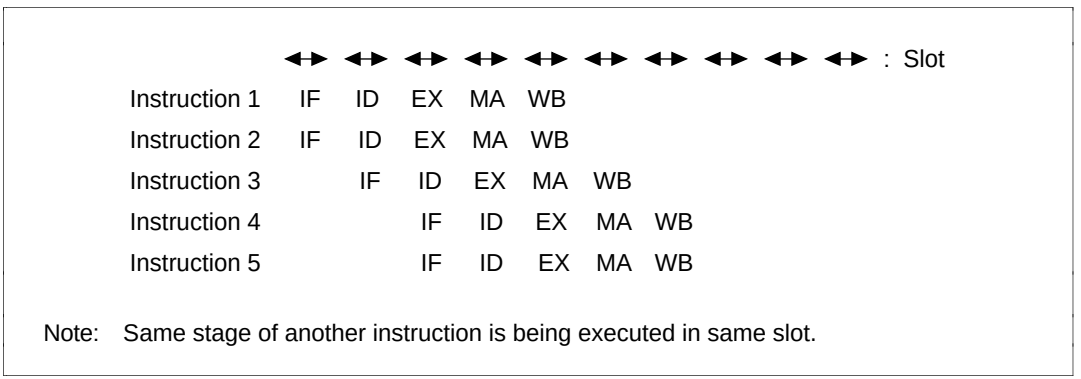


Figure 8.3 Impossible Pipeline Flow 2

8.2.3 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.

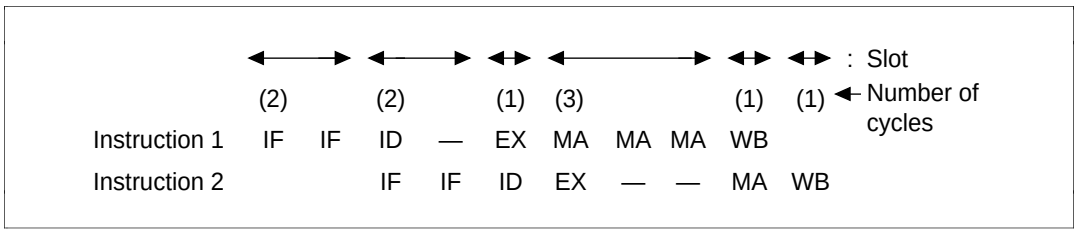


Figure 8.4 Slots Requiring Multiple Cycles

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 the case of 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 execution time between instructions 1 and 3 in figure 8.5 is seven cycles (5 + 1 + 1).

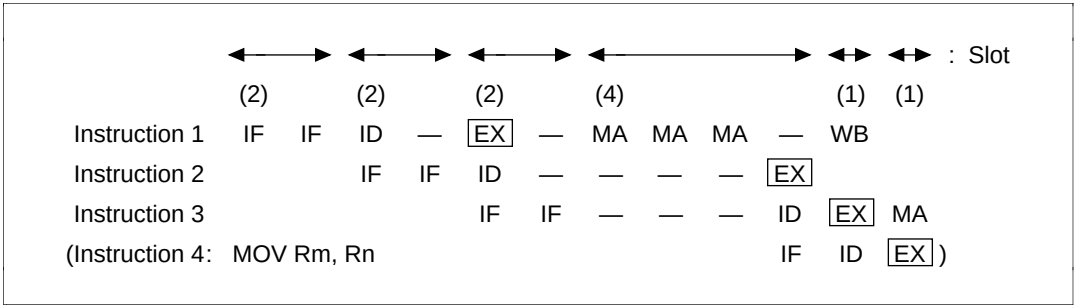


Figure 8.5 How Instruction Execution Cycles Are Counted

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

8.4.1 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.

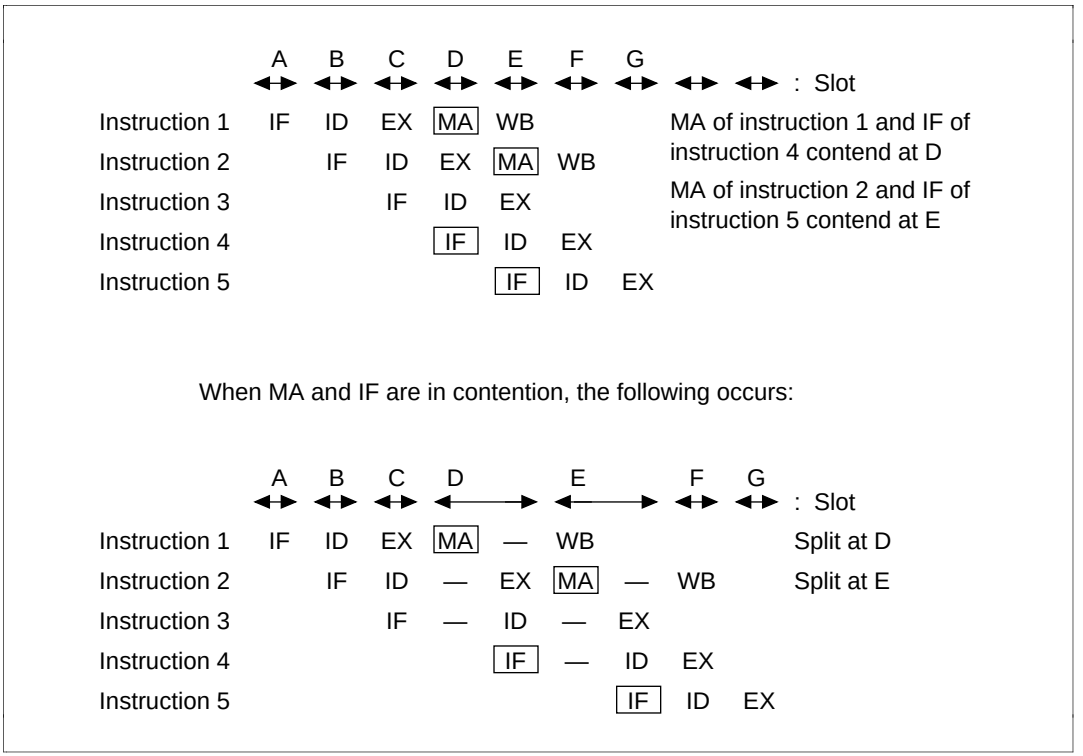


Figure 8.6 Operation when IF and MA Are in Contention

The slots in which MA and IF contend are split. MA and WB are given priority to execute in the first half, 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 and the WB of instruction 1 are given priority, and the EX of instruction 3, the ID of instruction 4, and the IF of instruction 5, are 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.

8.4.2 Relationship between IF and the Location of Instructions in Memory

When the instruction is located in memory, the SH microcomputer accesses the memory in 32-bit units. The SH microcomputer instructions are all fixed at 16 bits, so basically 2 instructions can be fetched in a single IF stage access. Whether an IF fetches one or two instructions depends on the memory location (word or longword boundary).

If an instruction is located on a longword boundary, an IF can get two instructions at each instruction fetch. The IF of the next instruction does not generate a bus cycle to fetch an instruction from memory. Since the next instruction IF also fetches two instructions, the instruction IFs after that do not generate a bus cycle either.

This means that IFs of instructions that are located so they start from the longword boundaries within instructions located in memory (the position when the bottom two bits of the instruction address are 00 is A1 = 0 and A0 = 0) also fetch two instructions. The IF of the next instruction does not generate a bus cycle. IFs that do not generate bus cycles are written in lower case as “if”. These ifs always take one cycle.

When branching results in a fetch from an instruction located so it starts from the word boundaries (the position when the bottom two bits of the instruction address are 10 is A1 = 1, A0 = 0), the bus cycle of the IF fetches only the specified instruction more than one of said instructions. The IF of the next instruction thus generates a bus cycle, and fetches two instructions. Figure 8.7 illustrates these operations.

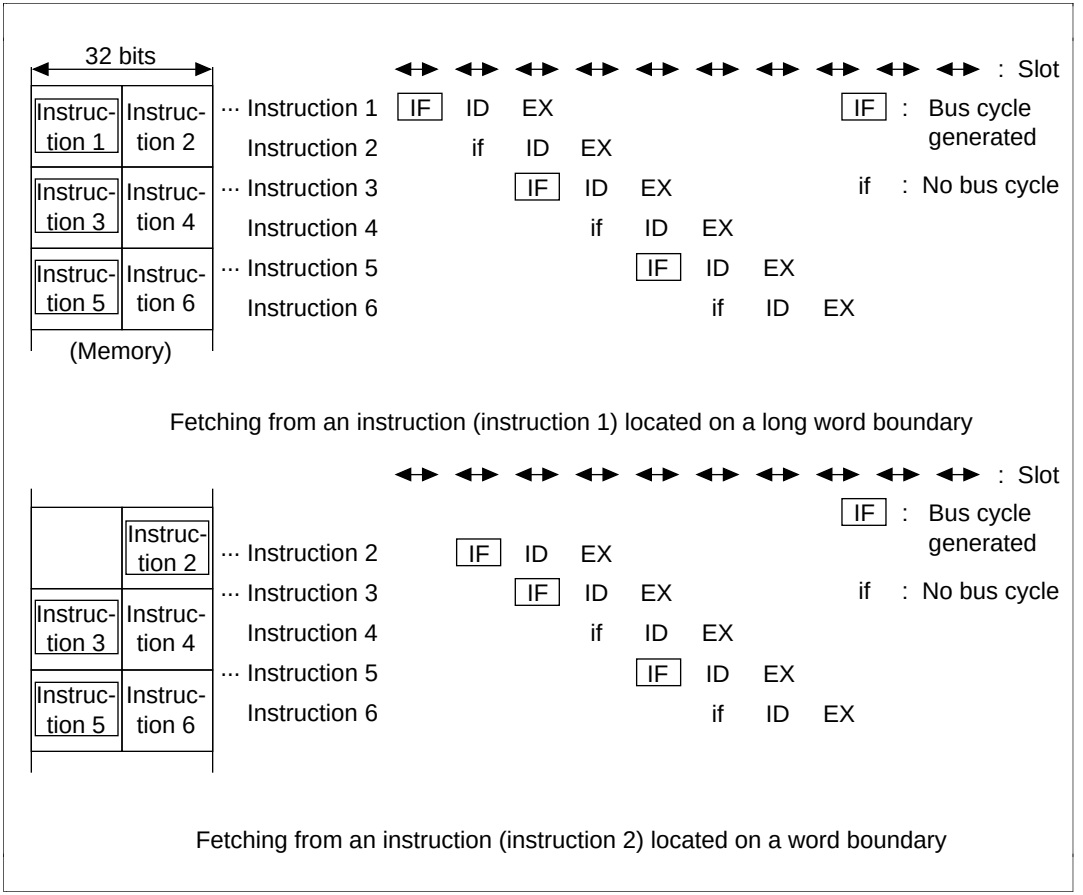


Figure 8.7 Relationship between IF and Location of Instructions in Memory

8.4.3 Relationship between Position of Instructions Located in Memory and Contention between IF and MA

When an instruction is located in memory, there are instruction fetch stages ("if", written in lower case) that do not generate bus cycles as explained in section 8.4.2 above. 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, as 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. Instructions that have 4 (5)-stage pipelines of IF, ID, EX, MA, (WB) prevent stalls when they are located, so they start from the longword boundaries in memory (the position when the bottom 2 bits of instruction address are 00 is A1 = 0 and A0 = 0) because the MA of the instruction falls in the same slot as ifs that follow.

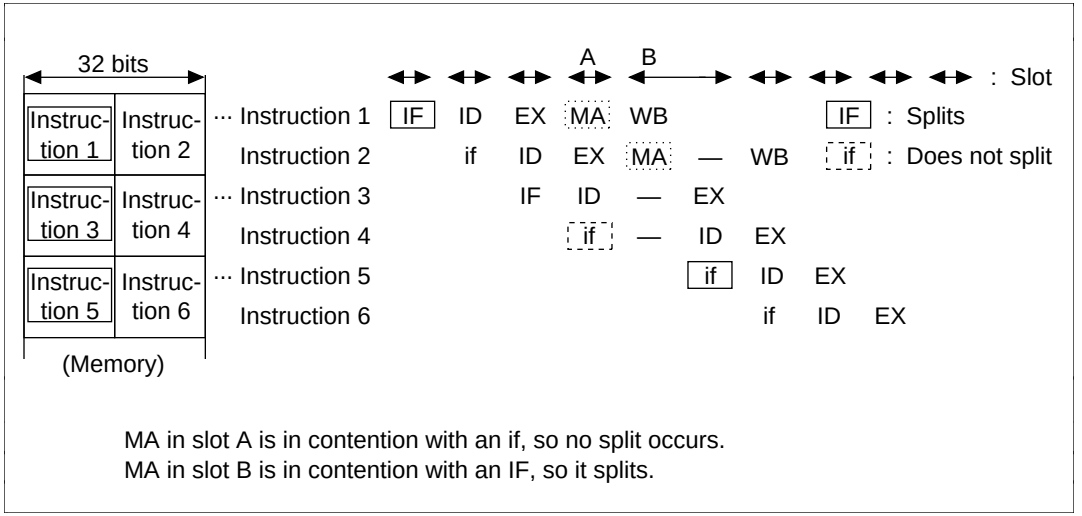


Figure 8.8 Relationship between the Location of Instructions in Memory and Contention between IF and MA

8.5 Effects of Memory Load Instructions on Pipelines

Instructions that involve loading from memory access data in memory at the MA stage of the pipeline. In the case of a load instruction (instruction 1) and the following instruction (instruction 2), the EX stage of instruction 2 starts before the MA stage of instruction 1 ends.

When instruction 2 uses the same data that instruction 1 is loading, the contents of that register will not be ready, so any slot containing the MA of instruction and EX of instruction 2 will split. No split occurs, however, 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 illustrated 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.

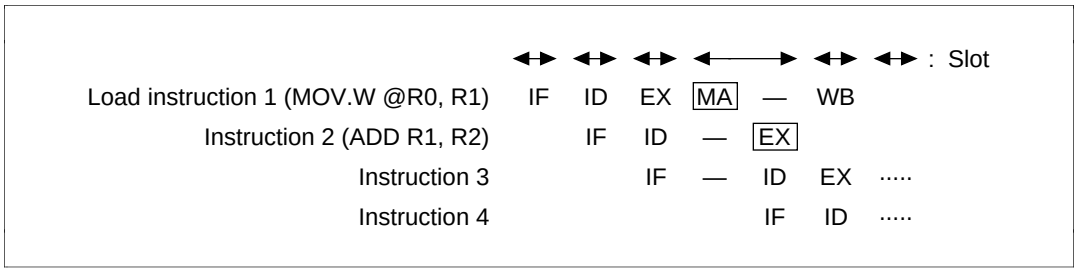


Figure 8.9 Effects of Memory Load Instructions on the Pipeline

8.6 Multiplier Access Contention

A multiplier-type instruction (multiply/accumulate calculations, multiplier instructions), an instruction in which the multiply and accumulate registers (MACH, MACL) are accessed, can cause a contention in the multiplier access.

In the multiplier instruction, the multiplier takes action regardless of the slots after the ending of the last MA. In the double precision (64 bytes) type multiplier instruction and the multiply/accumulate calculations instruction, the multiplier takes action in three states. In the single precision (32 bytes) type multiplier instruction, the action is taken in two states.

When MA (when there are two, the first MA takes precedence) of the multiplier instruction (multiply/accumulate calculations, multiplier instruction) contends with the multiplier access (mm) of the preceding multiplier instruction, the MA bus cycle is extended until the mm ends. The extended MA then becomes one slot.

The MA instruction which accesses the multiply/accumulate register (MACH, MACL) also accesses the multiplier. Similar to the multiplier instruction, the MA bus cycle is extended until the mm of the preceding multiplier-type instruction ends, and the extended MA becomes one slot. In particular, in the instruction (STS, STS.L), which reads out the multiply/accumulate register (MACH, MACL,MA) is extended until one slot has elapsed after the ending of the mm, the extended MA becomes one slot.

On the other hand, when the instruction has two MAs, the succeeding ID instruction is stalled for a one-slot period.

Because the multiplier-type instruction and the multiply/accumulate register access instruction both have MA cycles, a contention with IF may develop.

Examples of multiplier access contention are shown in figures 8.10 and 8.11. In these cases, the contention between MA and IF is not taken into consideration.

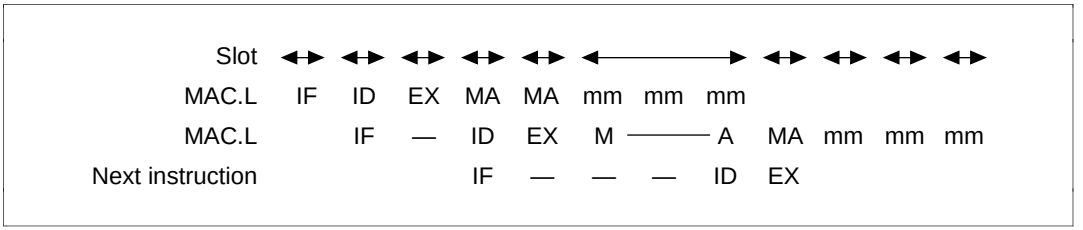


Figure 8.10 Contention between Two MAC.L Instructions

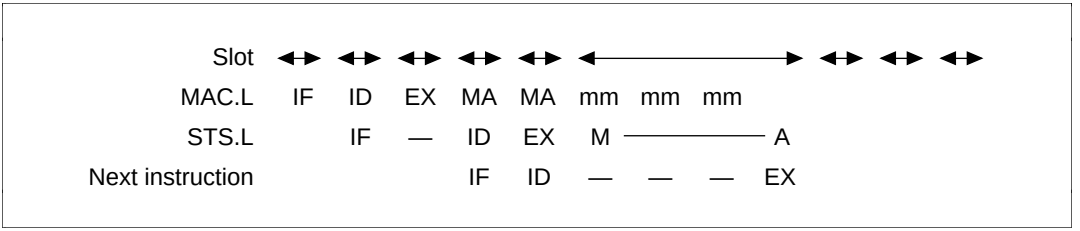


Figure 8.11 Contention between the MAC.L and STS.L Instructions

8.7 Programming Guide

To improve instruction execution speed, consider the following when programming:

- To prevent contention between MA and IF, locate instructions that have MA stages so they start from the longword boundaries of on-chip memory (the position when the bottom two bits of the instruction address are 00 is A1 = 0 and A0 = 0) wherever possible.
- The instruction that immediately follows an instruction that loads from memory should not use the same destination register as the load instruction.
- Locate instructions that use the multiplier nonconsecutively.

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	• Contention occurs if the instruction placed immediately after this one uses the same destination register • 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
	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)

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction
Data transfer instructions (cont)	Memory store instructions (cont)	4	1	• MA contends with IF	`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)`
	Cache instruction	4	1	—	`PREF @Rn`
Arithmetic instructions	Arithmetic instructions between registers (except multiplica-tion 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`
					`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`

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction
Arithmetic instructions (cont)	Multiply/ accumulate instruction	7	2 (to 5)*1	• Contention with the multiplier occurs when an instruction that uses the multiplier comes after a MAC instruction • MA contends with IF	`MAC.W @Rm+,@Rn+`
	Double length/ multiply accumulate instruction	9	2 (to 5)*1	• Contention with the multiplier occurs when an instruction that uses the multiplier comes after a MAC instruction • MA contends with IF	`MAC.L @Rm+,@Rn+`
	Multiplic-ation instruction	6	1 (to 3)*1	• Contention with the multiplier occurs when an instruction that uses the multiplier comes after a MUL instruction • MA contends with IF	`MULS.W Rm,Rn` `MULU.W Rm,Rn`

	Double length multipli-cation instructions	9	2 (to 5)*1	• Contention with the multiplier occurs when an instruction that uses the multiplier comes after a MUL instruction
				• MA contends with IF
					`DMULS.L Rm,Rn`
					`DMULU.L Rm,Rn`
					`MUL.L Rm,Rn`

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction	
Logic operation instructions	Register to 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)
					XOR.B	#imm,@(R0,GBR)
	TAS instruction	6	3	• 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
	Dynamic shift instructions	3	1	—	SHAD	Rm,Rn
					SHLD	Rm,Rn

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction	
Branch instructions	Conditional branch instructions	3	3/1*2	—	BF	disp
					BT	disp
	Delayed conditional branch instructions	3	2/1*2	—	BF/S	label
					BT/S	label
	Uncondi-tional branch instructions	3	2	—	BRA	disp
					BRAF	Rn
					BSR	disp
					BSRF	Rn
					JMP	@Rn
					JSR	@Rn
					RTS	
System control instructions	System control ALU instructions	3	1	—	CLRS	
					CLRT	
					LDC	Rm,SR
					LDC	Rm,GBR
					LDC	Rm,VBR
					LDC	Rm,SSR
					LDC	Rm,SPC
					LDC	Rm,R0_BANK
					LDC	Rm,R1_BANK
					LDC	Rm,R2_BANK
					LDC	Rm,R3_BANK
					LDC	Rm,R4_BANK
					LDC	Rm,R5_BANK
					LDC	Rm,R6_BANK
					LDC	Rm,R7_BANK
					LDS	Rm,PR
					LDTLB	
					NOP	
					SETS	
					SETT	

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction	
System control instructions (cont)	System control ALU instructions (cont)	3	1	—	STC	SR,Rn
					STC	GBR,Rn
					STC	VBR,Rn
					STC	SSR,Rn
					STC	SPC,Rn
					STC	R0_BANK,Rn
					STC	R1_BANK,Rn
					STC	R2_BANK,Rn
					STC	R3_BANK,Rn
					STC	R4_BANK,Rn
					STC	R5_BANK,Rn
					STC	R6_BANK,Rn
					STC	R7_BANK,Rn
					STS	PR,Rn
	LDC instructions (SR)	5	5	—	LDC	Rm,SR
	LDC.L instructions (SR)	7	7	• MA contends with IF	LDC.L	@Rm+,SR
	LDC.L instructions	5	1	• Contention occurs when an instruction that uses the same destination register is placed immediately after this instruction • MA contends with IF	LDC.L	@Rm+,GBR
					LDC.L	@Rm+,VBR
					LDC.L	@Rm+,SSR
					LDC.L	@Rm+,SPC
					LDC.L	@Rm+,R0_BANK
					LDC.L	@Rm+,R1_BANK
					LDC.L	@Rm+,R2_BANK
					LDC.L	@Rm+,R3_BANK
					LDC.L	@Rm+,R4_BANK
					LDC.L	@Rm+,R5_BANK
					LDC.L	@Rm+,R6_BANK
					LDC.L	@Rm+,R7_BANK

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction	
System control instructions (cont)	STC.L instructions	4	1	• MA contends with IF	STC.L	SR,@–Rn
					STC.L	GBR,@–Rn
					STC.L	VBR,@–Rn
					STC.L	SSR,@–Rn
					STC.L	SPC,@–Rn
		5	2	• MA contends with IF	STC.L	R0_BANK,@–Rn
					STC.L	R1_BANK,@–Rn
					STC.L	R2_BANK,@–Rn
					STC.L	R3_BANK,@–Rn
					STC.L	R4_BANK,@–Rn
					STC.L	R5_BANK,@–Rn
					STC.L	R6_BANK,@–Rn
					STC.L	R7_BANK,@–Rn
	LDS.L instructions (PR)	5	1	• 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
	STS.L instruction (PR)	4	1	• MA contends with IF	STS.L	PR,@–Rn
	Register → MAC transfer instruction	4	1	• Contention occurs with multiplier • MA contends with IF	CLRMAC	
					LDS	Rm,MACH
					LDS	Rm,MACL
	Memory → MAC transfer instructions	4	1	• Contention occurs with multiplier • MA contends with IF	LDS.L	@Rm+,MACH
					LDS.L	@Rm+,MACL

Table 8.1 Number of Instruction Stages and Execution Cycles (cont)

Type	Category	Stages	Cycles	Contention	Instruction
System control instructions (cont)	MAC → register transfer instruction	5	1	• 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
	MAC → memory transfer instruction	4	1	• Contention occurs with multiplier • MA contends with IF	STS.L MACH,@–Rn STS.L MACL,@–Rn
	RTE instruction	5	4	—	RTE
	TRAP instruction	9	6	—	TRAPA #imm
	SLEEP instruction	3	4	—	SLEEP

Notes: 1. Indicates the normal minimum number of execution states (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 to 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

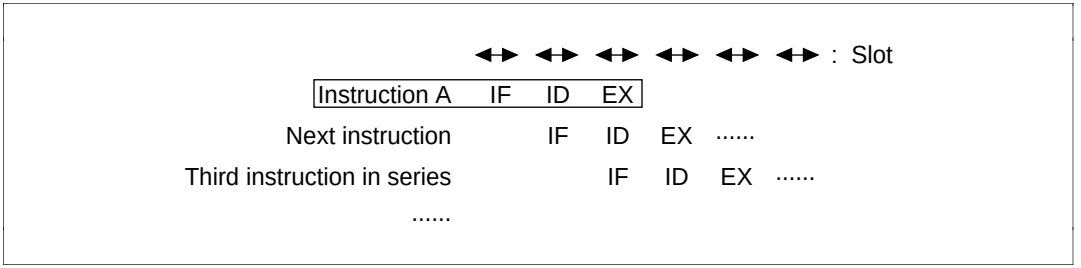


Figure 8.12 Register to Register Transfer Instruction Pipeline

The pipeline ends after three stages: IF, ID, and EX. Data is transferred in the EX stage via the ALU (figure 8.12).

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

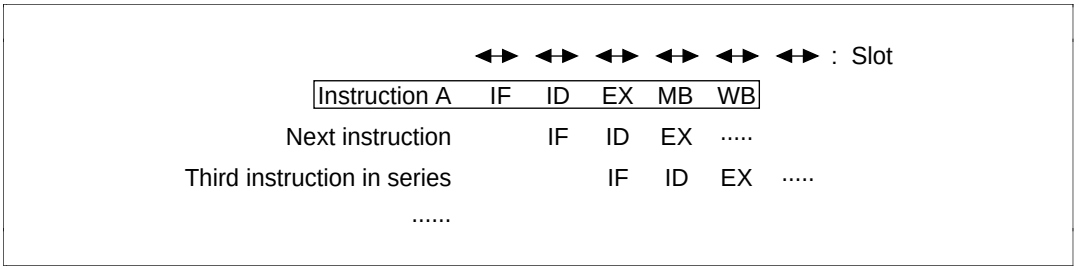


Figure 8.13 Memory Load Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.13). 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 Pipelines.)

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)

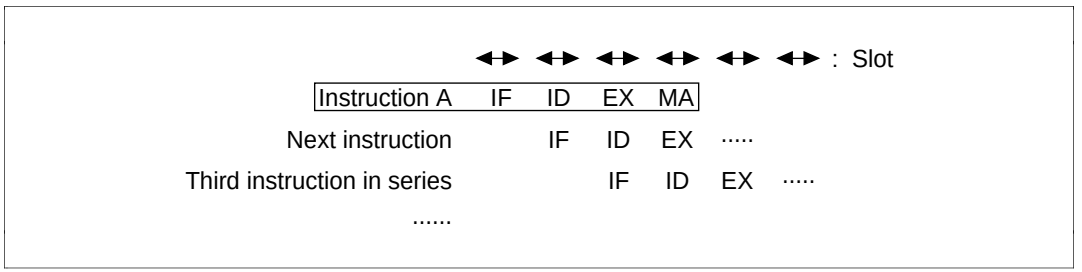


Figure 8.14 Memory Store Instructions Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.14). 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): Instruction types:

- 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
- 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

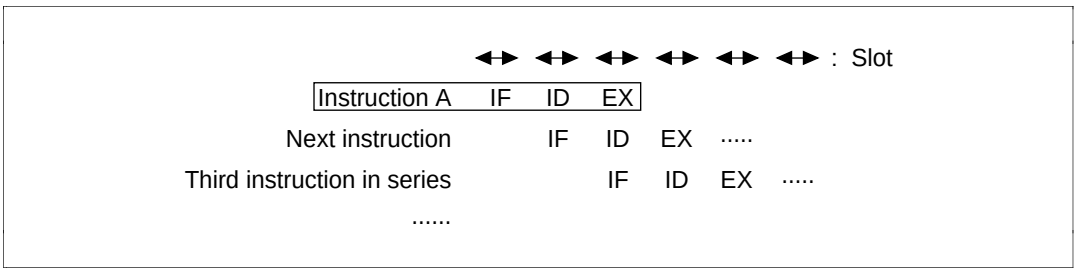


Figure 8.15 Pipeline for Arithmetic Instructions between Registers Except Multiplication Instructions

The pipeline has three stages: IF, ID, and EX (figure 8.15). The data operation is completed in the EX stage via the ALU.

Multiply/Accumulate Instruction: Instruction type:

- MAC.W @Rm+, @Rn+

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
	MAC	IF	ID	EX	MA	MA	mm	mm	mm
Next instruction			IF	—	ID	EX	MA	WB	
Third instruction in series					IF	ID	EX	MA	WB
.....									

Figure 8.16 Multiply/Accumulate Instruction Pipeline

The pipeline has eight stages: IF, ID, EX, MA, MA, mm, mm, and mm (figure 8.16). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three cycles after the final MA ends, regardless of slot. The ID of the instruction after the MAC instruction is stalled for one slot. The two MAs of the MAC 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 instruction, the MAC 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 instruction, contention occurs with the multiplier, so operation is not as normal.

Double-Length Multiply/Accumulate Instruction: Instruction type:

- MAC.L @Rm+, @Rn+

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄► : Slot
MAC.L	IF	ID	EX	MA	MA	mm	mm	mm	
Next instruction		IF	—	ID	EX	MA	WB		
Third instruction				IF	ID	EX	MA	WB	
.....									

Figure 8.17 Multiply/Accumulate Instruction Pipeline

The pipeline has eight stages: IF, ID, EX, MA, MA, mm, mm, and mm (figure 8.17). The second MA reads the memory and accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three 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 not as normal.

Multiplication Instructions: Instruction types:

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

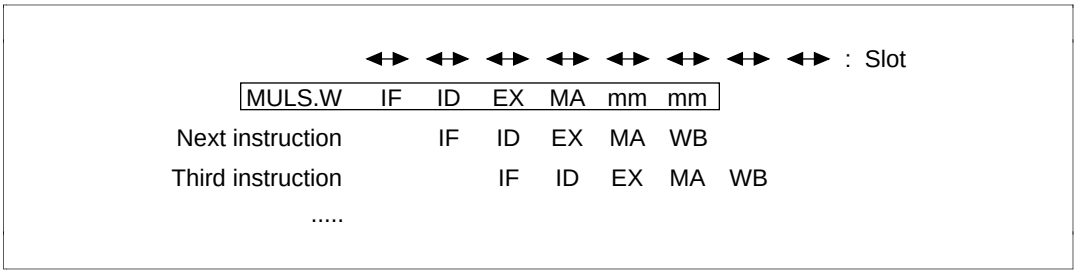


Figure 8.18 Multiplication Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, mm, and mm (figure 8.18). The MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for two cycles after the MA ends, regardless of the 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 come after the MULS.W instruction, however, contention occurs with the multiplier, so operation is not as normal.

Double-Length Multiplication Instructions: Instruction types:

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

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
DMULS.L	IF	ID	EX	MA	MA	mm	mm	mm				
Next instruction		IF	—	ID	EX	MA	WB					
Third instruction				IF	ID	EX	MA	WB				
.....												

Figure 8.19 Multiplication Instruction Pipeline

The pipeline has eight stages: IF, ID, EX, MA, MA, mm, mm, and mm (figure 8.19). The MA accesses the multiplier. The mm indicates that the multiplier is operating. The mm operates for three 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 come after the DMULS.L instruction, however, contention occurs with the multiplier, so operation is not as normal.

8.8.3 Logic Operation Instructions

Register to Register Logic Operation Instructions: Instruction types:

- AND Rm, Rn
- AND #imm, R0
- NOT Rm, Rn
- OR Rm, Rn
- OR #imm, R0
- TST Rm, Rn
- TST #imm, R0
- XOR Rm, Rn
- XOR #imm, R0

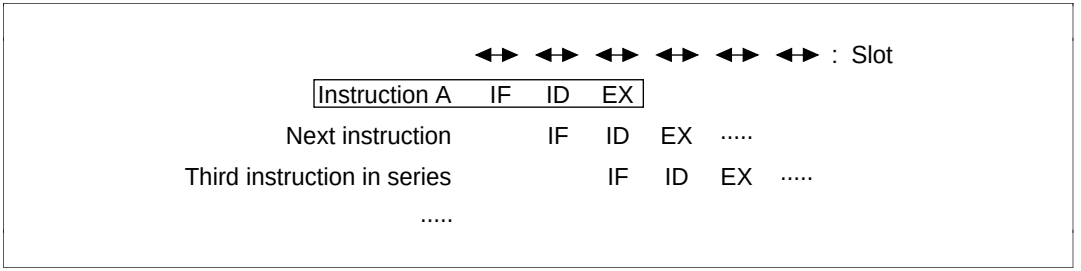


Figure 8.20 Register to Register Logic Operation Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.20). The data operation is completed in the EX stage via the ALU.

Memory Logic Operations Instructions: Instruction types:

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

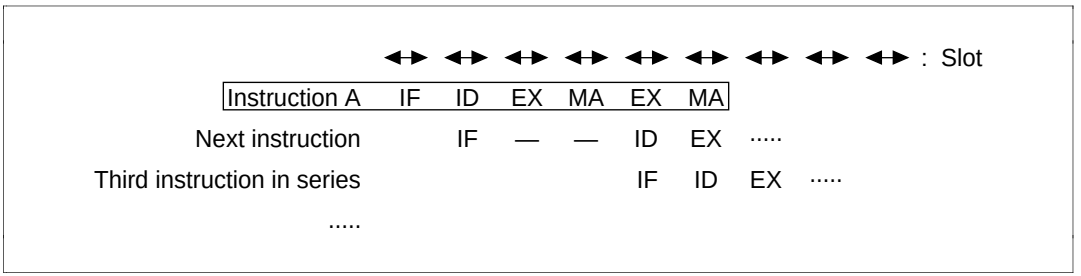


Figure 8.21 Memory Logic Operation Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 8.21). The ID of the next instruction stalls for 2 slots. The MAs of these instructions contend with IF.

TAS Instruction: Instruction type:

- TAS.B @Rn

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction A	IF	ID	EX	MA	EX	MA				
Next instruction		IF	—	—	ID	EX	—	·····		
Third instruction in series						IF	ID	EX	·····	
·····										

Figure 8.22 TAS Instruction Pipeline

The pipeline has six stages: IF, ID, EX, MA, EX, and MA (figure 8.22). The ID of the next instruction stalls for two slots. The MA of the TAS instruction contends with IF.

8.8.4 Shift Instructions

Shift Instructions: Instruction types:

- ROTL Rn
- ROTR Rn
- ROTCL Rn
- ROTCR Rn
- SHAL Rn
- SHAR Rn
- SHLL Rn
- SHLR Rn
- SHLL2 Rn
- SHLR2 Rn
- SHLL8 Rn
- SHLR8 Rn
- SHLL16 Rn
- SHLR16 Rn
- SHAD
- SHLD

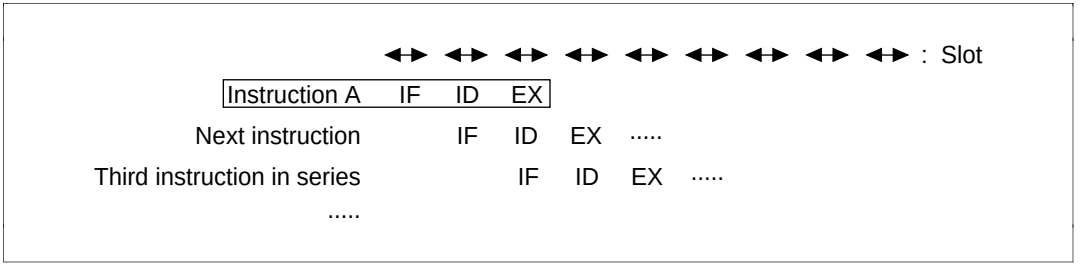


Figure 8.23 Shift Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.23). The data operation is completed in the EX stage via the ALU.

8.8.5 Branch Instructions

Conditional Branch Instructions: Instruction types:

- BF disp
- BT disp

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 8.24).

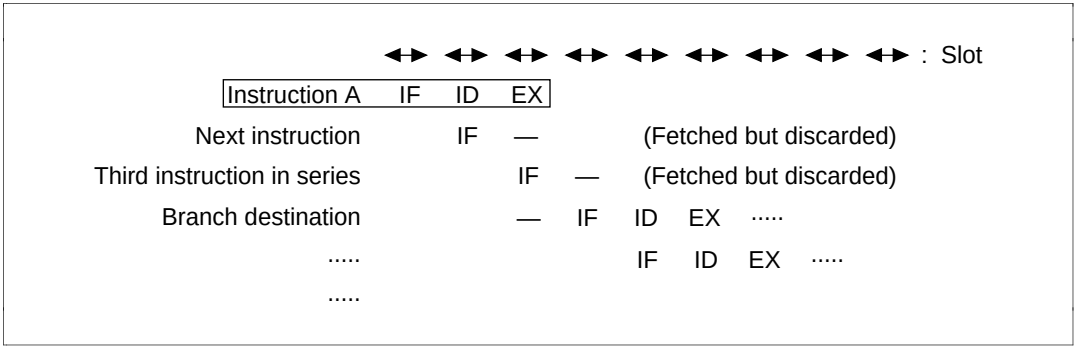


Figure 8.24 Branch Instruction when Condition is Satisfied

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 8.25).

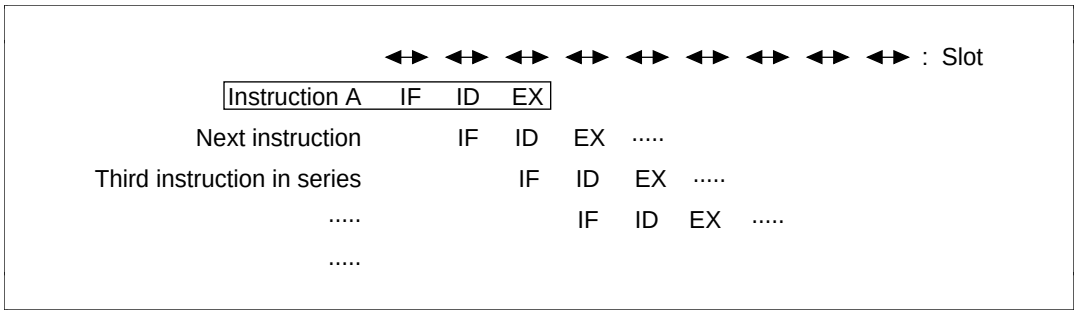


Figure 8.25 Branch Instruction when Condition is Not Satisfied

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 8.26).

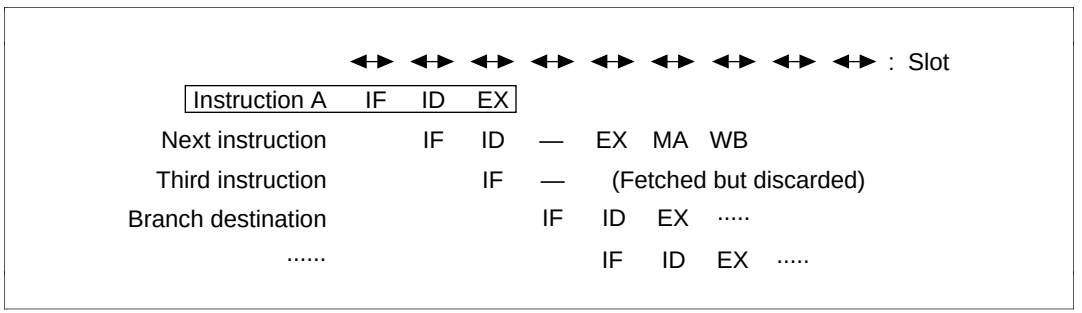


Figure 8.26 Branch Instruction when Condition is Satisfied

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 8.27).

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction A	IF	ID	EX							
Next instruction		IF	ID	EX	·····					
Third instruction			IF	ID	EX	·····				
·····				IF	ID	EX	·····			
·····										

Figure 8.27 Branch Instruction when Condition is Not Satisfied

Unconditional Branch Instructions: Include the following instruction types:

- BRA disp
- BRAF Rn
- BSR disp
- BSRF Rn
- JMP @Rn
- JSR @Rn
- RTS

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction A	IF	ID	EX							
Delay slot		IF	—	ID	EX	MA	WB			
Branch destination				IF	ID	EX	·····			
·····					IF	ID	EX	·····		
·····										

Figure 8.28 Unconditional Branch Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.28). 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 the conditional branch instructions are, but is then 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:

- CLRS
- CLRT
- LDC Rm, GBR
- LDC Rm, VBR
- LDC Rm, SSR
- LDC Rm, SPC
- LDC Rm, R0_BANK
- LDC Rm, R1_BANK
- LDC Rm, R2_BANK
- LDC Rm, R3_BANK
- LDC Rm, R4_BANK
- LDC Rm, R5_BANK
- LDC Rm, R6_BANK
- LDC Rm, R7_BANK
- LDS Rm, PR
- LDTLB
- NOP
- SETS
- SETT
- STC SR, Rn
- STC GBR, Rn
- STC VBR, Rn
- STC SSR, Rn
- STC SPC, Rn
- STC R0_BANK, Rn
- STC R1_BANK, Rn
- STC R2_BANK, Rn
- STC R3_BANK, Rn
- STC R4_BANK, Rn
- STC R5_BANK, Rn
- STC R6_BANK, Rn
- STC R7_BANK, Rn
- STS PR, Rn

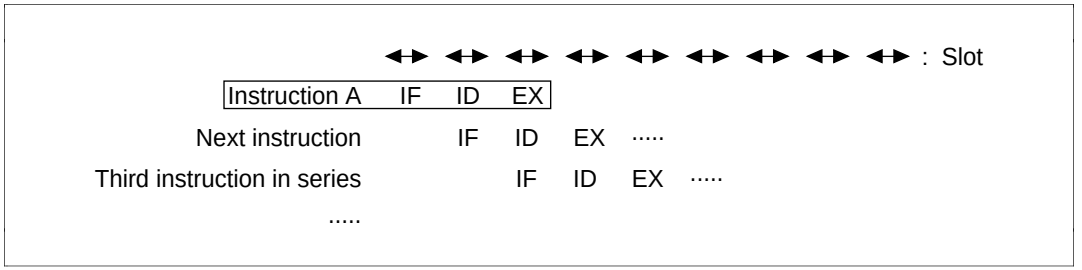


Figure 8.29 System Control ALU Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.29). The data operation is completed in the EX stage via the ALU.

LDC.L Instructions (SR): Instruction types:

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

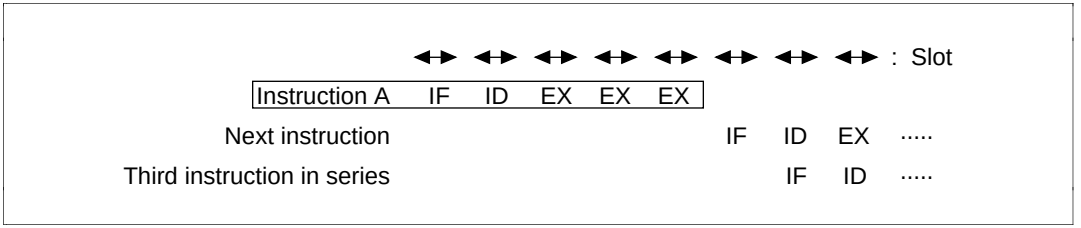


Figure 8.30 LDC, Rm, SR Instruction Pipeline

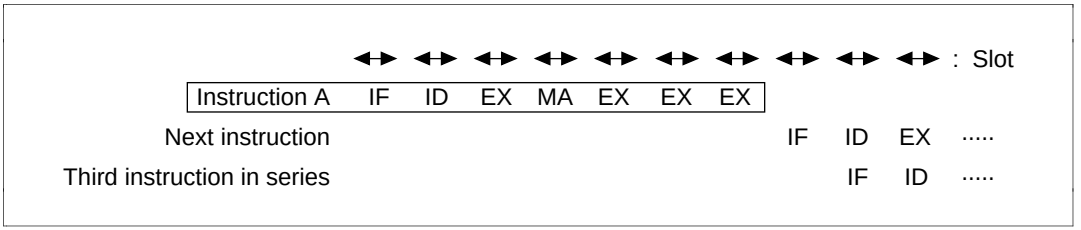


Figure 8.31 LDC.L @Rm+, SR Instruction Pipeline

The IF stage of the next instruction starts after the last EX stage of instruction A is completed.

LDC.L Instructions: Instruction types:

- LDC.L @Rm+, GBR
- LDC.L @Rm+, VBR
- LDC.L @Rm+, SSR
- LDC.L @Rm+, SPC
- LDC.L @Rm+, R0_BANK
- LDC.L @Rm+, R1_BANK
- LDC.L @Rm+, R2_BANK
- LDC.L @Rm+, R3_BANK
- LDC.L @Rm+, R4_BANK
- LDC.L @Rm+, R5_BANK
- LDC.L @Rm+, R6_BANK
- LDC.L @Rm+, R7_BANK

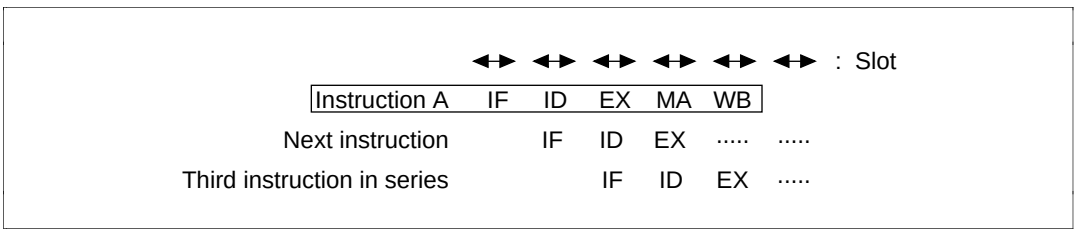


Figure 8.32 LDC.L Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB.

STC.L Instructions:

Include the following instruction types for the pipeline shown in figure 8.33:

- STC.L SR, @–Rn
- STC.L GBR, @–Rn
- STC.L VBR, @–Rn
- STC.L SSR, @–Rn
- STC.L SPC, @–Rn

Include the following instruction types for the pipeline shown in figure 8.34:

- STC.L R0_BANK,@–Rn
- STC.L R1_BANK,@–Rn
- STC.L R2_BANK,@–Rn
- STC.L R3_BANK,@–Rn
- STC.L R4_BANK,@–Rn
- STC.L R5_BANK,@–Rn
- STC.L R6_BANK,@–Rn
- STC.L R7_BANK,@–Rn

	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄►	◄► : Slot
Instruction A	IF	ID	EX	MA					
Next instruction		IF	ID	EX	·····	·····			
Third instruction in series			IF	ID	EX	·····	·····		
·····									

Figure 8.33 STC.L Instruction Pipeline (1)

The STC.L instruction pipeline shown in figure 8.33 has four stages: IF, ID, EX, and MA.

	◄►	◄►	◄►	◄►	◄►	: Slot
Instruction A	IF	ID	EX	EX	MA	
Next instruction		IF	ID	EX	·····	·····
Third instruction in series			IF	ID	EX	·····

Figure 8.34 STC.L Instruction Pipeline (2)

The STC.L instruction pipeline shown in figure 8.34 has five stages: IF, ID, EX, EX, and MA.

LDS.L Instruction (PR): Instruction type:

- LDS.L @Rm+, PR

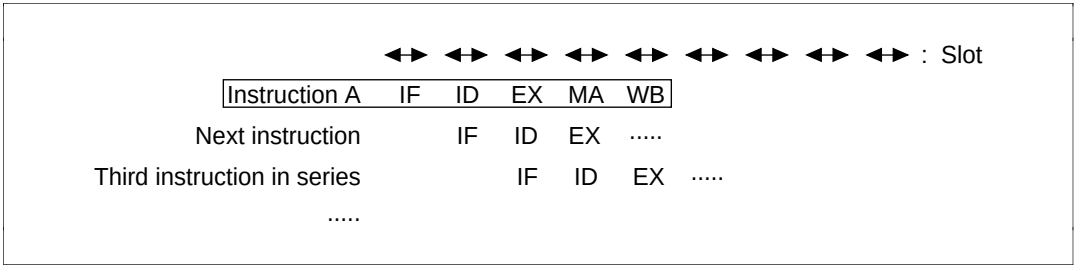


Figure 8.35 LDS.L Instructions (PR) Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.35). It is the same as an ordinary load instruction.

STS.L Instruction (PR): Instruction type:

- STS.L PR, @–Rn

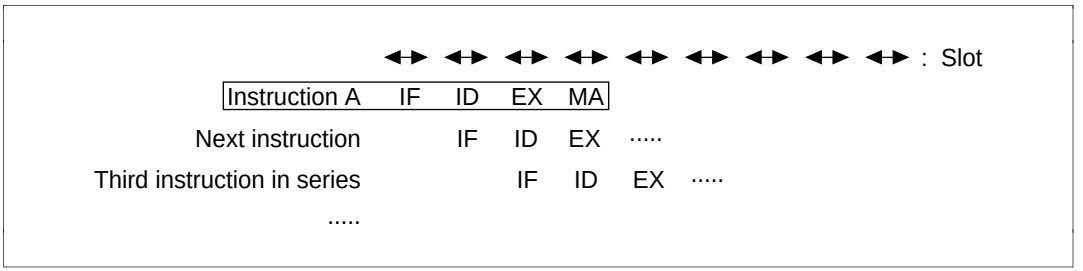


Figure 8.36 STS.L Instruction (PR) Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.36). It is the same as an ordinary load instruction.

Register → MAC Transfer Instructions: Instruction types:

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

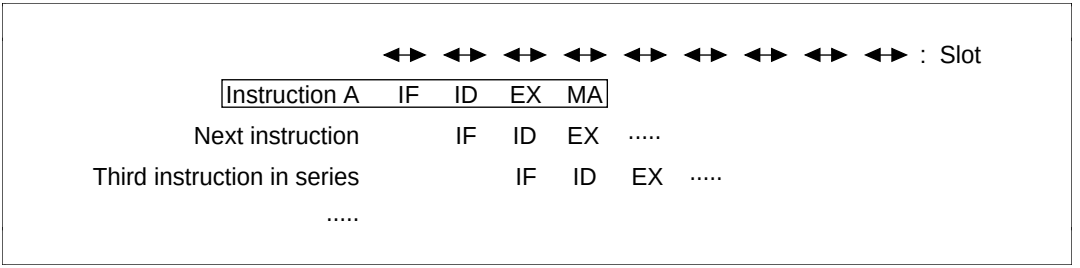


Figure 8.37 Register → MAC Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.37). 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 MAC and MUL instructions apply.

Memory → MAC Transfer Instructions: Instruction types:

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

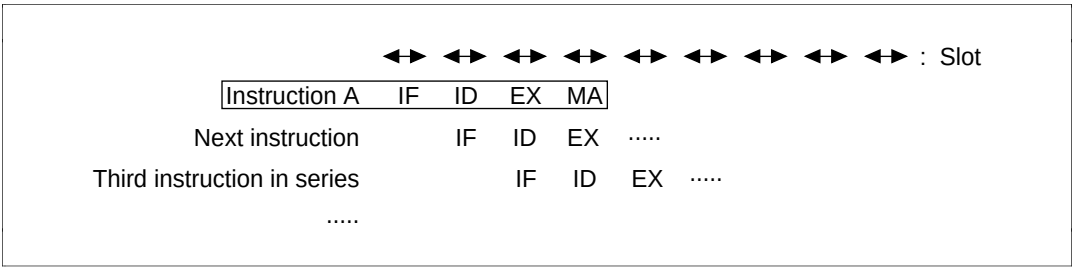


Figure 8.38 Memory → MAC Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.38). 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 MAC and MUL instructions apply.

MAC → Register Transfer Instructions: Instruction types:

- STS MACH, Rn
- STS MACL, Rn

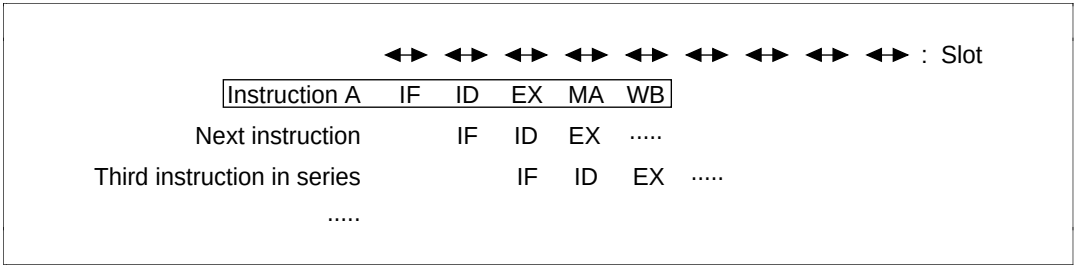


Figure 8.39 MAC → Register Transfer Instruction Pipeline

The pipeline has five stages: IF, ID, EX, MA, and WB (figure 8.39). 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 MAC and MUL instructions apply.

MAC → Memory Transfer Instructions: Instruction types:

- STS.L MACH, @–Rn
- STS.L MACL, @–Rn

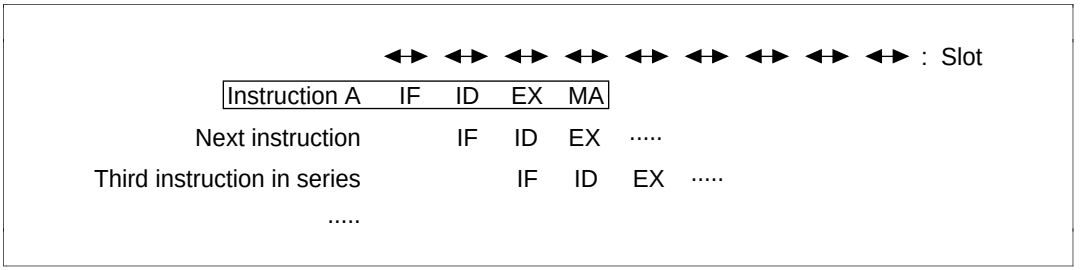


Figure 8.40 MAC → Memory Transfer Instruction Pipeline

The pipeline has four stages: IF, ID, EX, and MA (figure 8.40). 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 MAC and MUL instructions apply.

RTE Instruction: Instruction type:

- RTE

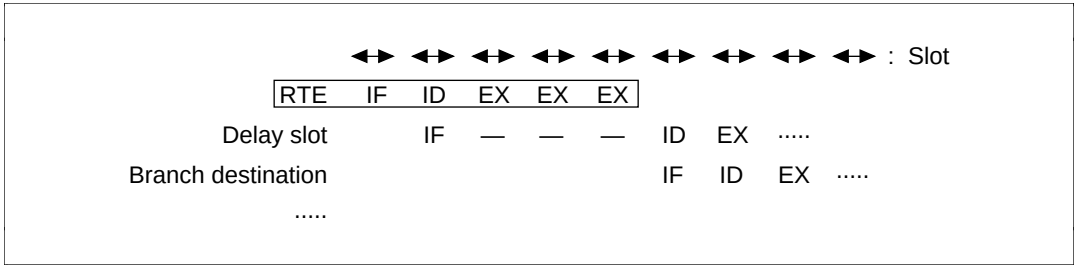


Figure 8.41 RTE Instruction Pipeline

The pipeline has five stages: IF, ID, EX, EX, and EX (figure 8.41). RTE is a delayed branch instruction. The ID of the delay slot instruction is stalled 3 slots. The IF of the branch destination instruction starts from the slot following the last EX of the RTE.

TRAP Instruction: Instruction type:

- TRAPA #imm

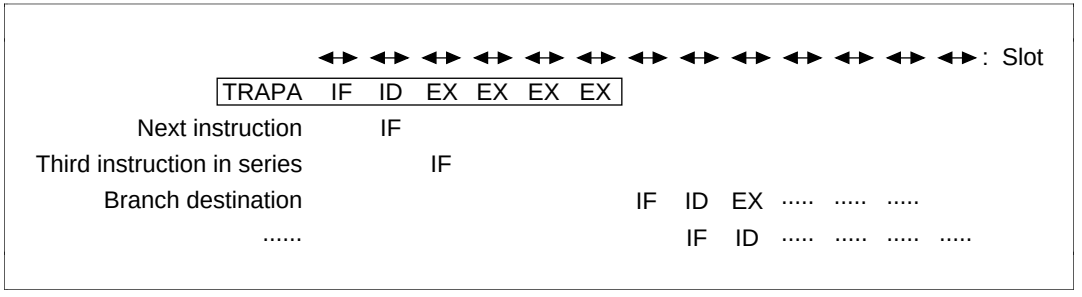


Figure 8.42 TRAP Instruction Pipeline

The pipeline has six stages: IF, ID, EX, EX, EX, and EX (figure 8.42). TRAP is not a delayed branch instruction. The two instructions after the TRAP instruction are fetched, but they are discarded without being executed. The IF of the branch destination instruction starts from the next slot of the last EX of the TRAP instruction.

SLEEP Instruction: Instruction type:

- SLEEP

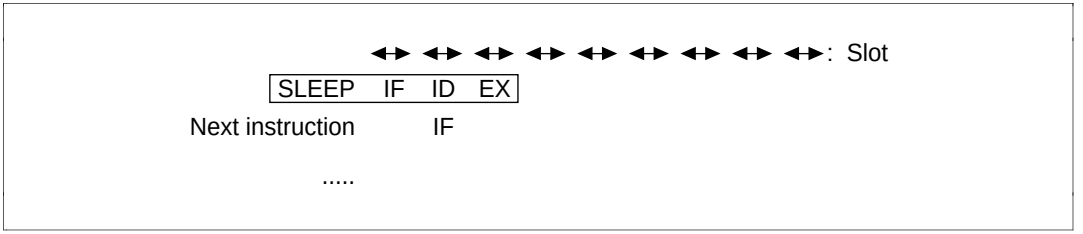


Figure 8.43 SLEEP Instruction Pipeline

The pipeline has three stages: IF, ID, and EX (figure 8.43). 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: Instruction type:

- Interrupt exception processing

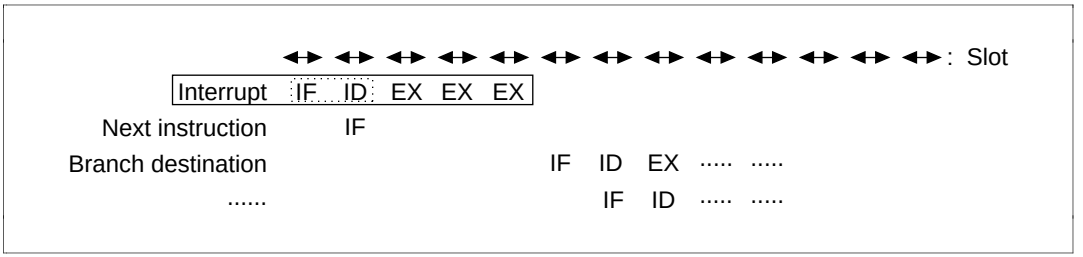


Figure 8.44 Interrupt Exception Processing Pipeline

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 five stages: IF, ID, EX, EX, and EX (figure 8.44). 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 following the final EX in the interrupt exception processing.

Interrupt sources are NMI, user break, IRQ, and on-chip peripheral module interrupts.

Address Error Exception Processing: Instruction type:

- Address error exception processing

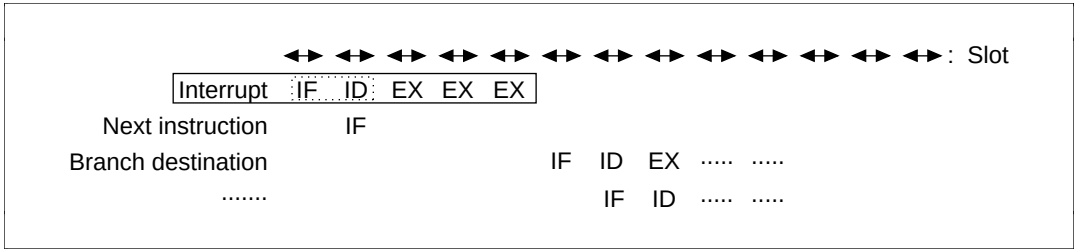


Figure 8.45 Address Error Exception Processing Pipeline

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 five stages: IF, ID, EX, EX, and EX (figure 8.45). 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 following the final EX in the address error exception processing.

Address errors are caused by instruction fetches and by data reads or writes. Fetching an instruction from an odd address or fetching an instruction from an on-chip peripheral register causes an instruction fetch address error. Accessing word data from other than a word boundary, accessing longword data from other than a longword boundary, and accessing an on-chip peripheral register 8-bit space by longword cause a read or write address error.

Illegal Instruction Exception Processing: Instruction type:

- Illegal instruction exception processing

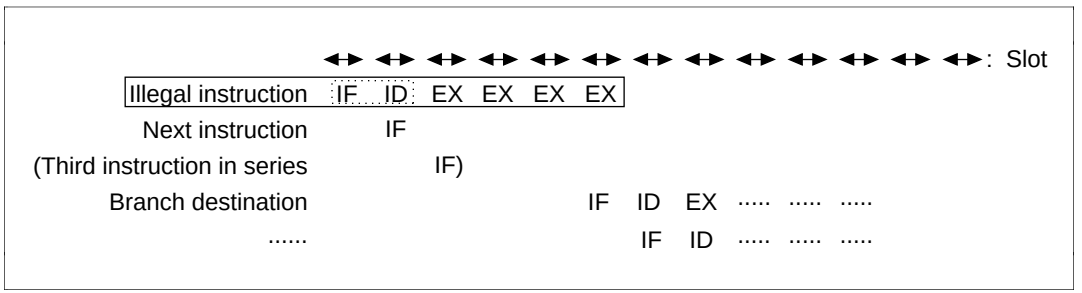


Figure 8.46 Illegal Instruction Exception Processing Pipeline

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 six stages: IF, ID, EX, EX, EX, and EX (figure 8.46). Illegal instruction exception processing is not a delayed branch. In illegal instruction exception processing, an overrun fetch (IF) occurs. 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 following 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.

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		11
Direct register addressing	Destination operand only	MOVT	Rn	18
	Source and destination operand	ADD	Rm,Rn	36
	Load and store with control register or system register	LDC STS	Rm,SR MACH,Rn	32
Indirect register addressing	Destination operand only	JMP	@Rn	4
	Data transfer direct from register	MOV.L	Rm,@Rn	6
Post-increment indirect register addressing	Multiply/accumulate operation	MAC.W	@Rm+,@Rn+	2
	Data transfer direct from register	MOV.L	@Rm+,Rn	3
	Load to control register or system register	LDC.L	@Rm+,SR	16
Pre-decrement indirect register addressing	Data transfer direct from register	MOV.L	Rm,@–Rn	3
	Store from control register or system register	STC.L	SR,@–Rn	16
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)	6
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
PC relative addressing with Rn	Branch instruction	BRAF	Rn	2
PC relative addressing	Branch instruction	BRA	disp	6
Immediate addressing	Arithmetic logical operations direct with register	ADD	#imm,Rn	7
	Specify exception processing vector	TRAPA	#imm	1
			Total:	188

A.1.1 No Operand

Table A.2 No Operand

Instruction	Operation	Code	Cycles	T Bit
CLRS	0 → S	0000000001001000	1	—
CLRT	0 → T	0000000000001000	1	0
CLRMAC	0 → MACH, MACL	0000000000101000	1	—
DIV0U	0 → M/Q/T	0000000000011001	1	0
LDTLB	PTEH/PTEL → TLB	0000000000111000	1	—
NOP	No operation	0000000000001001	1	—
RTE	Delayed branching, SSR/SPC → SR/PC	0000000000101011	4	—
RTS	Delayed branching, PR → PC	0000000000001011	2	—
SETS	1 → S	0000000001011000	1	—
SETT	1 → T	0000000000011000	1	1
SLEEP	Sleep	0000000000011011	4	—

A.1.2 Direct Register Addressing

Table A.3 Destination Operand Only

Instruction		Operation	Code	Cycles	T Bit
CMP/PL	Rn	Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
DT	Rn	Rn – 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLR	Rn	0 → Rn → T	0100nnnn00000001	1	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—

Table A.4 Source and Destination Operand

Instruction		Operation	Code	Cycles	T Bit
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	Overflow
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
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
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
CMP/STR	Rm,Rn	When a byte in Rn equals a bytes in Rm, 1 →T	0010nnnnmmmm1100	1	Comparison result
DIV1	Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	1	Calculation result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q →T	0010nnnnmmmm0111	1	Calculation result
DMULS.L	Rm,Rn	Signed operation of Rn x Rm → MACH, MACL	0011nnnnmmmm1101	2 (to 5)*	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 (to 5)*	—
EXTS.B	Rm,Rn	Sign – extend Rm from byte → Rn	0110nnnnmmmm1110	1	—
EXTS.W	Rm,Rn	Sign – extend Rm from word → Rn	0110nnnnmmmm1111	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	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	1	—

Table A.4 Source and Destination Operand (cont)

Instruction		Operation	Code	Cycles	T Bit
MUL.L	Rm,Rn	Rn × Rm → MAC	0000nnnnmmmm0111	2 (to 5)*	—
MULS.W	Rm,Rn	With sign, Rn × Rm → MAC	0010nnnnmmmm1111	1 (to 3)*	—
MULU.W	Rm,Rn	Unsigned, Rn × Rm → MAC	0010nnnnmmmm1110	1 (to 3)*	—
NEG	Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—
NEGC	Rm,Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (MSB→)Rn	0100nnnnmmmm1100	1	—
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (0→)Rn	0100nnnnmmmm1101	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 upper and lower halves of lower 2 bytes → Rn	0110nnnnmmmm1000	1	—
SWAP.W	Rm,Rn	Rm → Swap upper and lower word → Rn	0110nnnnmmmm1001	1	—
TST	Rm,Rn	Rn & Rm, when result is 0, 1 →T	0010nnnnmmmm1000	1	Test results
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XTRCT	Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

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.5 Load and Store with Control Register or System Register

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	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	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	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	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 Destination Operand Only

Instruction		Operation	Code	Cycles	T Bit
JMP	@Rn	Delayed branching, Rn → PC	0100nnnn00101011	2	—
JSR	@Rn	Delayed branching, PC → Rn, Rn → PC	0100nnnn00001011	2	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—
TAS.B	@Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	3	Test results

Table A.7 Data Transfer Direct to 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	—

A.1.4 Post-Increment Indirect Register Addressing

Table A.8 Multiply/Accumulate Operation

Instruction		Operation	Code	Cycles	T Bit
MAC.L	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	2 (to 5)*	—
MAC.W	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	2 (to 5)*	—

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.9 Data Transfer Direct from 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.10 Load to Control Register or System Register

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	1	LSB
LDC.L @Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
LDC.L @Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1	—
LDC.L @Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC.L @Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC.L @Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC.L @Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC.L @Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC.L @Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—
LDC.L @Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC.L @Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC.L @Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC.L @Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	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.11 Data Transfer Direct from 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.12 Store from Control Register or System 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	—
STC.L SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
STC.L SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—
STC.L R0_BANK, @–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L R1_BANK, @–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L R2_BANK, @–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L R3_BANK, @–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L R4_BANK, @–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L R5_BANK, @–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L R6_BANK, @–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L R7_BANK, @–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—
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.13 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.14 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	—
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	—

A.1.8 Indirect GBR Addressing with Displacement

Table A.15 Indirect GBR Addressing 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 + GBR)	11000001dddddddd	1	—
MOV.L	R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	1	—
MOV.B	@(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100dddddddd	1	—
MOV.W	@(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000101dddddddd	1	—
MOV.L	@(disp,GBR),R0	(disp + GBR) → R0	11000110dddddddd	1	—

A.1.9 Indirect Indexed GBR Addressing

Table A.16 Indirect Indexed GBR Addressing

Instruction		Operation	Code	Cycles	T Bit
AND.B	#imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	3	—
OR.B	#imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—
TST.B	#imm,@(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	3	Test results
XOR.B	#imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—

A.1.10 PC Relative Addressing with Displacement

Table A.17 PC Relative Addressing with Displacement

Instruction		Operation	Code	Cycles	T Bit
MOV.W	@(disp,PC),Rn	(disp + PC) → sign extension → Rn	1001nnnndddddddd	1	—
MOV.L	@(disp,PC),Rn	(disp + PC) → Rn	1101nnnndddddddd	1	—
MOVA	@(disp,PC),R0	disp + PC → R0	11000111dddddddd	1	—

A.1.11 PC Relative Addressing

Table A.18 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.19 PC Relative Addressing

Instruction		Operation	Code	Cycles	T Bit
BF	disp	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	disp	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	disp	Delayed branching, disp + PC → PC	1010dddddddddddd	2	—
BSR	disp	Delayed branching, PC → PR, disp + PC → PC	1011dddddddddddd	2	—

Note: One state when it does not branch.

A.1.12 Immediate

Table A.20 Arithmetic Logical Operations Direct with Register

Instruction		Operation	Code	Cycles	T Bit
ADD	#imm,Rn	Rn + #imm → Rn	0111nnnniiiiiiii	1	—
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
CMP/EQ	#imm,R0	When R0 = imm, 1 → T	10001000iiiiiiii	1	Comparison result
MOV	#imm,Rn	#imm → sign extension → Rn	1110nnnniiiiiiii	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
TST	#imm,R0	R0 & imm, when result is 0, 1 →T	11001000iiiiiiii	1	Test results
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—

Table A.21 Specify Exception Processing Vector

Instruction	Operation	Code	Cycles	T Bit
TRAPA #imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB, 0x160 → EXPEVT VBR + H'00000100 → PC	11000011iiiiiiii	6	—

A.2 Instruction Sets by Instruction Format

Tables A.22 to A.48 list instruction codes and execution cycles by instruction formats.

Table A.22 Instruction Sets by Format

Format	Category	Sample Instruction		Types
0	—	NOP		11
n	Direct register addressing	MOVT	Rn	18
	Direct register addressing (store with control or system registers)	STS	MACH,Rn	16
	Direct register addressing	JMP	@Rn	4
	Pre-decrement indirect register addressing	STC.L	SR,@–Rn	16
	PC relative addressing with Rn	BRAF	Rn	2
m	Direct register addressing (load with control or system registers)	LDC	Rm,SR	16
	Post-increment indirect register addressing	LDC.L	@Rm+,SR	16
nm	Direct register addressing	ADD	Rm,Rn	36
	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
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
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

Table A.22 Instruction Sets by Format (cont)

Format	Category	Sample Instruction		Types
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:	188

A.2.1 0 Format

Table A.23 0 Format

Instruction	Operation	Code	Cycles	T Bit
CLRS	0 →S	0000000001001000	1	—
CLRT	0 →T	0000000000001000	1	0
CLRMAC	0 → MACH, MACL	0000000000101000	1	—
DIV0U	0 → M/Q/T	0000000000011001	1	0
LDTLB	PTEH/PTEL → TLB	0000000000111000	1	—
NOP	No operation	0000000000001001	1	—
RTE	Delayed branch, SSR/SPC → SR/PC	0000000000101011	4	—
RTS	Delayed branching, PR → PC	0000000000001011	2	—
SETS	1 →S	0000000001011000	1	—
SETT	1 →T	0000000000011000	1	1
SLEEP	Sleep	0000000000011011	4	—

A.2.2 n Format

Table A.24 Direct Register

Instruction		Operation	Code	Cycles	T Bit
CMP/PL	Rn	Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
CMP/PZ	Rn	Rn ≥ 0, 1 → T	0100nnnn00010001	1	Comparison result
DT	Rn	Rn – 1 → Rn, when Rn is 0, 1 → T. When Rn is nonzero, 0 → T	0100nnnn00010000	1	Comparison result
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
ROTL	Rn	T ← Rn ← MSB	0100nnnn00000100	1	MSB
ROTR	Rn	LSB → Rn → T	0100nnnn00000101	1	LSB
ROTCL	Rn	T ← Rn ← T	0100nnnn00100100	1	MSB
ROTCR	Rn	T → Rn → T	0100nnnn00100101	1	LSB
SHAL	Rn	T ← Rn ← 0	0100nnnn00100000	1	MSB
SHAR	Rn	MSB → Rn → T	0100nnnn00100001	1	LSB
SHLL	Rn	T ← Rn ← 0	0100nnnn00000000	1	MSB
SHLR	Rn	0 → Rn → T	0100nnnn00000001	1	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
SHLL8	Rn	Rn << 8 → Rn	0100nnnn00011000	1	—
SHLR8	Rn	Rn >> 8 → Rn	0100nnnn00011001	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—

Table A.25 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	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	1	—
STS	MACH,Rn	MACH → Rn	0000nnnn00001010	1	—
STS	MACL,Rn	MACL → Rn	0000nnnn00011010	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	1	—

Table A.26 Indirect Register

Instruction		Operation	Code	Cycles	T Bit
JMP	@Rn	Delayed branching, Rn → PC	0100nnnn00101011	2	—
JSR	@Rn	Delayed branching, PC → Rn, Rn → PC	0100nnnn00001011	2	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—
TAS.B	@Rn	When (Rn) is 0, 1 → T, 1 → MSB of (Rn)	0100nnnn00011011	3	Test results

Table A.27 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	—
STC.L SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
STC.L SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—
STC.L R0_BANK, @–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
STC.L R1_BANK, @–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
STC.L R2_BANK, @–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
STC.L R3_BANK, @–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
STC.L R4_BANK, @–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
STC.L R5_BANK, @–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
STC.L R6_BANK, @–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—
STC.L R7_BANK, @–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—
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	—

Table A.28 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	—

A.2.3 m Format

Table A.29 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	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
LDC	Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
LDC	Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
LDC	Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
LDC	Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
LDC	Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
LDC	Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
LDC	Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
LDS	Rm,MACL	Rm → MACL	0100mmmm00011010	1	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	1	—

Table A.30 Indirect Post-Increment Register

Instruction		Operation	Code	Cycles	T Bit
LDC.L	@Rm+,SR	(Rm) → SR, Rm + 4 → Rm	0100mmmm00000111	1	LSB
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
LDC.L	@Rm+,VBR	(Rm) → VBR, Rm + 4 → Rm	0100mmmm00100111	1	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC.L	@Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC.L	@Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC.L	@Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC.L	@Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC.L	@Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—
LDC.L	@Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC.L	@Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC.L	@Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC.L	@Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	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.2.4 nm Format

Table A.31 Direct Register

Instruction		Operation	Code	Cycles	T Bit
ADD	Rm,Rn	Rm + Rn → 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	Overflow
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
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
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
CMP/STR	Rm,Rn	When a byte in Rn equals a byte in Rm, 1 → T	0010nnnnmmmm1100	1	Comparison result
DIV1	Rm,Rn	1 step division (Rn ÷ Rm)	0011nnnnmmmm0100	1	Calculation result
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q →T	0010nnnnmmmm0111	1	Calculation result
DMULS.L	Rm,Rn	Signed operation of Rn × Rm → MACH, MACL	0011nnnnmmmm1101	2 (to 5)*	—
DMULU.L	Rm,Rn	Unsigned operation of Rn × Rm → MACH, MACL	0011nnnnmmmm0101	2 (to 5)*	—
EXTS.B	Rm,Rn	Sign-extend Rm from byte → Rn	0110nnnnmmmm1110	1	—

Table A.31 Direct Register (cont)

Instruction		Operation	Code	Cycles	T Bit
EXTS.W	Rm,Rn	Sign-extend Rm from word → Rn	0110nnnnmmmm1111	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	—
MOV	Rm,Rn	Rm → Rn	0110nnnnmmmm0011	1	—
MUL.L	Rm,Rn	Rn × Rm → MAC	0000nnnnmmmm0111	2 (to 5)*	—
MULS	Rm,Rn	With sign, Rn × Rm → MAC	0010nnnnmmmm1111	1 (to 3)*	—
MULU	Rm,Rn	Unsigned, Rn × Rm → MAC	0010nnnnmmmm1110	1 (to 3)*	—
NEG	Rm,Rn	0 – Rm → Rn	0110nnnnmmmm1011	1	—
NEGC	Rm,Rn	0 – Rm – T → Rn, Borrow → T	0110nnnnmmmm1010	1	Borrow
NOT	Rm,Rn	~Rm → Rn	0110nnnnmmmm0111	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
SHAD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (MSB→)Rn	0100nnnnmmmm1100	1	—
SHLD	Rm,Rn	Rn ≥ 0; Rn << Rm → Rn Rn < 0; Rn >> Rm → (0→)Rn	0100nnnnmmmm1101	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	Under-flow
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	—
TST	Rm,Rn	Rn & Rm, when result is 0, 1 →T	0010nnnnmmmm1000	1	Test results
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
XTRCT	Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—

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.32 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.33 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	2 (to 5)*	—
MAC.W @Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	2 (to 5)*	—

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.34 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.35 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.36 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	—

A.2.5 md Format

Table A.37 md Format

Instruction	Operation	Code	Cycles	T Bit
`MOV.B @(disp,Rm),R0`	(disp + Rm) → sign extension → R0	`10000100mmmmdddd`	1	—
`MOV.W @(disp,Rm),R0`	(disp + Rm) → sign extension → R0	`10000101mmmmdddd`	1	—

A.2.6 nd4 Format

Table A.38 nd4 Format

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	—

A.2.7 nmd Format

Table A.39 nmd Format

Instruction	Operation	Code	Cycles	T Bit
`MOV.L Rm,@(disp,Rn)`	Rm → (disp + Rn)	`0001nnnnmmmmdddd`	1	—
`MOV.L @(disp,Rm),Rn`	(disp + Rm) → Rn	`0101nnnnmmmmdddd`	1	—

A.2.8 d Format

Table A.40 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 + GBR)	11000001dddddddd	1	—
MOV.L	R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	1	—
MOV.B	@(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100dddddddd	1	—
MOV.W	@(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000101dddddddd	1	—
MOV.L	@(disp,GBR),R0	(disp + GBR) → R0	11000110dddddddd	1	—

Table A.41 PC Relative with Displacement

Instruction		Operation	Code	Cycles	T Bit
MOVA	@(disp,PC),R0	disp + PC → R0	11000111dddddddd	1	—

Table A.42 PC Relative

Instruction		Operation	Code	Cycles	T Bit
BF	disp	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	disp	When T = 1, disp + PC → PC; when T = 0, nop	10001001dddddddd	3/1	—
BT/S	label	If T = 1, disp + PC → PC; if T = 0, nop	10001101dddddddd	2/1*	

Note: One state when it does not branch.

A.2.9 d12 Format

Table A.43 d12 Format

Instruction	Operation	Code	Cycles	T Bit
`BRA disp`	Delayed branching, disp + PC → PC	`1010ddddddddddd`	2	—
`BSR disp`	Delayed branching, PC → PR, disp + PC → PC	`1011ddddddddddd`	2	—

A.2.10 nd8 Format

Table A.44 nd8 Format

Instruction	Operation	Code	Cycles	T Bit
`MOV.W @(disp,PC),Rn`	(disp + PC) → sign extension → Rn	`1001nnnnddddddd`	1	—
`MOV.L @(disp,PC),Rn`	(disp + PC) → Rn	`1101nnnnddddddd`	1	—

A.2.11 i Format

Table A.45 Indirect Indexed GBR

Instruction	Operation	Code	Cycles	T Bit
`AND.B #imm,@(R0,GBR)`	(R0 + GBR) & imm → (R0 + GBR)	`11001101iiiiiiii`	3	—
`OR.B #imm,@(R0,GBR)`	(R0 + GBR) | imm → (R0 + GBR)	`11001111iiiiiiii`	3	—
`TST.B #imm,@(R0,GBR)`	(R0 + GBR) & imm, when result is 0, 1 → T	`11001100iiiiiiii`	3	Test results
`XOR.B #imm,@(R0,GBR)`	(R0 + GBR) ^ imm → (R0 + GBR)	`11001110iiiiiiii`	3	—

Table A.46 Immediate (Arithmetic Logical Operation with Direct Register)

Instruction		Operation	Code	Cycles	T Bit
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
CMP/EQ	#imm,R0	When R0 = imm, 1 → T	10001000iiiiiiii	1	Comparison results
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
TST	#imm,R0	R0 & imm, when result is 0, 1 →T	11001000iiiiiiii	1	Test results
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—

Table A.47 Immediate (Specify Exception Processing Vector)

Instruction		Operation	Code	Cycles	T Bit
TRAPA	#imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB, 0x160 → EXPEVT VBR + H'00000100 → PC	11000011iiiiiiii	6	—

A.2.12 ni Format

Table A.48 ni Format

Instruction		Operation	Code	Cycles	T Bit
ADD	#imm,Rn	Rn + #imm → Rn	0111nnnniiiiiiii	1	—
MOV	#imm,Rn	#imm → sign extension → Rn	1110nnnniiiiiiii	1	—

A.3 Instruction Set by Instruction Code

Table A.49 lists instruction codes and execution cycles by instruction code.

Table A.49 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
DIV0U		0 → M/Q/T	0000000000011001	1	0
SLEEP		Sleep	0000000000011011	4	—
CLRMAC		0 → MACH, MACL	0000000000101000	1	—
RTE		Delayed branch, SSR/SPC → SR/PC	0000000000101011	4	—
LDTLB		PTEH/PTEL → TLB	0000000000111000	1	—
CLRS		0 → S	0000000001001000	1	—
SETS		1 → S	0000000001011000	1	—
STC	SR, Rn	SR → Rn	0000nnnn00000010	1	—
BSRF	Rn	Delayed branch, PC → PR, Rn + PC → PC	0000nnnn00000011	2	—

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
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	Rn	Delayed branch, Rn + PC → PC	0000nnnn00100011	2	—
MOVT	Rn	T → Rn	0000nnnn00101001	1	—
STS	PR,Rn	PR → Rn	0000nnnn00101010	1	—
STC	SSR,Rn	SSR → Rn	0000nnnn00110010	1	—
STC	SPC,Rn	SPC → Rn	0000nnnn01000010	1	—
STC	R0_BANK,Rn	R0_BANK→ Rn	0000nnnn10000010	1	—
PREF	@Rn	(Rn) → cache	0000nnnn10000011	1	—
STC	R1_BANK,Rn	R1_BANK→ Rn	0000nnnn10010010	1	—
STC	R2_BANK,Rn	R2_BANK→ Rn	0000nnnn10100010	1	—
STC	R3_BANK,Rn	R3_BANK→ Rn	0000nnnn10110010	1	—
STC	R4_BANK,Rn	R4_BANK→ Rn	0000nnnn11000010	1	—
STC	R5_BANK,Rn	R5_BANK→ Rn	0000nnnn11010010	1	—
STC	R6_BANK,Rn	R6_BANK→ Rn	0000nnnn11100010	1	—
STC	R7_BANK,Rn	R7_BANK→ Rn	0000nnnn11110010	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	—
MAC.L	@Rm+,@Rn+	Signed operation of (Rn) × (Rm) + MAC → MAC	0000nnnnmmmm1111	2 (to 5)*1	—
MOV.L	Rm,@(disp,Rn)	Rm → (disp + Rn)	0001nnnnmmmmdddd	1	—
MOV.B	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0000	1	—
MOV.W	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0001	1	—

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
MOV.L	Rm,@Rn	Rm → (Rn)	0010nnnnmmmm0010	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	—
DIV0S	Rm,Rn	MSB of Rn → Q, MSB of Rm → M, M ^ Q →T	0010nnnnmmmm0111	1	Calculation result
TST	Rm,Rn	Rn & Rm, when result is 0, 1 →T	0010nnnnmmmm1000	1	Test results
AND	Rm,Rn	Rn & Rm → Rn	0010nnnnmmmm1001	1	—
XOR	Rm,Rn	Rn ^ Rm → Rn	0010nnnnmmmm1010	1	—
OR	Rm,Rn	Rn | Rm → Rn	0010nnnnmmmm1011	1	—
CMP/STR	Rm,Rn	When a byte in Rn equals a byte in Rm, 1 → T	0010nnnnmmmm1100	1	Comparison result
XTRCT	Rm,Rn	Rm: Center 32 bits of Rn → Rn	0010nnnnmmmm1101	1	—
MULU	Rm,Rn	Unsigned, Rn × Rm → MAC	0010nnnnmmmm1110	1 (to 3)*1	—
MULS	Rm,Rn	Signed, Rn × Rm → MAC	0010nnnnmmmm1111	1 (to 3)*1	—
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 5)*1	—
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

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
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 5)*1	—
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	—
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	7	LSB
SHLL2	Rn	Rn << 2 → Rn	0100nnnn00001000	1	—
SHLR2	Rn	Rn >> 2 → Rn	0100nnnn00001001	1	—
LDS	Rm,MACH	Rm → MACH	0100mmmm00001010	1	—
JSR	@Rn	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	Compariso n result
CMP/PZ	Rn	Rn ≥ 0, 1 → T	0100nnnn00010001	1	Compariso n result
STS.L	MACL,@–Rn	Rn – 4 → Rn, MACL → (Rn)	0100nnnn00010010	1	—
STC.L	GBR,@-Rn	Rn – 4 → Rn, GBR → (Rn)	0100nnnn00010011	1	—

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
CMP/PL	Rn	Rn > 0, 1 → T	0100nnnn00010101	1	Comparison result
LDS.L	@Rm+,MACL	(Rm) → MACL, Rm + 4 → Rm	0100mmmm00010110	1	—
LDC.L	@Rm+,GBR	(Rm) → GBR, Rm + 4 → Rm	0100mmmm00010111	1	—
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	3	Test results
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	1	—
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	1	—
SHLL16	Rn	Rn << 16 → Rn	0100nnnn00101000	1	—
SHLR16	Rn	Rn >> 16 → Rn	0100nnnn00101001	1	—
LDS	Rm,PR	Rm → PR	0100mmmm00101010	1	—
JMP	@Rn	Delayed branching, Rn → PC	0100nnnn00101011	2	—
LDC	Rm,VBR	Rm → VBR	0100mmmm00101110	1	—
STC.L	SSR,@–Rn	Rn–4 → Rn, SSR → (Rn)	0100nnnn00110011	1	—
LDC.L	@Rm+,SSR	(Rm) → SSR, Rm + 4 → Rm	0100mmmm00110111	1	—
LDC	Rm,SSR	Rm → SSR	0100mmmm00111110	1	—
STC.L	SPC,@–Rn	Rn–4 → Rn, SPC → (Rn)	0100nnnn01000011	1	—

Table A.49 Instruction Set by Instruction Code (cont)

Instruction	Operation	Code	Cycles	T Bit
LDC.L @Rm+,SPC	(Rm) → SPC, Rm + 4 → Rm	0100mmmm01000111	1	—
LDC Rm,SPC	Rm → SPC	0100mmmm01001110	1	—
STC.L R0_BANK,@–Rn	Rn–4 → Rn, R0_BANK → (Rn)	0100nnnn10000011	2	—
LDC.L @Rm+,R0_BANK	(Rm) → R0_BANK, Rm + 4 → Rm	0100mmmm10000111	1	—
LDC Rm,R0_BANK	Rm → R0_BANK	0100mmmm10001110	1	—
STC.L R1_BANK,@–Rn	Rn–4 → Rn, R1_BANK → (Rn)	0100nnnn10010011	2	—
LDC.L @Rm+,R1_BANK	(Rm) → R1_BANK, Rm + 4 → Rm	0100mmmm10010111	1	—
LDC Rm,R1_BANK	Rm → R1_BANK	0100mmmm10011110	1	—
STC.L R2_BANK,@–Rn	Rn–4 → Rn, R2_BANK → (Rn)	0100nnnn10100011	2	—
LDC.L @Rm+,R2_BANK	(Rm) → R2_BANK, Rm + 4 → Rm	0100mmmm10100111	1	—
LDC Rm,R2_BANK	Rm → R2_BANK	0100mmmm10101110	1	—
STC.L R3_BANK, @–Rn	Rn–4 → Rn, R3_BANK → (Rn)	0100nnnn10110011	2	—
LDC.L @Rm+,R3_BANK	(Rm) → R3_BANK, Rm + 4 → Rm	0100mmmm10110111	1	—
LDC Rm,R3_BANK	Rm → R3_BANK	0100mmmm10111110	1	—
STC.L R4_BANK,@–Rn	Rn–4 → Rn, R4_BANK → (Rn)	0100nnnn11000011	2	—
LDC.L @Rm+,R4_BANK	(Rm) → R4_BANK, Rm + 4 → Rm	0100mmmm11000111	1	—
LDC Rm,R4_BANK	Rm → R4_BANK	0100mmmm11001110	1	—
STC.L R5_BANK,@–Rn	Rn–4 → Rn, R5_BANK → (Rn)	0100nnnn11010011	2	—
LDC.L @Rm+,R5_BANK	(Rm) → R5_BANK, Rm + 4 → Rm	0100mmmm11010111	1	—
LDC Rm,R5_BANK	Rm → R5_BANK	0100mmmm11011110	1	—
STC.L R6_BANK,@–Rn	Rn–4 → Rn, R6_BANK → (Rn)	0100nnnn11100011	2	—

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
LDC.L	@Rm+,R6_BANK	(Rm) → R6_BANK, Rm + 4 → Rm	0100mmmm11100111	1	—
LDC	Rm,R6_BANK	Rm → R6_BANK	0100mmmm11101110	1	—
STC.L	R7_BANK,@–Rn	Rn–4 → Rn, R7_BANK → (Rn)	0100nnnn11110011	2	—
LDC.L	@Rm+,R7_BANK	(Rm) → R7_BANK, Rm + 4 → Rm	0100mmmm11110111	1	—
LDC	Rm,R7_BANK	Rm → R7_BANK	0100mmmm11111110	1	—
SHAD	Rm,Rn	Rn ≥ 0 when Rn << Rm → Rn, Rn < 0 when Rn >> Rm → (MSB→) Rn	0100nnnnmmmm1100	1	—
SHLD	Rm,Rn	Rn ≥ 0 when Rn << Rm → Rn, Rn < 0 when Rn >> Rm → (0→) Rn	0100nnnnmmmm1101	1	—
MAC.W	@Rm+,@Rn+	With sign, (Rn) × (Rm) + MAC → MAC	0100nnnnmmmm1111	2 (to 5)*1	—
MOV.L	@(disp,Rm),Rn	(disp + Rm) → Rn	0101nnnnmmmmdddd	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	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

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
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	0111nnnniiiiiiii	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	10000100mmmmdddd	1	—
MOV.W	@(disp,Rm),R0	(disp + Rm) → sign extension → R0	10000101mmmmdddd	1	—
CMP/EQ	#imm,R0	When R0 = imm, 1 →T	10001000iiiiiiii	1	Comparison results
BT	disp	When T = 1, disp + PC → PC; when T = 1, nop.	10001001dddddddd	3/1*2	—
BF	disp	When T = 0, disp + PC → PC; when T = 1, nop	10001011dddddddd	3/1*2	—
BT/S	label	If T = 1, disp + PC → PC; if T = 0, nop	10001101dddddddd	2/1*2	—
BF/S	label	If T = 0, disp + PC → PC; if T = 1, nop	10001111dddddddd	2/1*2	—
MOV.W	@(disp,PC),Rn	(disp + PC) → sign extension → Rn	1001nnnnddddddd	1	—
BRA	disp	Delayed branching, disp + PC → PC	1010dddddddddddd	2	—
BSR	disp	Delayed branching, PC → PR, disp + PC → PC	1011dddddddddddd	2	—

Table A.49 Instruction Set by Instruction Code (cont)

Instruction		Operation	Code	Cycles	T Bit
MOV.B	R0,@(disp,GBR)	R0 → (disp + GBR)	11000000dddddddd	1	—
MOV.W	R0,@(disp,GBR)	R0 → (disp + GBR)	11000001dddddddd	1	—
MOV.L	R0,@(disp,GBR)	R0 → (disp + GBR)	11000010dddddddd	1	—
TRAPA	#imm	imm → TRA, PC → SPC, SR → SSR, 1 → SR.MD/BL/RB, 0x160 → EXPEVT VBR + H'00000100 → PC	11000011iiiiiiii	6	—
MOV.B	@(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000100dddddddd	1	—
MOV.W	@(disp,GBR),R0	(disp + GBR) → sign extension → R0	11000101dddddddd	1	—
MOV.L	@(disp,GBR),R0	(disp + GBR) → R0	11000110dddddddd	1	—
MOVA	@(disp,PC),R0	disp + PC → R0	11000111dddddddd	1	—
TST	#imm,R0	R0 & imm, when result is 0, 1 → T	11001000iiiiiiii	1	Test results
AND	#imm,R0	R0 & imm → R0	11001001iiiiiiii	1	—
XOR	#imm,R0	R0 ^ imm → R0	11001010iiiiiiii	1	—
OR	#imm,R0	R0 | imm → R0	11001011iiiiiiii	1	—
TST.B	#imm,@(R0,GBR)	(R0 + GBR) & imm, when result is 0, 1 → T	11001100iiiiiiii	3	Test results
AND.B	#imm,@(R0,GBR)	(R0 + GBR) & imm → (R0 + GBR)	11001101iiiiiiii	3	—
XOR.B	#imm,@(R0,GBR)	(R0 + GBR) ^ imm → (R0 + GBR)	11001110iiiiiiii	3	—
OR.B	#imm,@(R0,GBR)	(R0 + GBR) | imm → (R0 + GBR)	11001111iiiiiiii	3	—
MOV.L	@(disp,PC),Rn	(disp + PC) → Rn	1101nnnnddddddd	1	—
MOV	#imm,Rn	#imm → sign extension → Rn	1110nnnniiiiiiii	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.50 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	00 MD	0010	STC SR,Rn	STC GBR,Rn	STC VBR,Rn	STC SSR,Rn
0000	Rn	01 MD	0010	STC SPC,Rn			
0000	Rn	10 MD	0010	STC R0_BANK, Rn	STC R1_BANK, Rn	STC R2_BANK, Rn	STC R3_BANK, Rn
0000	Rn	11 MD	0010	STC R4_BANK, Rn	STC R5_BANK, Rn	STC R6_BANK, Rn	STC R7_BANK, Rn
0000	Rn	00 MD	0011	BSRF Rn		BRAF Rn	
0000	Rn	10 MD	0011	PREF @Rn			
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	00 MD	1000	CLRT	SETT	CLRMAC	LDTLB
0000	0000	01 MD	1000	CLRS	SETS		
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				
0000	Rn	Fx	1010	STS MACH,Rn	STS MACL,Rn	STS PR,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	

Table A.50 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
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	
0100	Rn	Fx	0010	STS.L MACH,@–Rn	STS.L MACL,@–Rn	STS.L PR,@–Rn	
0100	Rn	00 MD	0011	STC.L SR,@–Rn	STC.L GBR,@–Rn	STC.L VBR,@–Rn	STC.L SSR,@–Rn
0100	Rn	01 MD	0011	STC.L SPC,@–Rn			
0100	Rn	10 MD	0011	STC.L R0_BANK, @–Rn	STC.L R1_BANK, @–Rn	STC.L R2_BANK, @–Rn	STC.L R3_BANK, @–Rn
0100	Rn	11 MD	0011	STC.L R4_BANK, @–Rn	STC.L R5_BANK, @–Rn	STC.L R6_BANK, @–Rn	STC.L R7_BANK, @–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	
0100	Rm	00 MD	0111	LDC.L @Rm+,SR	LDC.L @Rm+,GBR	LDC.L @Rm+,VBR	LDC.L @Rm+,SSR
0100	Rm	01 MD	0111	LDC.L @Rm+,SPC			

Table A.50 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
0100	Rm	10 MD	0111	LDC.L @Rm+, R0_BANK	LDC.L @Rm+, R1_BANK	LDC.L @Rm+, R2_BANK	LDC.L @Rm+, R3_BANK
0100	Rm	11 MD	0111	LDC.L @Rm+, R4_BANK	LDC.L @Rm+, R5_BANK	LDC.L @Rm+, R6_BANK	LDC.L @Rm+, R7_BANK
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	
0100	Rn	Fx	1011	JSR @Rn	TAS.B @Rn	JMP @Rn	
0100	Rn	Rm	1100	SHAD Rm,Rn			
0100	Rn	Rm	1101	SHLD Rm,Rn			
0100	Rm	00 MD	1110	LDC Rm,SR	LDC Rm,GBR	LDC Rm,VBR	LDC Rm,SSR
0100	Rm	01 MD	1110	LDC Rm,SPC			
0100	Rm	10 MD	1110	LDC Rm,R0_BANK	LDC Rm,R1_BANK	LDC Rm,R2_BANK	LDC Rm,R3_BANK
0100	Rm	11 MD	1110	LDC Rm,R4_BANK	LDC Rm,R5_BANK	LDC Rm,R6_BANK	LDC Rm,R7_BANK
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	00 MD	Rn	disp	MOV.B R0,@(disp:4, Rn)	MOV.W R0, @(disp:4,Rn)		

Table A.50 Operation Code Map (cont)

Instruction Code				Fx: 0000	Fx: 0001	Fx: 0010	Fx: 0011–1111
MSB			LSB	MD: 00	MD: 01	MD: 10	MD: 11
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
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	—						

Appendix B Pipeline Operation and Contention

The SH7700 series 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:

- 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

Table B.1 Instructions and Their Contention Patterns

Contention	Cycles	Stages	Instructions
None	1	3	• Transfers between registers • Operations between registers (except when a multiplier is involved) • Logical operations between registers • Shift and dynamic shift instructions • System control ALU instructions
	1	4	PREF instruction
	2	3	Unconditional branches
	3/1*2	3	Conditional branches
	2/1*2	3	Delayed conditional branch instructions
	4	3	SLEEP instruction
	4	5	RTE instruction
	6	9	TRAP instruction
MA contends with IF	5	5	LDC.L Rm, SR
	1	4	• Memory store instructions • STS.L instruction (PR)
	1 (2)*3	4 (5)*3	STC.L instruction
	3	6	Memory logic operations
	3	6	TAS instruction
MA contends with IF and sometimes with memory loads as well.	7	7	LDC.L @Rm+, SR
	1	5	• Memory load instructions • LDS.L instruction (PR)
	1	5	LDC.L instruction