

PanaX Series

The One to Watch for Constant Innovation-Making the Future Come Alive

MICROCOMPUTER

MN103E

MN103E Series

Instruction Manual

Pub.No.13350-020E

Panasonic

MS-DOS is a registered trademark of Microsoft Corporation.

Windows is a trademark of Microsoft Corporation.

PanaXSeries is a trademark of Matsushita Electric Industrial Co., Ltd.

The other corporation names, logotype and product names written in this book are trademarks or registered trademarks of their corresponding corporations.

Request for your special attention and precautions in using the technical information and semiconductors described in this book

- (1) An export permit needs to be obtained from the competent authorities of the Japanese Government if any of the products or technologies described in this book and controlled under the "Foreign Exchange and Foreign Trade Law" is to be exported or taken out of Japan.
- (2) The technical information described in this book is limited to showing representative characteristics and applied circuits examples of the products. It neither warrants non-infringement of intellectual property right or any other rights owned by our company or a third party, nor grants any license.
- (3) We are not liable for the infringement of rights owned by a third party arising out of the use of the product or technologies as described in this book.
- (4) The products described in this book are intended to be used for standard applications or general electronic equipment (such as office equipment, communications equipment, measuring instruments and household appliances).
Consult our sales staff in advance for information on the following applications:
 - Special applications (such as for airplanes, aerospace, automobiles, traffic control equipment, combustion equipment, life support systems and safety devices) in which exceptional quality and reliability are required, or if the failure or malfunction of the products may directly jeopardize life or harm the human body.
 - Any applications other than the standard applications intended.
- (5) The products and product specifications described in this book are subject to change without notice for modification and/or improvement. At the final stage of your design, purchasing, or use of the products, therefore, ask for the most up-to-date Product Standards in advance to make sure that the latest specifications satisfy your requirements.
- (6) When designing your equipment, comply with the guaranteed values, in particular those of maximum rating, the range of operating power supply voltage, and heat radiation characteristics. Otherwise, we will not be liable for any defect which may arise later in your equipment.
Even when the products are used within the guaranteed values, take into the consideration of incidence of break down and failure mode, possible to occur to semiconductor products. Measures on the systems such as redundant design, arresting the spread of fire or preventing glitch are recommended in order to prevent physical injury, fire, social damages, for example, by using the products.
- (7) When using products for which damp-proof packing is required, observe the conditions (including shelf life and amount of time let standing of unsealed items) agreed upon when specification sheets are individually exchanged.
- (8) This book may be not reprinted or reproduced whether wholly or partially, without the prior written permission of Matsushita Electric Industrial Co., Ltd.

If you have any inquiries or questions about this book or our semiconductors, please contact one of our sales offices listed at the back of this book.
--

About This Manual

This manual describes the details of the instruction set for the MN103E series. MN103E series has the following four kinds of microprocessors; AM33-1/AM33-2/AM33-2A/AM34-1, and this manual describes mainly the AM33-2 microprocessor core. When specifications differ according to microprocessor core, this describes the contents with each microprocessor-core mark. Chapter 2 explains the overview of the instruction system, instruction functions, instruction format. Chapter 2 explains all instruction operations, flag changes and etc. Chapter 3 explains the pipeline architecture, cautions on the instruction description and recommendations on the instruction description. Moreover, the instruction lists: instruction set and instruction map, are shown as appendix.

- How to search

There are four search ways in this manual in order to find necessary information quickly.

1. See the chapter title index at the beginning of the manual for finding each chapter title.
2. See the table of contents at the beginning of the manual for finding each title.
3. Each page has chapter title at the top end of the page and smaller title at the bottom end. You can roughly know the contents described on pages with turning pages.
4. See the index at the end of the manual for finding instructions. Use the indexes showing the instruction functions on the right pages for search.

- Related Manuals

The following related manuals are available. Please contact your sales representative for more details.

MN103E Series Cross Assembler User's Manual

<Describes the assembler syntax and writing method>

MN103E Series C compiler User's Manual: Usage Guide

<Describes the installation, startup and option of the C compiler>

MN1030/MN103S/MN103E Series C Compiler User's Manual: Language Description

<Describes the C-language syntax>

MN1030/MN103S/MN103E Series C Compiler User's Manual: Library Reference

<Describes the C-compiler standard library>

MN103E Series C Source Code Debugger for Windows® User's Manual

<Describes the use of the C Source Code Debugger for Windows®>

MN103E Series Installation Manual

<Describes the installation of Compiler, Crossassembler and C Source Code Debugger, and the procedure for bringing up the in-circuit emulator>

MN1030/MN103S/MN103E Series On-board Debug Unit Setup Manual

<Describes the connection of the On-board Debug Unit and the initial setting of On-board debugger>

MN101C/MN101D/MN101E/MN102E/MN102L/MN102H/MN1030/MN103S Series
PCI/PC Card installation Manual

<Describes the PCI/PC Card Installation>

- How to read

Chapter 1 consists of mainly titles, abstracts, main body and cautions.

Chapter 2 consists of instruction formats, operational descriptions and cautions.

Chapter 3 consists of pipeline architecture, sample program and cautions.

The following shows the layout and meaning of each part.

Section title →

Abstract
Introduction of this section →

List
The contents of the section
is described in the list. →

Chapter 1 Instruction introduction

3

Instruction functions

The instruction set is based on a simple instruction set. A C compiler will produce a generated code that is compact and optimized from this instruction set.

As the result of that the basic instruction word length is one byte, the instruction set is a simple one that limits data transfers with memory to load and store operations and it is possible to minimize the increase in code size due to the assembler program. Furthermore, since the generated code is compact, more instructions can be placed in the limited cache memory space, resulting in an improved cache hit ratio and making it possible to minimize the degradation of performance that results from accessing external memory on the event of a miss-hit.

The AM 33 microprocessor core instruction set consists of the following five instruction categories: basic instructions, extended basic instructions, extended operation instructions, LIW extended operation instructions, and floating-point operation instructions. The basic instructions are common throughout the entire AM 30 series; these instructions maintain compatibility between the different microprocessors in the series. The extended basic instructions are an extension of the basic instructions for the AM33 microprocessor core; these functions were added in order to enhance the interrupt functions and to support the extended registers. Functionally, these instructions are equivalent to basic instructions. The extended operation instructions provide compatibility with the extended instructions that were implemented in the AM31 microprocessor core. The LIW extended operation instructions support parallel operations on data. The floating-point operation instructions provide basic floating-point operations, such as arithmetic operations that handle single-precision floating-point numbers, multiply-and-accumulate operations.

The instructions are all listed in the following chart. There are 47 basic instructions, 18 extended operation instructions, 70 LIW extended operation instructions, and 15 floating-point operation instructions. (Almost all of the extended basic instructions were implemented as operand extensions, so they have been included in the count of the basic instructions.)

Transfer instructions	MOV, MOVU, MOVHU, MOVBU, MOVW, EXT, EXTH, EXTHU, EXTB, EXTBW, CLR, DCPF
Arithmetic operational instruction	ADD, ADDC, INC, INC4, SUB, SUBC, MUL, MULU, DIV, DIVU
Compare instruction	CMP
Logic operational instruction	AND, OR, XOR, NOT
Shift instruction	ASR, LSR, ASL, ASL2, ROR, ROL
Bit manipulation instruction	BTST, BSET, BCLR
Branch instruction	Bcc, Lcc, SETLB, JMP, CALL, CALLS, TRAP, RET, RETF, RETS, RTI, SYSCALL
Debug instruction	PI
NOP instruction	NOP

Instruction functions 9

Section title

Chapter2 Instruction details

FMOV

Transfer (Floating-point unit)

Abstract

Introduction of the section

FMOV Mem, FS*n*

Operation

FMOV in the case of (R*m*), FS*n*
mem32(R*m*) -> FS*n*

FMOV in the case of (SP), FS*n*
mem32(SP) -> FS*n*

FMOV in the case of (R*m*, R*i*), FS*n*
mem32(R*m*+R*i*) -> FS*n*

FMOV in the case of (disp, R*m*), FS*n*
mem32(R*m*+disp) -> FS*n*

FMOV in the case of (disp, SP), FS*n*
mem32(SP+disp) -> FS*n*

Memory contents are stored in the single-precision floating-point register (FS*n*).

Assembler mnemonic

Note

EC flag

FCC flag

Size

fmov (R*m*), FS*n*

-

-

-

-

-

-

-

3

fmov (SP), FS*n*

-

-

-

-

-

-

3

fmov (R*m*, R*i*), FS*n*

-

-

-

-

-

-

4

fmov (d8, R*m*), FS*n*

d8 is code-extended.

-

-

-

-

-

-

4

fmov (d24, R*m*), FS*n*

d24 is code-extended.

-

-

-

-

-

-

6

fmov (d32, R*m*), FS*n*

-

-

-

-

-

-

7

fmov (d8, SP), FS*n*

d8 is zero-extended.

-

-

-

-

-

-

4

fmov (d24, SP), FS*n*

d24 is zero-extended.

-

-

-

-

-

-

6

fmov (d32, SP), FS*n*

-

-

-

-

-

-

7

Flag change

VF : No change

ZF : No change

OF : No change

UF : No change

IF : No change

LF : No change

GF : No change

EF : No change

UF : No change

Flag change

● : Changed

- : No change

0 : Always 0

1 : Always 1

? : Undefined

* : User definition

Code size and cycle

number of instruction

The number of cycles

is calculated by the

following conditions.

(1) There is no

pipeline stall.

(2) Instruction fetch

is 2 cycle, and data

load/store is 1 cycle.

Caution

Please read in order to normally operate programs

!

When the memory address is not a multiple of 4, a system exception (address misalignment exception) occurs.

236 FMOV

Footer

Instructions are shown here.

Microprocessor mark

Applicable microprocessor cores are shown here.

< About This Manual 3 >

Description of the recommended instruction assignment

Chapter 3 Directions for using instructions

High-speed processing mark shows a recommended item to execute instructions at high speed.



High-speed processing

Recommended items

(3) Assignment of the instructions changing MDR at 2 cycles earlier than the final cycle of executing instructions and subsequent RETF

[Contents]

It is recommended that more than 1 cycle should be inserted between the instructions changing MDR at 2 cycles earlier than the final cycle of the instruction execution and the subsequent RETF.

[Examples]

[General description examples]

_ LABEL	FUNCINFO	_ inc	_ movm	_ (sp), [other]	_ <- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
					<- RET

Description example of general programs

A pipeline stall occurs and the operations of RETF is delayed until the write of MDR in MOVW has completed since the operations of RETF start before the write of MDR has completed.

Description example of recommended programs.

[Recommended description examples]

_ LABEL	FUNCINFO	_ movm	_ (sp), [other]	_ <- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
		inc	d0	
		retf		<- RET

RETF can be executed without the occurrence of pipeline stall since the write of MDR in MOVW has already completed when executing RETF.

Instructions applicable to recommended items

[Applicable instructions]

<Preceding instructions, or return target instruction> MOV (SP), reg, RET, RETF
<Subsequent instructions> RETF

Caution

Please read in order to normally operate programs



For the details of the description of FUNCINFO pseudo-instruction, refer to MN103E0 series cross-assembler user's manual.

Table of Contents

Chapter 1	Instruction Introduction
Chapter 2	Instruction Description
Chapter 3	Directions for using instructions
Chapter 4	Appendix
	Index

1

2

3

4

Table of Contents

Chapter 1 Instruction Introduction

1	Instruction system	2
2	Basic register set	3
2-1	Register set	3
2-1-1	Address register	4
2-1-2	Data register	4
2-1-3	Extension register	4
2-1-4	Stack pointer	4
2-1-5	Program counter	5
2-1-6	Processor status word	5
2-1-7	Loop instruction register	6
2-1-8	Loop address register	6
2-1-9	Multiply/divide register	7
2-2	Extended register set	7
2-2-1	Extended operation register	7
2-2-2	Multiply-and-accumulate operation registers	7
2-2-3	Multiply-and-accumulate overflow flag	7
2-3	Floating-point register set	8
2-3-1	Floating-point register	8
2-3-2	Floating-point unit control register	8
3	Instruction functions	9
3-1	Transfer instructions	11
3-2	Arithmetic operation instruction	11
3-3	Compare instruction	12
3-4	Logic operation instruction	12
3-6	Shift instructions	13
3-7	NOP instructions	13
3-5	Bit manipulation instructions	13
3-8	Branch instructions	14
3-9	Debug instruction	14
3-10	Extended operation instructions	15
3-11	LIW extended operation instructions	16
3-12	Floating-point operation instructions	17
4	Memory space	18
	Address space when using an MMU (logical address space)	18

Address space when not using an MMU	19
5 Addressing mode	20
5-1 Register direct	21
5-2 Immediate	21
5-3 Register indirect	21
5-4 Register relative indirect	22
5-5 Absolute	22
5-6 Register indirect with indexed addressing	22
6 Instruction formats	23
6-1 Data formats	25
6-2 Endian	26

Chapter 2 Instruction Description

Notations	30
Transfer instructions	
MOV Reg1, Reg2	33
MOV imm, Reg	33
MOV MSP, An <Privileged instruction>	34
MOV Am, MSP <Privileged instruction>	34
MOV SSP, An <Privileged instruction>	35
MOV Am, SSP <Privileged instruction>	35
MOV USP, An <Privileged instruction>	36
MOV Am, USP <Privileged instruction>	36
MOV EPSW, Dn <Privileged instruction>	37
MOV Dm, EPSW <Privileged instruction>	37
MOV PSW, Dn <Privileged instruction>	38
MOV Dm, PSW <Privileged instruction>	38
MOV MDR, Dn	39
MOV Dm, MDR	39
MOV imm, SP	40
MOV Reg, SP	40
MOV SP, Reg	40
MOV imm, MDRQ	41
MOV MDRQ, Rn	41
MOV Rm, MDRQ	41
MOV imm, MCRH	42
MOV MCRH, Rn	42
MOV Rm, MCRH	42
MOV imm, MCRL	43

MOV MCRL, Rn	43
MOV Rm, MCRL	43
MOV imm, MCVF	44
MOV MCVF, Rn	44
MOV Rm, MDRQ	44
MOV PC, An	45
MOV Mem, Reg	46
MOV Reg, Mem	48
MOV (Rm+,imm), Rn	50
MOV (Rm+), Rn	50
MOV Rm, (Rn+,imm)	51
MOV Rm, (Rn+)	51
MOVU imm, Reg	52
MOVHU Mem, Reg	53
MOVHU Reg, Mem	55
MOVHU (Rm+, imm), Rn	57
MOVHU (Rm+), Rn	57
MOVHU Rm, (Rn+, imm)	58
MOVHU (Rm+), Rn	58
MOVBU Mem, Reg	59
MOVBU Reg, Mem	60
MOVM (SP), regs (SP relative)	61
MOVM regs, (SP) (SP relative)	63
MOVM (SP), regs (SP relative)	65
MOVM regs,(USP)(USP relative<Privileged instruction>)	67
EXT Reg	69
EXTH Reg.....	70
EXTH Reg1, Reg2	70
EXTHU Reg.....	71
EXTHU Reg1, Reg2	71
EXTB Reg.....	72
EXTB Reg1, Reg2	72
EXTBU Reg	73
EXTBU Reg1, Reg2	73
EXTBU Reg	74
DCPF Mem	75

Arithmetic instructions

ADD Reg1, Reg2	76
----------------------	----

ADD Reg1, Reg2, Reg3	76
ADD imm, Reg	77
ADD imm, SP	77
ADDC Reg1, Reg2	78
ADDC Reg1, Reg2, Reg3	78
ADDC imm, Reg	79
SUB Reg1, Reg2	80
SUB Reg1, Reg2, Reg3	80
SUB imm, Reg	81
SUBC Reg1, Reg2	82
SUBC Reg1, Reg2, Reg3	82
SUBC imm, Reg	83
MUL Reg1, Reg2	84
MUL Reg1, Reg2, Reg3, Reg4	84
MUL imm, Reg	85
MULU Reg1, Reg2	86
MULU Reg1, Reg2, Reg3, Reg4	86
MULU imm, Reg	87
DIV Reg1, Reg2	88
DIVU Reg1, Reg2	89
INC Reg	90
INC4 Reg	91
 Compare instructions	
CMP Reg1, Reg2	92
CMP imm, Reg	93
 Logical instructions	
AND Reg1, Reg2	94
AND Reg1, Reg2, Reg3	94
AND imm, Reg	95
AND imm16, PSW (Privileged instruction)	96
AND imm32, EPSW (Privileged instruction)	96
OR Reg1, Reg2	97
OR Reg1, Reg2, Reg3	97
OR imm, Reg	98
OR imm16, PSW (Privileged instruction)	99
OR imm32, EPSW (Privileged instruction)	99
XOR Reg1, Reg2	100

XOR Reg1, Reg2, Reg3	100
XOR imm, Reg	101
NOT Reg	102
 Bit manipulate instructions	
BTST imm, Reg	103
BTST imm, Mem	103
BSET Reg, Mem	104
BSET imm, Mem	105
BCLR Reg, Mem	106
BCLR imm, Mem	107
 Shift instructions	
ASR Reg1, Reg2	108
ASR Reg1, Reg2, Reg3	109
ASR imm, Reg	110
LSR Reg1, Reg2	111
LSR Reg1, Reg2, Reg3	112
LSR imm, Reg	113
ASL Reg1, Reg2	114
ASL Reg1, Reg2, Reg3	115
ASL imm, Reg	116
ASL2 Reg	117
ROR Reg	118
ROL Reg	119
 Branch instructions	
Bcc (d8, PC)	120
Lcc	121
SETLB	122
JMP (An)	123
JMP (d16, PC)	123
JMP (d32, PC)	124
CALL (d16, PC), regs, imm8	125
CALL (d32, PC), regs, imm8	126
CALLS (An)	127
CALLS (d16, PC)	128
CALLS (d32, PC)	128
RET regs, imm8	129
RETF	130

RETS	131
RTI (Privileged instruction)	132
TRAP	133
NOP	134
SYSCALL imm4	135
 Debug instructions	
PI	136
 Extension operation instructions	
DMULH Rm, Rn	137
DMULH Rm, Rn, Rd1, Rd2	137
DMULH imm, Rn	138
DMULHU Rm, Rn	139
DMULHU Rm, Rn, Rd1, Rd2	139
DMULHU imm, Rn	140
DMACH Rm, Rn, Rd	141
DMACH Rm, Rn	141
DMACH imm, Rn	142
DMACHU Rm, Rn, Rd	143
DMACHU Rm, Rn	143
DMACHU imm, Rn	144
MAC Rm, Rn, Rd1, Rd2	145
MAC Rm, Rn	145
MAC imm, Rn	146
MACU Rm, Rn, Rd1, Rd2	147
MACU Rm, Rn	147
MACU imm, Rn	148
MACH Rm, Rn	149
MACH Rm, Rn, Rd1, Rd2	149
MACH imm, Rn	150
MACHU Rm, Rn	151
MACHU Rm, Rn, Rd1, Rd2	151
MACHU imm, Rn	152
MACB Rm, Rn	153
MACB Rm, Rn, Rd	153
MACB imm, Rn	154
MACBU Rm, Rn	155

MACBU Rm, Rn, Rd	155
MACBU imm, Rn	156
SWHW Rm, Rn	157
SWAP Rm, Rn	158
SWAPH Rm, Rn	159
SAT16 Rm, Rn	160
SAT24 Rm, Rn	161
MCSTE Rm, Rn	162
MCSTE imm, Rn	164
BSCH Rm, Rn	166
BSCH Rm, Rn, Rd	167

LIW instructions

ADD_OP2 Rm1, Rn1, Rm2, Rn2	168
ADD_OP2 Rm1, Rn1, imm, Rn2	169
ADD_OP2 imm, Rn1, Rm2, Rn2	170
ADD_OP2 imm, Rn1, imm, Rn2	171
CMP_OP2 Rm1, Rn1, Rm2, Rn2	172
CMP_OP2 Rm1, Rn1, imm, Rn2	172
CMP_OP2 imm, Rn1, Rm2, Rn2	173
CMP_OP2 imm, Rn1, imm, Rn2	173
SUB_OP2 Rm1, Rn1, Rm2, Rn2	174
SUB_OP2 Rm1, Rn1, imm, Rn2	175
SUB_OP2 imm, Rn1, Rm2, Rn2	176
SUB_OP2 imm, Rn1, imm, Rn2	177
MOV_OP2 Rm1, Rn1, Rm2, Rn2	178
MOV_OP2 Rm1, Rn1, imm, Rn2	179
MOV_OP2 imm, Rn1, Rm2, Rn2	180
MOV_OP2 imm, Rn1, imm, Rn2	181
AND_OP2 Rm1, Rn1, Rm2, Rn2	182
AND_OP2 Rm1, Rn1, imm, Rn2	183
OR_OP2 Rm1, Rn1, Rm2, Rn2	184
OR_OP2 Rm1, Rn1, imm, Rn2	185
XOR_OP2 Rm1, Rn1, Rm2, Rn2	186
XOR_OP2 Rm1, Rn1, imm, Rn2	187
DMACH_OP2 Rm1, Rn1, Rm2, Rn2	188
DMACH_OP2 Rm1, Rn1, imm, Rn2	189
SWHW_OP2 Rm1, Rn1, Rm2, Rn2	190
SWHW_OP2 Rm1, Rn1, imm, Rn2	191
SAT16_OP2 Rm1, Rn1, Rm2, Rn2	192

SAT16_OP2	Rm1, Rn1, imm, Rn2	193
OP1_ADD	Rm1, Rn1, Rm2, Rn2	194
OP1_ADD	imm, Rn1, Rm2, Rn2	195
OP1_ADD	Rm1, Rn1, imm, Rn2	196
OP1_ADD	imm, Rn1, imm, Rn2	197
OP1_SUB	Rm1, Rn1, Rm2, Rn2	198
OP1_SUB	imm, Rn1, Rm2, Rn2	199
OP1_SUB	Rm1, Rn1, imm, Rn2	200
OP1_SUB	imm, Rn1, imm, Rn2	201
OP1_CMP	Rm1, Rn1, Rm2, Rn2	202
OP1_CMP	imm, Rn1, Rm2, Rn2	202
OP1_CMP	Rm1, Rn1, imm, Rn2	203
OP1_CMP	imm, Rn1, imm, Rn2	203
OP1_MOV	Rm1, Rn1, Rm2, Rn2	204
OP1_MOV	imm, Rn1, Rm2, Rn2	205
OP1_MOV	Rm1, Rn1, imm, Rn2	206
OP1_MOV	imm, Rn1, imm, Rn2	207
OP1_ASR	Rm1, Rn1, Rm2, Rn2	208
OP1_ASR	imm, Rn1, Rm2, Rn2	209
OP1_ASR	Rm1, Rn1, imm, Rn2	210
OP1_ASR	imm, Rn1, imm, Rn2	211
OP1_LSR	Rm1, Rn1, Rm2, Rn2	212
OP1_LSR	imm, Rn1, Rm2, Rn2	213
OP1_LSR	Rm1, Rn1, imm, Rn2	214
OP1_LSR	imm, Rn1, imm, Rn2	215
OP1_ASX	Rm1, Rn1, Rm2, Rn2	216
OP1_ASX	imm, Rn1, Rm2, Rn2	217
OP1_ASX	Rm1, Rn1, imm, Rn2	218
OP1_ASX	imm, Rn1, imm, Rn2	219
MOV_Lcc	(Rm+, imm), Rn	220

UDF instructions

UDF00	Dm, Dn (MULQ Dm, Dn)	222
UDF00	imm, Dn (MULQ imm, Dn)	222
UDF01	Dm, Dn (MULQU Dm, Dn)	223
UDFU01	imm, Dn (MULQU imm, Dn)	223
UDF02	Dm, Dn (MCST32, MCST16, MCST8)	224
UDF03	Dm, Dn (MCST9 Dn)	226

UDF04 Dm, Dn (SAT16 Dm, Dn)	227
UDF05 Dm, Dn (SAT24 Dm, Dn)	228
UDF06 Dm, Dn (MCST48 Dn)	229
UDF07 Dm, Dn (BSCH Dm, Dn)	230
UDF08 Dm, Dn (SWAP Dm, Dn)	231
UDF09 Dm, Dn (SWAPH Dm, Dn)	232
UDF12 Dm, Dn (GETCHX Dn)	233
UDF13 Dm, Dn (GETCLX Dn).....	234
UDF15 Dm, Dn (GETX, Dn).....	235

Floating-point operation instructions

FMOV Mem, FS _n	236
FMOV FS _m , Mem	237
FMOV (R _m +), FS _n	238
FMOV (R _m +, imm), FS _n	239
FMOV FS _m , (R _n +)	240
FMOV FS _m , (R _n +, imm)	241
FMOV FS _m , FS _n	242
FMOV FS _m , R _n	242
FMOV R _m ,FS _n	243
FMOV imm,FS _n	243
FMOV FPCR,R _n	244
FMOV R _m ,FPCR	245
FMOV imm,FPCR.....	246
FMOV Mem,FD _n	247
FMOV FD _m ,Mem	248
FMOV (R _m +),FD _n	249
FMOV (R _m +,imm),FD _n	250
FMOV FD _m , (R _n +).....	251
FMOV FD _m , (R _n +,imm).....	252
FABS FS _n	253
FABS FS _m , FS _n	253
FNEG FS _n	254
FNEG FS _m , FS _n	254
FRSQRT FS _n	255
FRSQRT FS _m , FS _n	255
FCMP FS _m 1, FS _m 2	257
FCMP imm, FS _m	257
FADD FS _m , FS _n	260

FADD FSm1, FSm2, FS _n	260
FADD imm, FSm, FS _n	261
FSUB FSm1, FSm2, FS _n	263
FSUB FSm, FS _n	263
FSUB imm, FSm, FS _n	264
FMUL FSm, FS _n	266
FMUL FSm1, FSm2, FS _n	266
FMUL imm32, FSm, FS _n	267
FDIV FSm, FS _n	269
FDIV FSm1, FSm2, FS _n	269
FDIV imm, FSm, FS _n	270
FMADD FSm1, FSm2, FSm3, FS _n	272
FMSUB FSm1, FSm2, FSm3, FS _n	278
FNMADD FSm1, FSm2, FSm3, FS _n	284
FNMSUB FSm1, FSm2, FSm3, FS _n	290
FBCC (d8, PC)	296
FLCC	297

Chapter 3 Directions for using instructions

Cautions for programming	300
1 Pipeline architecture	301
1-1 Pipeline operation	301
1-2 Pipeline operations of operations between registers	302
1-3 Pipeline operation of data load	302
1-4 Pipeline operations of data store	303
1-5 Branch pipeline operations	303
1-6 Pipeline operations of SETLB and LCC	304
1-7 Number of instruction executing cycles	305
1-7-1 No dependence between instructions	305
1-7-2 Register dependence between instructions	306
1-7-3 Flag dependence between instructions	308
1-7-4 When the FPU instruction is subsequent to CPU load/store instruction	309
2 Cautions on the instruction description	310
3 Recommendations on Instruction description	311
3-1 Instruction assignment subsequent to branch instruction	313
3-2 Instruction assignment subsequent to SETLB	316
3-3 Assignment of the instructions preceding RETF	317
3-4 Assignment of the instructions of CALL/CALLS branch targets	321

Chapter 4 Appendix

 Instruction cycle 324

 Instruction set 343

 Instruction map 377

INDEX

 Index 388

1

Instruction system

32-bit microcontroller MN103E series is a upward microprocessor core of our 32-bit microcontroller MN103S series. This has the instruction extended in MN103E series as well as the MN103S-series instruction set and can be designed for a wide range of applications from information processing to signal processing. It consists of MN103E extending part to implement architecture extension, MMU, cache memory, FPU, and bus control circuit as centering a compact 32-bit CPU core with the instruction set having 1-byte basic instruction word length.

CPU core

AM33-1, AM33-2, AM33-2A and AM34-1 are the 32-bit microprocessor cores of Matsushita-original C language-oriented 8/16/32 bit microprocessor AM series. The instruction specification is partly different depending on the microprocessor core. Each core is defined as shown below.

AM33-1...32 bit processor-typed microprocessor core.



This consists of MN103E extending part to implement architecture extension, MMU, cache memory, and four modules of the bus control circuit.

AM33-2...32 bit processor-typed microprocessor core.



This consists of MN103E extending part to implement architecture extension, MMU, cache memory, and four FPU modules..

AM33-2A...32 bit processor-typed microprocessor core.



This consists of MN103E extending part to implement architecture extension, MMU, and three modules of cache memory.

AM34-1...32 bit processor-typed microprocessor core.



This consists of MN103E extending part to implement architecture extension, MMU, cache memory, and four FPU modules.

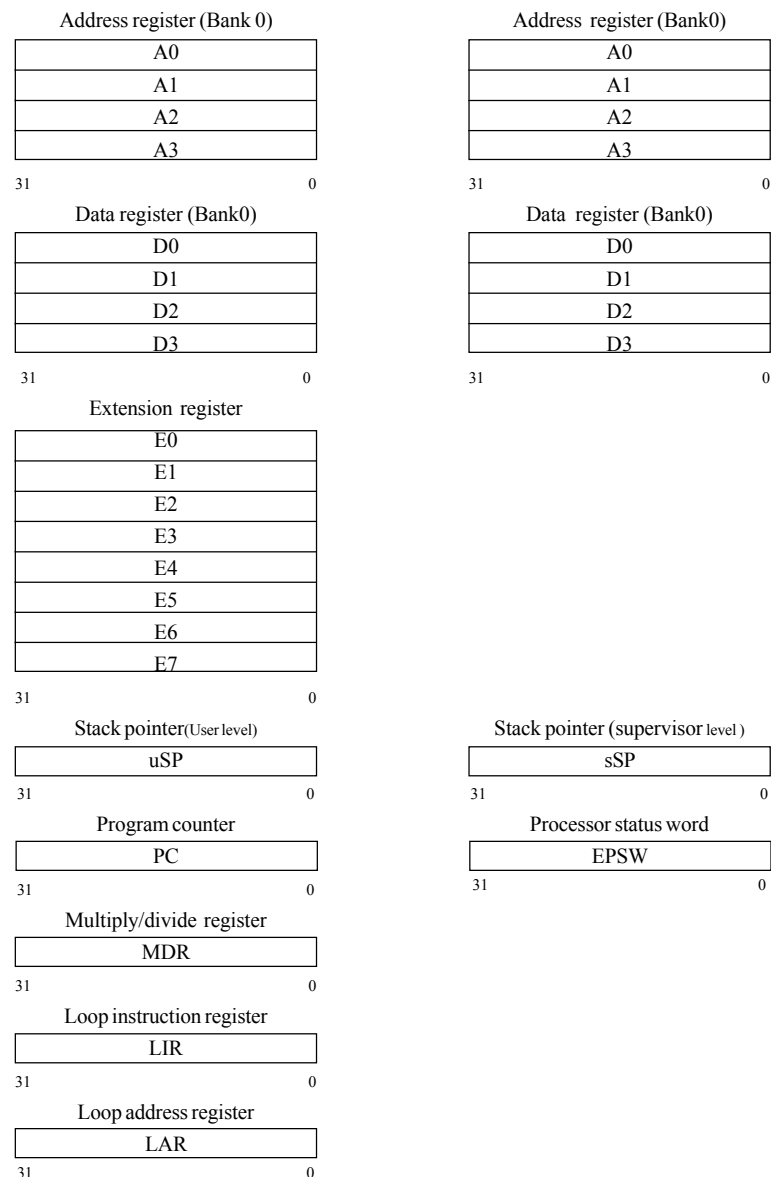
This manual describes mainly AM33-1 microprocessor core, and when each microprocessor core has different specifications, it shows the specifications with the above microprocessor core marks.

2 Register set

The register set is divided into a basic register set and an extending register set for the extension operating instruction implemented in MN103E series.

2-1 Basic register set

The basic register set consists of the followings: data register for operations such as addition and subtraction, address register for pointers, extension register capable of general-purpose use, stack pointer, program pointer, processor status word, multiply/divide register, loop instruction register, loop address register. It highly contributes to compressing the instruction code size and improving the performance, and enables programming by high-level languages such as C language. Especially, the data register and address register are comprised of banks and can have different registers between the user and supervisor levels. Total of 16 registers in the data register, address register and extension register can be used as flat general-purpose registers. (indicated as Rn for convenience in writing .)



2-1-1 Address register

A0-A3: Address register

This register is mainly used as pointer of the address. In the operational instruction of AM31 compatibility, it can operate only through the instruction for calculating address (Addition and subtraction, comparison). It is used as address pointer to data in the transfer instruction of AM31 compatibility, and the transfer to the memory is always carried out at 32-bit length. It can be used as general-purpose register in the extending basic instruction.

The substance of the register is comprised of the two banks that one of them is selected by nAR bit of EPSW. For the sake of convenience, the register bank, which is selected when nAR bit of EPSW is "0", is shown as Bank 0, and the register bank, which is selected when nAR bit of EPSW is "1", is shown as Bank 1. Normally Bank 0 is called "bank register", and Bank 1 is called "alternative bank register". The registers of Bank 0 and Bank 1 cannot be simultaneously accessed.

2-1-2 Data register

D0-D3: Data register (32 bits x 4)

This is an operational register that can be used generally for all operations. Operations are carried out in 32-bit length. When using 8-bit or 16-bit data, changing the data size is performed through data transfer to memory or EXTB/EXTH instruction execution.

The substance of the register is comprised of the two banks that one of them is selected by nAR bit of EPSW. For the sake of convenience, the register bank, which is selected when nAR bit of EPSW is "0", is shown as Bank 0, and the register bank, which is selected when nAR bit of EPSW is "1", is shown as Bank 1. Normally Bank 0 is called "bank register", and Bank 1 is called "alternative bank register". The registers of Bank 0 and Bank 1 cannot be simultaneously accessed.

2-1-3 Extension register

E0-E7: Extension register (32 bits x 8)

This is a general-purpose register to store an operation parameter and operation intermediate result. It is positioned as the extension register of data register (D0-D3) and address register (A0-A3) in the basic register set, and is used through the extension basic instruction, extension operational instruction, and DSP extension operational instruction.

2-1-4 Stack pointer

uSP, sSP: Stack pointer

This is a register to store a pointer indicating the start address of the stack area. The contents of the uSP at the user level and the sSP at the supervisor level are used as stack pointer. The execution of the instruction referring to this register is allowed only at the monitor and supervisor levels.

Privileged levels

Two kinds of status are defined at the privileged levels. The accessible memory space differs depending on each privileged level.

2-1-5 Program counter

PC: Program counter (32 bits x 1)

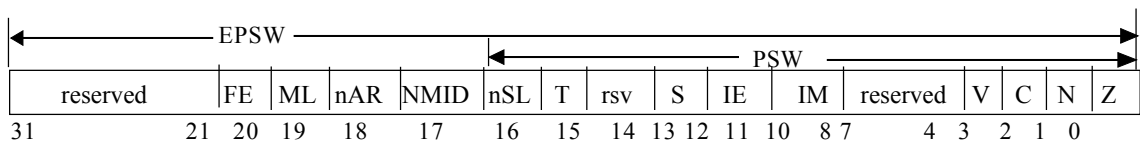
This is a register to store the address of the instruction currently being executed in CPU.

2-1-6 Processor status word

PSW: Processor status word

This is a register to show the status of CPU. It stores the flag of the operational result, interrupt mask level and etc. PSW of AM30 series microprocessor core is extended up to 32 bits and defined as EPSW in AM33 series microprocessor core. The lower 16 bits are shown as PSW and are identical to those in the AM30/AM31/AM32 microprocessor cores.

The execution of the instruction referring directly to this register is allowed only at the monitor and supervisor levels.



Z: Zero flag

This bit indicates whether the result of an operation is "0" or not.

For details, refer to the operational descriptions of the individual instructions.

N: Negative flag

This bit indicates whether the result of an operation is negative or not.

For details, refer to the operational descriptions of the individual instructions.

C: Carry flag

This bit indicates whether the execution of an operation caused a carry out of the MSB/borrow to the MSB or not.

For details, refer to the operational descriptions of the individual instructions.

V: Overflow flag

This bit indicates whether the execution of an operation caused an overflow as a value with a code or not.

For details, refer to the operational descriptions of the individual instructions.

IM2-0: Interrupt mask

These bits store the current asynchronous interrupt mask level. When IE=1, maskable interrupts with a level higher than that indicated by the IM bits are accepted. If a maskable interrupt is accepted, these bits are changed to the level of the maskable interrupt that was accepted after the PSW is saved.

IE: Interrupt enable

This is used to enable the acceptance of asynchronous interrupts, except for nonmaskable interrupts and reset interrupts. When IE=1, maskable interrupts are accepted. If an asynchronous interrupt is accepted, IE is set to "0". (disabling maskable interrupts)

S1-0: Software auxiliary bits

These auxiliary bits can be used in any desired fashion by system software and etc.

The application of these bits depend on the content of the software.

T: Trace enable

When T flag is set to 1, a single step interrupt is caused per execution of an instruction.

This bit cannot be normally updated.

nSL: Supervisor level

This indicates the current execution level. When $nSL = 0$, supervisor level is in effect; when $nSL = 1$, user level is in effect. When starting up after a reset in normal mode, $nSL = 0$ (i.e. supervisor level). If an asynchronous interrupt and a synchronous interrupt is generated, nSL is set to "0" after the EPSW is saved. After returning from the asynchronous interrupt/ synchronous interrupt and restoring the saved contents to the EPSW, the execution level in effect when the asynchronous interrupt/ synchronous interrupt was generated is restored.

NMID: Nonmaskable interrupt disable

This flag disables the acceptance of nonmaskable interrupts.

When $NMID = 0$, nonmaskable interrupts are enabled. When starting up after a reset, $NMID = 0$, that is, nonmaskable interrupts are accepted. If a nonmaskable interrupt is accepted, $NMID$ is set to "1" (disabling interrupts).

nAR: Register bank control

This indicates that the register of Bank 1 in the bank portion of the register file (A0-A3, D0-D3) is to be used. If an asynchronous interrupt/ synchronous interrupt is accepted, nAR is set to "0" after the EPSW is saved.

ML: Monitor level

This indicates the current execution level. $ML = 1$ is the monitor level and $ML = 0$ is non-monitor level. ML has priority over $PSW.nSL$.

$EPSW.ML$ is fixed to "0" in a normal mode. Accordingly, it cannot become to the monitor level. In a debug mode, if the monitor interrupts are caused, ML is set to "1" (monitor level) after the EPSW is saved.

FE: FPU enable

This enables the use of the FPU (floating-point unit) and enables the execution of the floating-point operating instructions.

When $FE = 1$, floating-point operating instructions can be executed.

When $FE = 0$ and floating-point instructions are to be executed, FPU disable exceptions are generated. For the details of the floating-point unit, refer to the chapter of "floating-point unit".

reserved/ rsv: Reserved field

These are reserved for future functional extension. Reading these fields always returns a value of "0". When writing these fields, always write "0".

2-1-7 Loop instruction register**LIR: Loop instruction register**

This register stores the first four bytes of the branch destination instruction stream when executing a LOOP instruction. This register is used to speed up the execution of LOOP instructions.

This register is set by the SETLB instruction. This register can also be saved to and restored from the stack area by using the MOV instruction.

2-1-8 Loop address register**LAR: Loop address register**

This register stores the leading address of the instruction stream subsequent to the instruction stream set in LIR; the jump address+4 of the LOOP instruction.

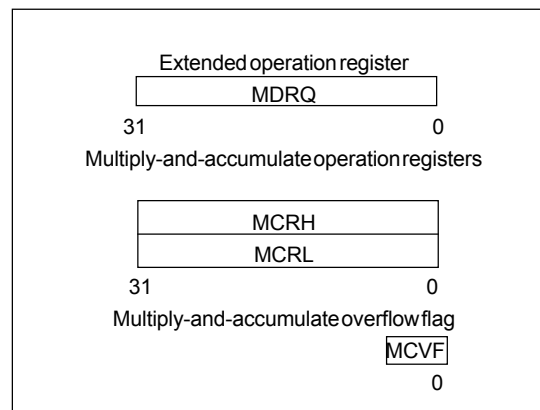
2-1-9 Multiply/divide register

MDR: Multiply/divide register

This register is provided for multiply/divide instructions. This register stores the upper 32 bits of a 64-bit multiplication result. For a division operation, this register stores the upper 32 bits of the divided and the remainder (32 bits). For details, refer to the operational descriptions of the individual instructions.

2-2 Extended register set

The extended register set supports the extended operation instructions that are provided in the MN103E series.



2-2-1 Extended operation register

MDRQ: Extended operation register

This register is used by the multiply/divide.

This stores the upper 32 bits of the 64-bit multiplication result in multiplication.

This stores the 32 bits of the surplus in division.

2-2-2 Multiply-and-accumulate operation registers

MCRH, MCRL: Multiply-and-accumulate operation registers

These registers are provided as accumulators for the multiply-and-accumulate operation that is performed by the executed operation unit.

MCRH stores the upper 32 bits of the 64-bit multiply-and accumulate operation result, and MCRL stores the lower 32 bits of the 64-bit multiply-and accumulate operation result.

For details, refer to the description of the operation of the individual instructions.

2-2-3 Multiply-and-accumulate overflow flag

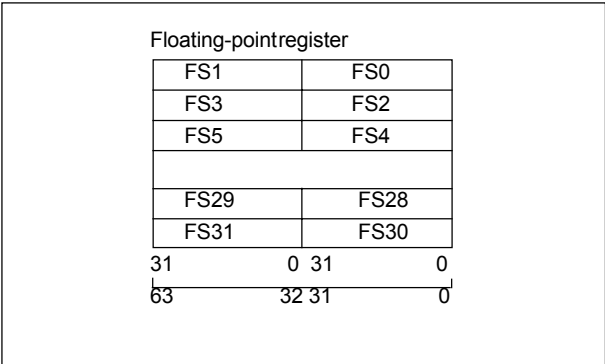
MCVF: Multiply-and-accumulate overflow flag

This register is used to store the result of overflow detection for the result produced by executing the multiply-and-accumulate operation instruction.

For details, refer to the description of the operation of the individual instructions.

2-3 Floating-point register set

The floating-point register set is used to execute the floating-point operation instruction in the floating-point operation unit. This consists of the 32-bit single-precision floating-point registers, FS0-FS31, and the floating-point unit control register (FPCR) having the operation control and flag information of the floating-point unit. These floating-point registers can be treated as sixteen 64-bit double-precision floating-point register, FD0-FD30. The floating-point register set can be access in the user level.



2-3-1 Floating-point register

FS0-FS31/FD0-FD30: Floating-point register

This is a general-purpose floating-point register which stores the operation parameter and the floating-point operational intermediate result of the floating-point operations. This consists of the floating-point registers comprising 32-bit single-precision floating-point registers, FS0-FS31. These floating-point registers can be treated as sixteen 64-bit double-precision floating-point register, FD0-FD30.

2-3-2 Floating-point unit control register

FPCR: Floating-point unit control register

This is a control register which has the operation control of the floating-point unit and the flag information. It has the floating-point operation flag and floating-point operational exception flag as well as controls the rounding mode of the floating-point operation and the floating-point exceptional operations.

3

Instruction functions

The instruction set is based on a simple instruction set. A C compiler will produce a generated code that is compact and optimized from this instruction set.

As the result of that the basic instruction word length is one byte, the instruction set is a simple one that limits data transfers with memory to load and store operations and it is possible to minimize the increase in code size due to the assembler program. Furthermore, since the generated code is compact, more instructions can be placed in the limited cache memory space, resulting in an improved cache hit ratio and making it possible to minimize the degradation of performance that results from accessing external memory on the event of a miss-hit.

The AM 33 microprocessor core instruction set consists of the following five instruction categories: basic instructions, extended basic instructions, extended operation instructions, LIW extended operation instructions, and floating-point operation instructions. The basic instructions are common throughout the entire AM 30 series; these instructions maintain compatibility between the different microprocessors in the series. The extended basic instructions are an extension of the basic instructions for the AM33 microprocessor core; these functions were added in order to enhance the interrupt functions and to support the extended registers. Functionally, these instructions are equivalent to basic instructions. The extended operation instructions provide compatibility with the extended instructions that were implemented in the AM31 microprocessor core. The LIW extended operation instructions support parallel operations on data. The floating-point operation instructions provide basic floating-point operations, such as arithmetic operations that handle single-precision floating-point numbers, multiply-and-accumulate operations.

The instructions are all listed in the following chart. There are 47 basic instructions, 18 extended operation instructions, 70 LIW extended operation instructions, and 15 floating-point operation instructions. (Almost all of the extended basic instructions were implemented as operand extensions, as they have been included in the count of the basic instructions.)

Transfer instructions	MOV,MOVU,MOVHU,MOVBU,MOVM,EXT,EXTH,EXTHU,EXTB,EXTBU,CLR,DCPF
Arithmetic operational instruction	ADD,ADDC,INC,INC4,SUB,SUBC,MUL,MULU,DIV,DIVU
Compare instruction	CMP
Logic operational instruction	AND,OR,XOR,NOT
Shift instruction	ASR,LSR,ASL,ASL2,ROR,ROL
Bit manipulation instruction	BTST,BSET,BCLR
Branch instruction	Bcc,Lcc,SETLB,JMP,CALL,CALLS,TRAP,RET,RETF,RETS,RTI,SYSCALL
Debug instruction	PI
NOP instruction	NOP

Extended operational instruction	DMULH,DMULHU,MAC,MACU,MACH,MACHU,MACB,MACBU,DMACH,DMACHU,SAT16,SAT24,MCSTE,BSCH,SWAP,SWAPH,SWHW
LIW-typed extended operational instruction	ADD_ADD,ADD_SUB,ADD_CMP,ADD_MOV,ADD_ASR,ADD_LSR,ADD_ASL, SUB_ADD,SUB_SUB,SUB_CMP,SUB_MOV,SUB_ASR,SUB_LSR,SUB_ASL, CMP_ADD,CMP_SUB,CMP_MOV,CMP_ASR,CMP_LSR,CMP_ASL, MOV_ADD,MOV_SUB,MOV_CMP,MOV_MOV,MOV_ASR,MOV_LSR,MOV_ASL, AND_ADD,AND_SUB,AND_CMP,AND_MOV,AND_ASR,AND_LSR,AND_ASL, OR_ADD,OR_SUB,OR_CMP,OR_MOV,OR_ASR,OR_LSR,OR_ASL, XOR_ADD,XOR_SUB,XOR_CMP,XOR_MOV,XOR_ASR,XOR_LSR,XOR_ASL, DMACH_ADD,DMACH_SUB,DMACH_CMP,DMACH_MOV,DMACH_ASR,DMACH_LSR,DMACH_ASL, SAT16_ADD,SAT16_SUB,SAT16_CMP,SAT16_MOV,SAT16_ASR,SAT16_LSR,SAT16_ASL, SWHW_ADD,SWHW_SUB,SWHW_CMP,SWHW_MOV,SWHW_ASR,SWHW_LSR,SWHW_ASL, MOV_Lcc
UDF-typed instruction	UDF00,UDF01,UDF02,UDF03UDF05,UDF06,UDF07,UDF08,UDF09,UDF12,UDF13,UDF15
Floating-point operational instruction	FMOV,FABS,FNEG,FCMP,FRSQRT,FADD,FSUB,FMUL,FDIVFMADD,FMSUB,FNMADD,FNMSUB,FBCC,FLCC

3-1 Transfer instructions

Transfer instructions are used to transfer data between registers, between memory and registers. Transfer instructions are grouped as MOV-typed instructions, EXT-typed instructions, and CLR-typed instructions. MOV-typed instructions provide data transfer functions using various addressing modes. Depending on the operation, displacement and immediate values also carry a sign extension. EXT-typed instructions provide transfer functions between registers with a sign extension. The CLR-typed instruction clears the contents of registers (by providing a function that transfer "0" into the registers.) Except for the CLR-typed instruction, none of these instructions generates any changes in flags.

Instruction	Operation
MOV	Word transfer between registers, Transfer of immediate values to registers Word transfer between a register and memory (load/store) Word transfer with post-increment between a register and memory (load/store)
MOVU	Immediate value transfer to registers (immediate value with zero extension)
MOVHU	Zero-extension and half-word transfer between a register and memory (load/store) Zero-extension and half-word transfer with post-increment between a register and memory (load/store)
MOVB	Zero-extension byte transfer between a register and memory (load/store)
MOVM	Block transfer between multiple registers and memory (load/store)
EXT	64-bit signed extension for word data (processing between registers)
EXTH	32-bit signed extension for half-word data (processing between registers)
EXTHU	32-bit zero extension for half-word data (processing between registers)
EXTB	32-bit signed extension for byte data (processing between registers)
EXTBU	32-bit zero extension for byte data (processing between registers)
CLR	Data clear (transfers 0 to registers)

3-2 Arithmetic operation instructions

The arithmetic operation instructions perform arithmetic operations on source operands and store the results in a register. These instructions may cause changes in flags. The "+1" and "+4" addition instructions, which are frequently used for address calculation, have been established as separate instructions.

Instruction	Operation
ADD	Addition between registers, addition between an immediate value and a register
ADDC	Addition with carry between registers, addition with carry between an immediate value and a register
SUB	Subtraction between registers, subtraction between an immediate value and a register
SUBC	Subtraction with borrow between registers, subtraction with borrow between an immediate value and a register
MUL	Signed multiplication between registers, signed multiplication between an immediate value and a register
MULU	Unsigned multiplication between registers, unsigned multiplication between an immediate value and a register
DIV	Signed division between registers, signed division between an immediate value and a register

DIVU	Unsigned division between registers, unsigned division between an immediate value and a register
INC	Adds "1" to the value stored in a register
INC4	Adds "4" to the value stored in a register

3-3 Compare instruction

The compare instruction compares the contents of two registers, or compares an immediate value with the content of a register. This instruction is primarily used ahead of a condition branch instruction. This instruction may cause changes in flags.

Instruction	Operation
CMP	Comparison of the contents of two registers, or the contents of an immediate value and a register

3-4 Logical operation instruction

The logical operation instructions perform logical operations on source operands and store the results in a register. These instructions may cause changes in flags.

Instruction	Operation
AND	AND operation between registers, or AND operation between an immediate value and a register
OR	OR operation between registers, or OR operation between an immediate value and a register
XOR	Exclusive OR operation between registers, or exclusive OR operation between an immediate value and a register
NOT	Inversion of all bits in a register (one's complement processing)

3-5 Bit manipulation instructions

The bit manipulation instructions perform bit manipulation operations between immediate values and the contents of registers; between immediate values and the contents of memory, or between the contents of registers and the contents of memory. These instructions may cause changes in flags.

Instruction	Operation
BTST	Multiple bit test (between an immediate value and a register, between an immediate value and memory)
BSET	Multiple bit test and set (between a register and memory, between an immediate value and memory)
BCLR	Multiple bit test and clear (between a register and memory, between an immediate value and a register)

3-6 Shift instructions

The shift instructions perform bit shifts of the specified amount. Regardless of the amount of the shift, the instructions can be performed in one cycle. These instructions may cause changes in flags.

Instruction	Operation
ASR	Arithmetic shift right any number of bits
LSR	Logic shift right any number of bits
ASL	Arithmetic shift left any number of bits
ASL2	Arithmetic shift left two bits
ROR	Rotate right one bit
ROL	Rotate left one bit

3-7 NOP instruction

NOP instruction performs no operation.

Instruction	Operation
NOP	No operation

3-8 Branch instructions

Branch instructions are instructions that change the flow of program execution according to some conditions. There are two types of conditional branch instructions: normal conditional branch instructions and loop-only conditional branch instructions. The loop-only conditional branch instructions minimize the branching penalty and permit fast loop execution by using dedicated registers. Subroutine calls and returns are a highly functional method of manipulating the PC, saving /restoring multiple registers to/from the stack, and allocating/releasing stack area.

Instruction	Operation
Bcc	Conditional branch (branches to PC-relative address)
Lcc	Loop-dedicated conditional branch (branches to start of loop set by SETLB)
SETLB	Registration of loop start information
JMP	Unconditional branch (PC relative, register indirect)
CALL	Subroutine call (saves next PC and multiple registers to stack, and allocates stack area)
RET	Return from subroutine (restores stack contents and releases stack area)
RETF	Return from subroutine (restores stack contents and releases stack area)
RETS	Return from subroutine (restores PC only)
RTI	Return from interrupt program
TRAP	Subroutine call to a specific address
SYSCALL	System call

3-9 Debug instruction

This instruction is used by debuggers and is reserved for debuggers.

Instruction	Operation
PI	This instruction is reserved by a debugger. Normally, an unimplemented instruction exception occurs when this instruction is executed.

3-10 Extended operation instructions

Extended operation instructions are defined for an add-on typed extended operation unit. The instruction formats are predefined and the instruction map is also reserved. In addition to maintaining compatibility with the extended instructions that were implemented in the AM 31 microprocessor core, the AM 33 microprocessor core also supports dual multiplication/dual multiply-and-accumulate operations that multiply, in parallel, two 16-bit data values that are packed in word data. These functions accelerate audio data signal processing.

Instruction	Operation
DMULH	Signed dual multiplication between registers, signed dual multiplication between an immediate value and a register
DMULHU	Unsigned dual multiplication between registers, unsigned dual multiplication between an immediate value and a register
DMACH	Signed dual sum-of-products operation between registers, signed dual sum-of-products operation between an immediate value and a register
DMACHU	Unsigned dual sum-of-products operation between registers, unsigned dual sum-of-products operation between an immediate value and a register
MAC	Signed sum-of-products operation between registers, signed sum-of-products operation between an immediate value and a register
MACU	Unsigned sum-of-products operation between registers, unsigned sum-of-products operation between an immediate value and a register
MACH	Signed half-word sum-of-products operation between registers, signed half-word sum-of-products operation between an immediate value and a register
MACHU	Unsigned half-word sum-of-products operation between registers, unsigned half-word sum-of-products operation between an immediate value and a register
MACB	Signed byte sum-of-products operation between registers, signed byte sum-of-products operation between an immediate value and a register
MACBU	Unsigned byte sum-of-products operation between registers, unsigned byte sum-of-products operation between an immediate value and a register
SWHW	Data ordering swap (half-word reordering within a word)
SWAP	Data ordering swap (byte reordering within a word)
SWAPH	Data ordering swap (byte reordering within a half-word)
SAT16	16-bit saturation processing
SAT24	24-bit saturation processing
MCSTE	Sum-of-products operation result saturation processing
BSCH	Bit search

3-11 LIW extended operation instructions

The LIW extended operation instructions perform two operations in a single instruction. For details on each instruction, refer to the instruction specifications in chapter 2.

Instruction	Operation
ADD_OP2	Parallel execution of addition between registers, and OP2 ADD_ADD, ADD_SUB, ADD_CMP, ADD_MOV, ADD_ASR, ADD_LSR, ADD_ASX
SUB_OP2	Parallel execution of subtraction between registers, and OP2 SUB_ADD, SUB_SUB, SUB_CMP, SUB_MOV, SUB_ASR, SUB_LSR, SUB_ASX
CMP_OP2	Parallel execution of comparison between registers, and OP2 CMP_ADD, CMP_SUB, CMP_MOV, CMP_ASR, CMP_LSR, CMP_ASX
AND_OP2	Parallel execution of AND operation between registers, and OP2 AND_ADD, AND_SUB, AND_CMP, AND_MOV, AND_ASR, AND_LSR, AND_ASX
OR_OP2	Parallel execution of OR operation between registers, and OP2 OR_ADD, OR_SUB, OR_CMP, OR_MOV, OR_ASR, OR_LSR, OR_ASX
XOR_OP2	Parallel execution of XOR operation between registers, and OP2 XOR_ADD, XOR_SUB, XOR_CMP, XOR_MOV, XOR_ASR, XOR_LSR, XOR_ASX
DMACH_OP2	Parallel execution of signed dual sum-of-products operation between registers, and OP2 DMACH_ADD, DMACH_SUB, DMACH_CMP, DMACH_MOV, DMACH_ASR, DMACH_LSR, DMACH_ASX
SAT16_OP2	Parallel execution of 16-bit saturation processing, and OP2 SAT16_ADD, SAT16_SUB, SAT16_CMP, SAT16_MOV, SAT16_ASR, SAT16_LSR, SAT16_ASX
SWHW_OP2	Parallel execution of half-word reordering within word, and OP2 SWHW_ADD, SWHW_SUB, SWHW_CMP, SWHW_MOV, SWHW_ASR, SWHW_LSR, SWHW_ASX

3-12 Floating-point operation instructions

Floating -point operations are executed through using the floating-point operation unit. For details on each instruction, refer to the instruction specifications in chapter 2.

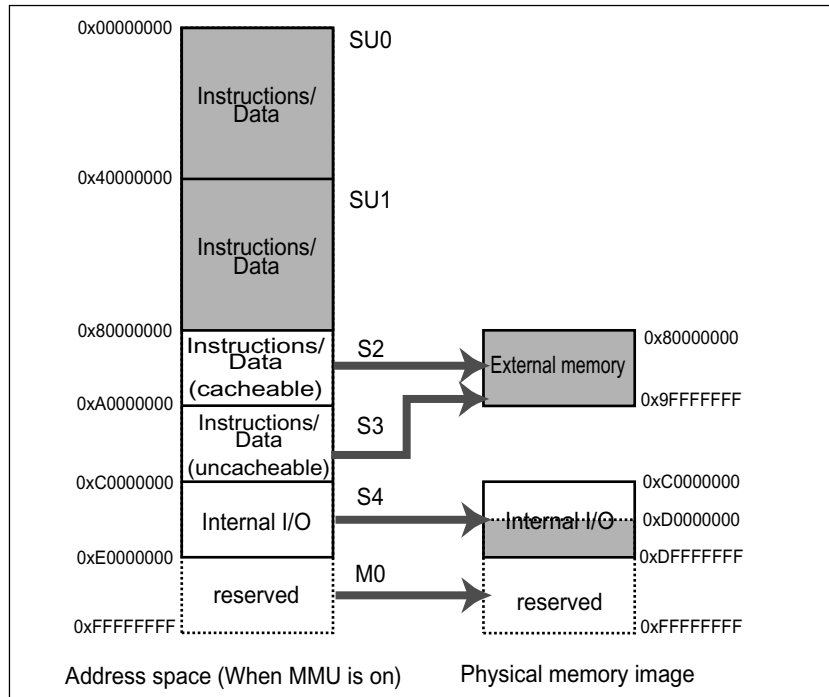
Instruction	Operation
FMOV	Data transfer between floating-point registers, transfer of immediate value to a floating-point register Data transfer between a floating-point register and memory (load/store) Data transfer with post-increment between a floating-point register and memory (load/store) Data transfer between a floating-point register and a register
FABS	Absolute value operation between floating-point registers
FNEG	Sign inversion operation between floating-point registers
FRSQRT	Square root reciprocal operation between floating-point registers ($1/\sqrt{}$)
FCMP	Compare operation between floating-point registers
FADD	Addition between floating-point registers, addition between an immediate value and a floating-point register
FSUB	Subtraction between floating-point registers, subtraction between an immediate value and a floating-point register
FMUL	Multiplication between floating-point registers, multiplication between an immediate value and a floating-point register
FDIV	Division between floating-point registers, division between an immediate value and a floating-point register
FMADD FNMADD	Compound multiplication and addition operation between floating-point registers
FMSUB FNMSUB	Compound multiplication and subtraction operation between floating-point registers
FBCC	Conditional branch by determining the floating-point conditions. (Branch of the PC relative address)
FLCC	Loop-only conditional branch by determining the floating-point conditions. (Branch to the loop head set by SETLB)

4

Memory space

Address space when using an MMU (logical address space)

The AM33 microprocessor core supports a 32-bit logical address space, and can access a 4 GB logical address space. The logical address space is divided into the following six areas: SU0, SU1, S2, S3, S4, and M0.



The SU0 space (0x00000000 to 0x3FFFFFFF: 1 GB) and the SU1 space (0x40000000 to 0x7FFFFFFF: 1 GB) can both be used at the user level and the supervisor level, and are mapped onto the physical address space in page units according to the content of the address translation table.

Caching control for the SU0 space and the SU1 space is controlled as follows, according to the value of the MMUCTR.CE (Cacheable bit Enable) bit:

1. When the MMUCTR.CE bit is set to "0"

SU0 space: Cacheable

SU1 space: Uncacheable

2. When the MMUCTR.CE bit is set to "1"

SU0 space: Whether each page is cacheable or uncacheable can be individually controlled through the status of the PTE.C bit.

SU1 space: Whether each page is cacheable or uncacheable can be individually controlled through the status of the PTE.C bit.

The physical address spaces onto which the SU0 and SU1 spaces can be mapped are 0x40000000 to 0xBFFFFFFF in external memory for instruction accesses, 0x40000000 to 0xBFFFFFFF in external memory and 0x0xD0000000 to 0xDFFFFFFF in the internal I/O space for data accesses.

The S2 (0x80000000 to 0x9FFFFFFF: 0.5 GB) and S3 (0xA0000000 to 0xBFFFFFFF: 0.5 GB) spaces are supervisor level-only spaces; the logical addresses are mapped to physical addresses 0x80000000 to 0x9FFFFFFF in a fixed manner.

The S4 (0xC0000000 to 0xDFFFFFFF: 0.5 GB) space is a supervisor level-only space; the logical addresses are mapped to the internal I/O space 0xC0000000 to 0xDFFFFFFF in a fixed manner.

The M0 (0xE0000000 to 0xFFFFFFFF: 0.5 GB) space is a reserved space and cannot be accessed.



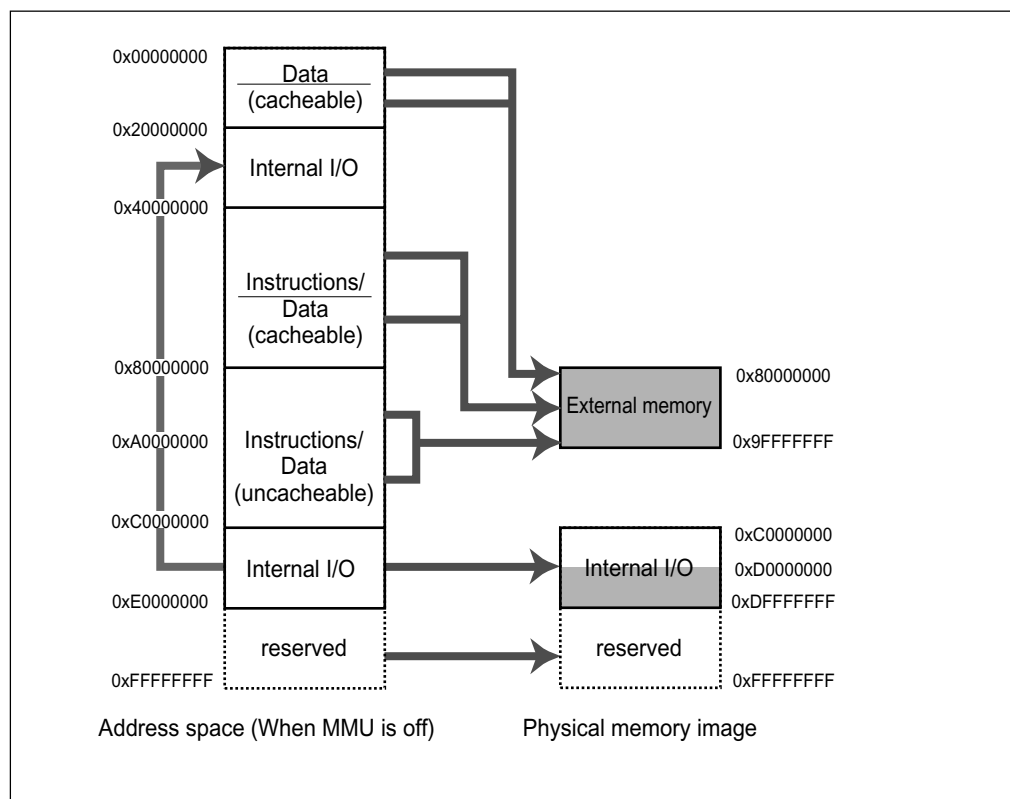
Logical addresses cannot be mapped to the control register space, which is the space with addresses from 0xC0000000 to 0xCFFFFFFF in the internal I/O space. This space can be accessed through the S4 area at the privileged level or higher.

Pages must be aligned by the page size. The hardware cannot convert addresses properly if page sizes other than those described above are assigned.

The instruction MMU cannot map the internal I/O space to the SU0/SU1 areas.

Address space when not using an MMU

The AM33 microprocessor core supports a 32-bit address space, and is upwardly compatible with the AM30/AM31/AM32 microprocessor core and memory map. As shown in the below figure, the addresses are mapped to the physical address space in a fixed manner.



In order to maintain compatibility with the AM30/AM31/AM32 microprocessor core, the internal I/O space 0x20000000 to 0x3FFFFFFF is mirrored on 0xC0000000 to 0xDFFFFFFF.

5

Addressing mode

There are the following six addressing modes that are frequently used by the compiler.

Data transfer instructions permit the use of six different addressing modes: register direct, immediate, register indirect, register relative indirect, absolute, and index qualifier register indirect.

Register operation instructions permit the use of two addressing mods: register direct and immediate.

Register indirect with indexed addressing is used to efficiently access data in an array, etc.

Addressing mode	Address calculation	Logical address
Register direct Rm/Rn/XRm/XRn MDR/PSW/EPsw/SP	_____	_____
Immediate value imm8/regs,imm16 imm24,imm32 imm40,imm48	_____	_____
Register indirect (Rm)/(Rn) (Rm+)/(Rn+)	$\begin{array}{ c } \hline \text{Rm / Rm+ / Rn / Rn+} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div>	$\begin{array}{ c } \hline \text{32-bit address} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div>
Register relative indirect (d8,Rm)/(d8,Rn) (d16,Am)/(d16,An) (d24,Rm)/(d24,Rn) (d32,Rm)/(d32,Rn) d8,d16,d24 : Sign extension	$\begin{array}{ c } \hline \text{Rm / Am / Rn / An} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div> <p style="text-align: center;">+</p> $\begin{array}{ c } \hline \text{d32 / d24 / d16 / d8} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 23 15 7 0 </div>	$\begin{array}{ c } \hline \text{32-bit address} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div>
	$\begin{array}{ c } \hline \text{PC} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div> <p style="text-align: center;">+</p> $\begin{array}{ c } \hline \text{d32 / d16 / d8} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 15 7 0 </div>	
	$\begin{array}{ c } \hline \text{SP} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div> <p style="text-align: center;">+</p> $\begin{array}{ c } \hline \text{d32 / d24 / d16 / d8} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 23 15 7 0 </div>	
Absolute (abs8),(abs16) (abs24),(abs32) d8,d16,d24 : Zero extension	$\begin{array}{ c } \hline \text{abs32 / 24 / 16 / 8} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 23 15 7 0 </div>	$\begin{array}{ c } \hline \text{32-bit address} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div>
Indexed register indirect (Ri,Rm)/(Ri,Rn)	$\begin{array}{ c } \hline \text{Rm / Rn} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div> <p style="text-align: center;">+</p> $\begin{array}{ c } \hline \text{Ri} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div>	$\begin{array}{ c } \hline \text{32-bit address} \\ \hline \end{array}$ <div style="display: flex; justify-content: space-between; width: 100%;"> 31 0 </div>

5-1 Register direct

The register-direct addressing mode specifies a register directly. Registers which can be specified are as follows:

Rm/Rn:

XRm/XRn:

MDR: Multiply/divide register (32 bits)

PSW: Processor status word (16 bits)

EPSW: Processor status word (32 bits)

SP: Stack pointer (32 bits)

5-2 Immediate

The immediate addressing mode directly specifies transferred values through operand value added to instruction codes, mask value, and multiple registers to be transferred (regs).

The immediate sizes are 8 bits (imm8), 16 bits (imm16), 24 bits (imm24), 32 bits (imm32), 40 bits (imm40), and 48 bits (imm48).

As shown below, regs (8-bit immediate) can specify the registers of D2, D3, A2, and A3, and other registers (other, EXREG0, EXREG1, EXOTHER)

7	6	5	4	3	2	1	0
D2	D3	A2	A3	other	EXREG0	EXREG1	EXOTHER

other: D0, D1, A0, A1, MDR, LIR, LAR

EXREG0: E2, E3

EXREG1: E4, E5, E6, E7

EXOTHER: E0, E1, MDRQ, MCRH, MCRL, MCVF

5-3 Register indirect

The 32-bit address which is shown by the address register (An/Am) is effective in the register-indirect addressing mode.

Register-indirect description: (Rm) / (Rn)

(Rm+) / (Rn+)

5-4 Register relative indirect

The register relative-indirect addressing mode is addressing shown by addition of the address register (An/Am) or program counter (PC), stack pointer (SP) and displacement. The displacement sizes are 8 bits, 16 bits, and 32 bits.

Address computations are carried out for the address register (Am/An) or program counter (PC) and sign-extended 8/16 or 32-bit displacement, and for the stack pointer (SP) and zero-extended 8/16 or 32-bit displacement.

Register relative-indirect description: (d8, Rm) / (d8, Rn) : d8 is code-extended.
 (d16, Am) / (d16, An): d16 is code-extended.
 (d24, Rm) / (d24, Rn): d24 is code-extended.
 (d32, Rm) / (d32, Rn):
 (d8, PC): d8 is zero-extended
 (d16, SP): d16 is zero-extended.
 (d24, SP): d24 is zero-extended.
 (d32, SP):

5-5 Absolute

The absolute addressing mode specifies directly a 32-bit address through the 16, 24, and 32-bit operand value added to the instruction code.

Absolute description: (abs 16): abs 16 is zero-extended.
 (abs 24): abs 24 is zero-extended.
 (abs 32):

5-6 Register indirect with indexed addressing

The register indirect with indexed addressing is addressing shown by the address register (Rm/Rn) and data register (Ri). The 32-bit contents of the address registers (Rm/Rn) and data registers (Ri) are added.

Description of the register indirect with indexed addressing: (RI, Rm) / (Ri, Rn)

6 Instruction formats

There are 14 instruction formats. The instruction set has a variable word length in which the basic word length is one byte, and the instruction length can vary in units of one byte. The shortest format is S0 format, which is one-byte long. The longest formats are S6 format, D5 format, and T4 format, each of which is seven-byte long.

Format S0	OP
Format S1	OP imm8/d8
Format S2	OP imm16/d16/abs16
Format S4	OP imm32/d32/abs32
Format S6	OP imm48
Format D0	OP OP
Format D1	OP OP imm8/d8
Format D2	OP OP imm16/d16/abs16
Format D3	OP OP imm24
Format D4	OP OP imm32/d32/abs32
Format D5	OP OP imm40
Format T0	OP OP OP
Format T1	OP OP OP imm8/d8
Format T3	OP OP OP imm24/d24/abs24
Format T4	OP OP OP imm32/d32/abs32
Format Q0	OP OP OP OP

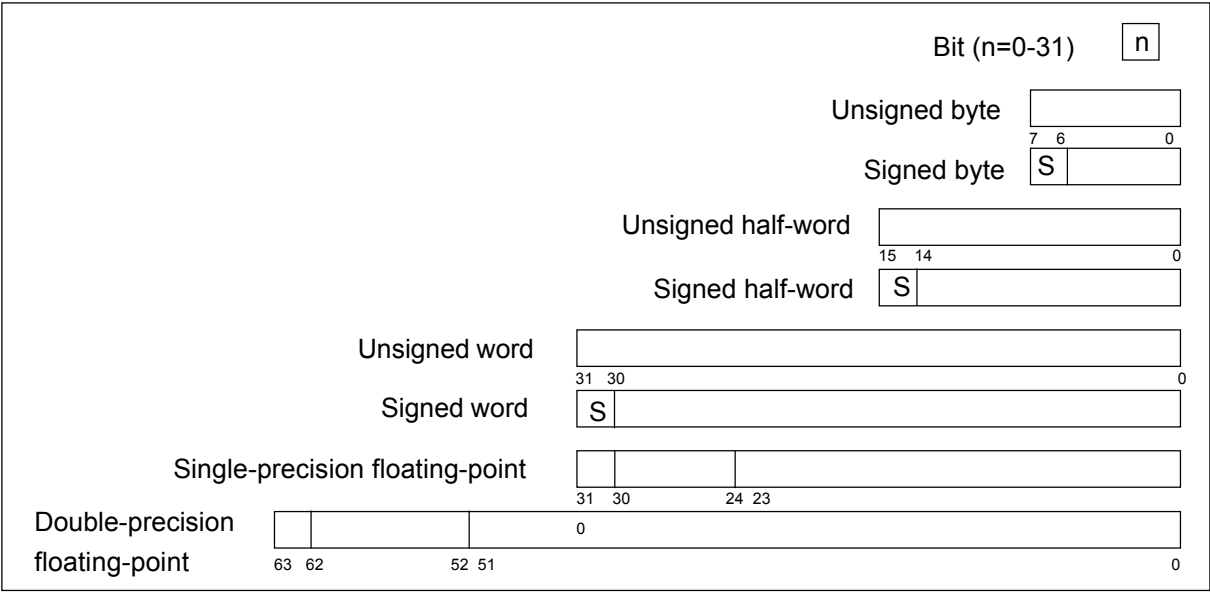
Normally, the opcode is followed by an 8-, 16-, or 32-bit immediate value, displacement value, or absolute value.

However, in instructions with formats S2, S4, S6, D2, and D5, the opcode is followed by two or more immediate values, displacement values, or absolute values; these are noted as a whole as 16-bit immediate values (imm 16), 24-bit immediate values (imm 24), 32-bit immediate values (imm 32), 40-bit immediate values (imm 40), and 48-bit immediate values (imm 48). According to these notations, the following instructions accept 16-, 24-, 32-, 40-, or 48-bit immediate values.

imm16:	RET	regs, imm8
	RETF	regs, imm8
	BTST	imm8, (d8, An)
	BSET	imm8, (d8, An)
imm24:	BCLR	imm8, (d8, An)
	BTST	imm8, (abs16)
	BSET	imm8, (abs16)
imm32:	BCLR	imm8, (abs16)
	CALL	(d16, PC), regs, imm8
	BTST	imm8, (abs32)
imm40:	BSET	imm8, (abs32)
	BCLR	imm8, (abs32)
imm48:	CALL	(d32, PC), regs, imm8

6-1 Data formats

There are four data types for integer data: bit, byte, half-word, and word. Byte data, half-word data, and word data can all be handled as either signed data or unsigned data. In the case of signed data, the MSB is the sign bit. For floating-point values, there are two formats: single precision floating-point values and double-precision floating-point values.



Integer data in memory must be aligned properly. In short, the two bits on the LSB side of an address where word data is stored must be "00" (i.e. the address is a multiple of 4), and the one bit on the LSB side of an address where half-word data is stored must be "0" (i.e. the address is a multiple of 2). If data that is not properly aligned is accessed, a misalignment exception is generated.

Floating-point data in memory must also be aligned in a similar fashion. In short, the two bits on the LSB side of an address where single-precision floating-point data is stored must be "00" (i.e. the address is a multiple of 4). When double-precision floating-point data is stored in memory and the double-precision floating-point load/store instruction (FMOV) is being used to access memory, the three bits on the LSB side of the address must be "000" (i.e. the address is a multiple of 8). When using two single-precision floating-point load/store instructions to access data, the two bits on the LSB side of the address must be "00" (i.e. the address is a multiple of 4).

Bit No.	31	24	23	16	15	8	7	0
Memory address	(4n+3)		(4n+2)		(4n+1)		(4n)	
Word data	Address : 4n							
Half-word data	Address : 4n+2				Address : 4n			
Byte data	Address : 4n+3		Address : 4n+2		Address : 4n+1		Address : 4n	

6-2 Endian

The bytes are positioned according to Little Endian format. Therefore, the address of the byte data that is on the MSB side of half-word data is one greater than the address of the byte data that is on the LSB side of the half-word data. The address of the byte data that is on the MSB side of word data is three greater than the address of the byte data that is on the LSB side of the word data. Concerning bit numbers, in the case of that the bit on the LSB side is regarded as "bit 0", the bit numbers increase sequentially towards the MSB.

When 16-bit immediate/displacement/absolute (imm16/d16/abs16) and 32-bit immediate/displacement/absolute (imm32/d32/abs32) follow the opcode, the bytes are partitioned per 8 bits from the lowest, and positioned in the order from the lower address to higher address. (Little Endian)

[e.g.] 32-bit immediate value, 0x1234567 is arrayed on the memory in the Little Endian format.

Address n	0x67
Address n+1	0x45
Address n+2	0x23
Address n+3	0x01

However, the instructions of S2, S4, S6, D2, D3, and D5 in the instruction formats have above-2 immediate, displacement, and absolute values connected after the opcode, and compose as a whole 16-bit immediate values (imm16), 24-bit immediate values (imm24), 32-bit immediate values (imm32), 40-bit immediate values (imm40), and 48-bit immediate values (imm48). They are shown below. The 16-bit displacement (d16) and 32-bit displacement/absolute values (d32/abs32) in the below figures are positioned in the Little Endian format as same as the above example.

RET regs, imm8 / RETF regs, imm8

Address n	RET/RETF	... Opcode of RET or RETF
Address n+1	regs	
Address n+2	imm8	

BTST imm8,(d8,An) / BSET imm8,(d8,An) / BCLR imm8,(d8,An)

Address n	BTST/BSET/BCLR	} ... Opcode of BTST or BSET
Address n+1		
Address n+2	d8	
Address n+3	imm8	

BTST imm8,(abs16) / BSET imm8,(abs16) / BCLR imm8,(abs16)

Address n	BTST/BSET/BCLR	} ... Opcode of BTST or BSET
Address n+1		
Address n+2	abs16	
Address n+3		
Address n+4	imm8	

CALL (d16,PC), regs, imm8

Address n	CALL	... Opcode of CALL
Address n+1		
Address n+2	abs16	
Address n+3	reg	
Address n+4	imm8	

BTST imm8,(abs32) / BSET imm8,(abs32) / BCLR imm8,(abs32)

Address n	BTST/BSET/BCLR	} ... Opcode of BTST, BSET or BCLR
Address n+1		
Address n+2		
Address n+3	abs32	
Address n+4		
Address n+5		
Address n+6	imm8	

CALL (d32, PC), regs, imm8

Address n	CALL	... Opcode of CALL
Address n+1		
Address n+2		
Address n+3	abs32	
Address n+4		
Address n+5	regs	
Address n+6	imm8	

Notations

The symbols used in the instruction description are shown below.

op1,op2,op3	: Instruction operation
Reg1,Reg2,Reg3,Reg4	:Register (used as the general meaning)
Am, An	:Address register (m, n = 3-0)
Dm, Dn,Di	:Data register (m, n, i = 3-0)
Rm,Rn,Rd,Rd1,Rd2	:CPU general register
	A0-A3: Address register
	D0-D3: Data register
	E0-E7: Extension register
MDR	:Multiply divide register
EPSW	:Processor status word (32 bits)
PSW	:Processor status word (lower 16 bits)
	PSW[15:0] = EPSW[15:0]
MSP	:Monitor-level stack pointer
SSP	:Supervisor-level stack pointer
USP	:User-level stack pointer
SP	:Stack pointer
	(In other words, SP refers to USP when the current level is the user level, to SSP when it is the supervisor level, and to MSP when it is monitor level.)
PC	:Program counter
nPC	:Next instruction PC
LIR	:Loop instruction register
LAR	:Loop address register
MDRQ	:Extension multiply register
MCRH	:Multiply-accumulate upper 32-bit register
MCRL	:Multiply-accumulate lower 32-bit register
MCVF	:Multiply-accumulate overflow flag register
TMP	:Temporary register
{MCRH,MCRL}	:64-bit multiply-accumulate register
	(The upper 32 bits consists of MCRH, the lower 32 bit consists of MCRL.)
FSm,FSm1,FSm2,FSm3,FSn	:FPU single precision register
FDm,FDn	:FPU double precision register
Mem	:Memory (used as general meaning)
imm	:immediate value (used as general meaning)
imm4	:4-bit immediate value
imm8	:8-bit immediate value
imm16	:16-bit immediate value
imm24	:24-bit immediate value
imm32	:32-bit immediate value
disp	:diaplacement
d8	:8-bit displacement
d16	:16-bit displacement
d24	:24-bit displacement
d32	:32-bit displacement

abs8	:8-bit absolute
abs16	:16-bit absolute
abs24	:24-bit absolute
abs32	:32-bit absolute
()	:Indirect addressing
For details, refer to Chapter 1, Instruction description, 5. Addressing mode.	
regs	:Multiple register definition
0x	:hexadecimal (The numbers following 0x indicates hexadecimal.)
.bpn	:Bit position (n indicates the bit positions 0-31.)
[n]	:Bit position (n indicates the bit position.)
.lsb	:Bit position (Bit 0)
.msb	:Bit position (Bit 31)
+	:Addition
-	:Subtraction
*	:Multiplication
/	:Division
Sqrt	:Square root
%	:Surplus
&	:Logical product
	:Logical sum
^	:Exclusive logical sum
~	:Bit reverse
<<n	:n-bit left shift
>>n	:n-bit right shift
->	:Transfer
:	:Reflection of operation results
(sign_ext)	:Sign extension
(zero_ext)	:Zero extension
label	:Address
VF	:Overflow flag
CF	:Carry flag
NF	:Negative flag
ZF	:Zero flag
mem8(xxx)	:8-bit data within the memory addressed by xxx
mem16(xxx)	:16-bit data within the memory addressed by xxx
mem32(xxx)	:32-bit data within the memory addressed by xxx
mem_cline(xxx)	:Data cache line data within the memory addressed by xxx
CodeSize	:Code size of assembler mnemonic

The symbols used in the flag change tables are shown below.

("Flag" is used generically for the lower 4 bits (V, C, N and Z) in PSW.)

Δ	:With flag change
—	:With no flag change
0	:Always "0"
1	:Always "1"
?	:Undefined
*	:Defined by user



: The instruction with this mark is implemented in AM33-1.



:The instruction with this mark is implemented in AM33-2.



:The instruction with this mark is implemented in AM33-2A.



:The instruction with this mark is implemented in AM34-1.

MOV

Transfer

MOV Reg1, Reg2						
Operation	Reg1 -> Reg2 This instruction transfers the contents of Reg1 to Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Am, An	n=m cannot be specified.	–	–	–	–	1
mov Dm, Dn	n=m cannot be specified.	–	–	–	–	1
mov Am, Dn		–	–	–	–	2
mov Dm, An		–	–	–	–	2
mov Am, Rn		–	–	–	–	2
mov Rm, An		–	–	–	–	2
mov Dm, Rn		–	–	–	–	2
mov Rm, Dn		–	–	–	–	2
mov Rm, Rn		–	–	–	–	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV imm, Reg						
Operation	imm -> Reg This instruction transfers imm to Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov imm8, An	imm8 is zero-extended.	–	–	–	–	2
mov imm16, An	imm16 is zero-extended.	–	–	–	–	3
mov imm32, An		–	–	–	–	6
mov imm8, Dn	imm8 is zero-extended.	–	–	–	–	2
mov imm16, Dn	imm16 is zero-extended.	–	–	–	–	3
mov imm32, Dn		–	–	–	–	6
mov imm8, Rn	imm8 is zero-extended.	–	–	–	–	4
mov imm24, Rn	imm24 is zero-extended.	–	–	–	–	6
mov imm32, Rn		–	–	–	–	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV MSP, An <Privileged instruction>

Operation	MSP -> An In the monitor level, this instruction transfers the contents of the monitor-level stack pointer (MSP) to the address register (An).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov MSP, An		-	-	-	-	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor level.
When the instruction is used in the supervisor and user levels, a system exception, privileged instruction execution exception, occurs.

MOV Am, MSP <Privileged instruction>

Operation	Am -> MSP In the monitor level, this instruction transfers the contents of the address register (Am) to the monitor-level stack pointer (MSP).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Am, MSP		-	-	-	-	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor level.
When the instruction is used in the supervisor and user levels, a system exception, privileged instruction execution exception, occurs.

MOV SSP, An <Privileged instruction>

Operation	SSP -> An In the monitor or supervisor level, this instruction transfers the contents of the supervisor-level stack pointer (SSP) to the address register (An).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov SSP, An		–	–	–	–	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV Am, SSP <Privileged instruction>

Operation	Am -> SSP In the monitor or supervisor level, this instruction transfers the contents of the address register (Am) to the supervisor-level stack pointer (SSP).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Am, SSP		–	–	–	–	1
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV USP, An <Privileged instruction>

Operation	USP -> An In the monitor or supervisor level, this instruction transfers the contents of the user-level stack pointer (USP) to the address register (An).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov USP, An		-	-	-	-	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV Am, USP <Privileged instruction>

Operation	Am ->USP In the monitor or supervisor level, this instruction transfers the contents of the address register (Am) to the user-level stack pointer (USP).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Am, USP		-	-	-	-	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV EPSW, Dn <Privileged instruction>

Operation	EPSW -> Dn In the monitor or supervisor level, this instruction transfers the contents of the processor-status-word (EPSW) to the data register (Dn).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov EPSW, Dn		–	–	–	–	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV Dm, EPSW <Privileged instruction>

Operation	Dm -> EPSW In the monitor or supervisor level, this instruction transfers the contents of the data register (Dm) to the processor-status-word (EPSW).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Dm, EPSW		●	●	●	●	2
Flag change						
VF: Bit 3 of Dm. CF: Bit 2 of Dm. NF: Bit 1 of Dm. ZF: Bit 0 of Dm.						



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV PSW, Dn <Privileged instruction>

Operation	PSW[15:0] -> Dn[15:0] 0x0000 -> Dn[31:16] In the monitor or supervisor level, this instruction transfers the zero-extended 16-bit processor-status-word (PSW) to the data register (Dn).								
Assembler mnemonic	Note				V	C	N	Z	Size
mov PSW, Dn					-	-	-	-	2
Flag change									
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.									



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV Dm, PSW <Privileged instruction>

Operation	Dm[15:0] -> EPSW In the monitor or supervisor level, this instruction transfers the lower 16 bits of the data register (Dm) to the processor-status-word (PSW). (The upper 16 bits are ignored.)								
Assembler mnemonic	Note				V	C	N	Z	Size
mov Dm, PSW					●	●	●	●	2
Flag change									
VF: Bit 3 of Dm. CF: Bit 2 of Dm. NF: Bit 1 of Dm. ZF: Bit 0 of Dm.									



This instruction is valid only in the monitor or supervisor level.
 When the instruction is used in the user level, a system exception, privileged instruction execution exception, occurs.

MOV MDR, Dn						
Operation	<p>MDR -> Dn</p> <p>This instruction transfers the contents of the multiply/divide register (MDR) to the data register (Dn).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mov MDR, Dn		–	–	–	–	2
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

MOV Dm, MDR						
Operation	<p>Dm -> MDR</p> <p>This instruction transfers the contents of the data register (Dm) to the multiply/divide register (MDR).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Dm, MDR		–	–	–	–	2
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

MOV SP, Reg

Operation	SP -> Reg This instruction transfers the contents of the stack pointer (SP) to the register (Reg).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov SP, An		–	–	–	–	1
mov SP, Rn		–	–	–	–	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV Reg, SP

Operation	Reg -> SP This instruction transfers the contents of Reg to SP.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Am, SP		–	–	–	–	2
mov Rm, SP		–	–	–	–	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV imm, SP

Operation	imm -> SP This instruction transfers imm to SP.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov imm8, SP	imm8 is zero-extended.	–	–	–	–	4
mov imm24, SP	imm24 is zero-extended.	–	–	–	–	6
mov imm32, SP		–	–	–	–	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV MDRQ, Rn									
Operation	MDRQ -> Rn								
	This instruction transfers the contents of the extended multiply register (MDRQ) to Rn.								
Assembler mnemonic		Note			V	C	N	Z	Size
mov MDRQ, Rn					0	0	Δ	Δ	3
Flag change									
VF: Always 0. CF: Always 0. NF: 1 when MSB of the transfer result is 1, 0 in all other cases. ZF: 1 when the transfer result is 0, 0 in all other cases.									

MOV Rm, MDRQ										
Operation	Rm -> MDRQ									
	This instruction transfers the contents of Rm to MDRQ.									
Assembler mnemonic		Note				V	C	N	Z	Size
mov Rm, MDRQ						-	-	-	-	3
Flag change										
VF: This is not changed.										
CF: This is not changed.										
NF: This is not changed.										
ZF: This is not changed.										

MOV imm, MDRQ							
Operation	imm -> MDRQ						
	This instruction transfers imm to MDRQ.						
Assembler mnemonic		Note	V	C	N	Z	Size
mov imm8, MDRQ		imm8 is zero-extended.	-	-	-	-	4
mov imm24, MDRQ		imm24 is zero-extended.	-	-	-	-	6
mov imm32, MDRQ			-	-	-	-	7
Flag change							
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.							

MOV MCRH, Rn

Operation	MCRH -> Rn MCVF -> EPSW.V This instruction transfers the contents of the upper 32 bits in the multiply-and-accumulate operation register (MCRH) to the Register (Rn). This transfers the contents of the multiply-and-accumulate overflow flag (MCVF) to the overflow flag of the processor status word (EPSW.V).				
Assembler mnemonic	Note	V	C	N	Z
mov MCRH, Rn		Δ	0	?	?
Flag change					
VF: 1 when MCVF is 1, 0 in all other cases. CF: Always 0. NF: Undefined. ZF: Undefined.					

MOV Rm, MCRH

Operation	Rm -> MCRH EPSW.V -> MCVF This instruction transfers the contents of the Register (Rm) to the upper 32 bits in the multiply-and-accumulate operation register (MCRH). This transfers the contents of the overflow flag of the processor status word (EPSW.V) to the multiply-and-accumulate overflow flag (MCVF).				
Assembler mnemonic	Note	V	C	N	Z
mov Rm, MCRH		-	-	-	-
Flag change					
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.					

MOV imm, MCRH

Operation	imm -> MCRH EPSW.V -> MCVF This instruction transfers imm to the upper 32 bits in the multiply-and-accumulate operation register (MCRH). This transfers the contents of the overflow flag of the processor status word (EPSW.V) to the multiply-and-accumulate overflow flag (MCVF).				
Assembler mnemonic	Note	V	C	N	Z
mov imm8, MCRH	imm8 is zero-extended.	-	-	-	-
mov imm24, MCRH	imm24 is zero-extended.	-	-	-	-
mov imm32, MCRH		-	-	-	-
Flag change					
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.					

MOV MCRL, Rn						
Operation	MCRL -> Rn MCVF -> EPSW.V This instruction transfers the contents of the lower 32 bits in the multiply-and-accumulate operation register (MCRL) to the Register (Rn). This transfers the contents of the multiply-and-accumulate overflow flag (MCVF) to the overflow flag of the processor status word (EPSW.V).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov MCRL, Rn		Δ	0	?	?	3
Flag change						
VF: 1 when MCVF is 1, 0 in all other cases. CF: Always 0. NF: Undefined. ZF: Undefined.						

MOV Rm, MCRL						
Operation	Rm -> MCRL EPSW.V -> MCVF This instruction transfers the contents of the Register (Rm) to the lower 32 bits in the multiply-and-accumulate operation register (MCRL). This transfers the contents of the overflow flag of the processor status word (EPSW.V) to the multiply-and-accumulate overflow flag (MCVF).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Rm, MCRL		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV imm, MCRL						
Operation	imm -> MCRL EPSW.V -> MCVF This instruction transfers imm to the lower 32 bits in the multiply-and-accumulate operation register (MCRL). This transfers the contents of the overflow flag of the processor status word (EPSW.V) to the multiply-and-accumulate overflow flag (MCVF).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov imm8, MCRL	imm8 is zero-extended.	-	-	-	-	4
mov imm24, MCRL	imm24 is zero-extended.	-	-	-	-	6
mov imm32, MCRL		-	-	-	-	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV MCVF, Rn

Operation	MCVF -> Rn[0] 0x00000000 -> Rn[31:1] This instruction transfers the contents of the multiply-and-accumulate overflow flag (MCVF) to the bit 0 of the register (Rn).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov MCVF, Rn		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV Rm, MDRQ

Operation	Rm[0] -> MCVF This instruction transfers the contents of bit 0 of Rm to MCVF.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Rm, MCVF		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV imm, MCVF

Operation	imm[0] -> MCVF This instruction transfers bit 0 of imm to MCVF.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov imm8, MCVF		-	-	-	-	4
mov imm24, MCVF		-	-	-	-	6
mov imm32, MCVF		-	-	-	-	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV PC, An						
Operation	PC -> An This instruction transfers the program counter (PC) of the current instruction to the register (An).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov PC, An		–	–	–	–	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOV Mem, Reg

Operation	<p>MOV (Reg1), Reg2 mem32(Reg1) -> Reg2</p> <p>MOV (disp, Reg1), Reg2 mem32(Reg1+disp) -> Reg2</p> <p>MOV (Reg1, Reg2), Reg3 mem32(Reg1+Reg2) -> Reg3</p> <p>MOV (abs), Reg1 mem32(abs) -> Reg1</p> <p>MOV (disp, SP), Reg1 mem32(SP+disp) -> Reg1</p> <p>This instruction performs the word-data transfer of the content of Mem to Reg.</p> <p>MOV (disp+Reg1), SP mem32(Reg1) -> SP</p> <p>This instruction performs the word-data transfer of the content of Mem to the stack pointer (SP).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mov (Am),Dn		-	-	-	-	1
mov (Am),An		-	-	-	-	2
mov (Rm),Rn		-	-	-	-	3
mov (d8,Am),Dn	d8 is sign-extended.	-	-	-	-	3
mov (d16,Am),Dn	d16 is sign-extended.	-	-	-	-	4
mov (d32,Am),Dn		-	-	-	-	6
mov (d8,Am),An	d8 is sign-extended.	-	-	-	-	3
mov (d16,Am),An	d16 is sign-extended.	-	-	-	-	4
mov (d32,Am),An		-	-	-	-	6
mov (d8,Rm),Rn	d8 is sign-extended.	-	-	-	-	4
mov (d24,Rm),Rn	d24 is sign-extended.	-	-	-	-	6
mov (d32,Rm),Rn		-	-	-	-	7
mov (Di,Am),Dn		-	-	-	-	2
mov (Di,Am),An		-	-	-	-	2
mov (Ri,Rm),Rn		-	-	-	-	4
mov (abs16),Dn	abs16 is zero-extended.	-	-	-	-	3
mov (abs32),Dn		-	-	-	-	6
mov (abs16),An	abs16 is zero-extended.	-	-	-	-	4
mov (abs32),An		-	-	-	-	6
mov (abs8),Rn	abs8 is zero-extended.	-	-	-	-	4
mov (abs24),Rn	abs24 is zero-extended.	-	-	-	-	6
mov (abs32),Rn		-	-	-	-	7
mov (d8,SP),Dn	d8 is zero-extended.	-	-	-	-	2
mov (d16,SP),Dn	d16 is zero-extended.	-	-	-	-	4
mov (d32,SP),Dn		-	-	-	-	6
mov (d8,SP),An	d8 is zero-extended.	-	-	-	-	2
mov (d16,SP),An	d16 is zero-extended.	-	-	-	-	4

Assembler mnemonic	Note	V	C	N	Z	Size
mov (d32,SP),An		–	–	–	–	6
mov (SP),Rn		–	–	–	–	3
mov (d8,SP),Rn	d8 is zero-extended.	–	–	–	–	4
mov (d24,SP),Rn	d24 is zero-extended.	–	–	–	–	6
mov (d32,SP),Rn		–	–	–	–	7
mov (d8,Am),SP	d8 is sign-extended.	–	–	–	–	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the Mem address is not a multiple of 4, a system exception (address misalignment exception) occurs.

MOV Reg, Mem

Operation	<p>MOV Reg1, (Reg2) Reg1 -> mem32(Reg2)</p> <p>MOV Reg1, (disp, Reg2) Reg1 -> mem32(Reg2+disp)</p> <p>MOV Reg1, (Reg2, Reg3) Reg1 -> mem32(Reg2+Reg3)</p> <p>MOV Reg1, (abs) Reg1 -> mem32(abs)</p> <p>MOV Reg1, (disp, SP) Reg1 -> mem32(SP+disp)</p> <p>This instruction performs the word-data transfer of the content of Reg to Mem.</p> <p>MOV SP, (disp+Reg1) SP -> mem32(Reg1+disp)</p> <p>This instruction performs the word-data transfer of the content of the stack pointer (SP) to Mem.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Dm,(An)		-	-	-	-	1
mov Am,(An)		-	-	-	-	2
mov Rm,(Rn)		-	-	-	-	3
mov Dm,(d8,An)	d8 is sign-extended.	-	-	-	-	3
mov Dm,(d16,An)	d16 is sign-extended.	-	-	-	-	4
mov Dm,(d32,An)		-	-	-	-	6
mov Am,(d8,An)	d8 is sign-extended.	-	-	-	-	3
mov Am,(d16,An)	d16 is sign-extended.	-	-	-	-	4
mov Am,(d32,An)		-	-	-	-	6
mov Rm,(d8,Rn)	d8 is sign-extended.	-	-	-	-	4
mov Rm,(d24,Rn)	d24 is sign-extended.	-	-	-	-	6
mov Rm,(d32,Rn)		-	-	-	-	7
mov Dm,(Di,An)		-	-	-	-	2
mov Am,(Di,An)		-	-	-	-	2
mov Rm,(Ri,Rn)		-	-	-	-	4
mov Dm,(abs16)	abs16 is zero-extended.	-	-	-	-	3
mov Dm,(abs32)		-	-	-	-	6
mov Am,(abs16)	abs16 is zero-extended.	-	-	-	-	4
mov Am,(abs32)		-	-	-	-	6
mov Rm,(abs8)	abs8 is zero-extended.	-	-	-	-	4
mov Rm,(abs24)	abs24 is zero-extended.	-	-	-	-	6
mov Rm,(abs32)		-	-	-	-	7
mov Dm,(d8,SP)	d8 is zero-extended.	-	-	-	-	2
mov Dm,(d16,SP)	d16 is zero-extended.	-	-	-	-	4
mov Dm,(d32,SP)		-	-	-	-	6
mov Am,(d8,SP)	d8 is zero-extended.	-	-	-	-	2
mov Am,(d16,SP)	d16 is zero-extended.	-	-	-	-	4

Assembler mnemonic	Note	V	C	N	Z	Size
mov Am,(d32,SP)		–	–	–	–	6
mov Rm,(SP)		–	–	–	–	3
mov Rm,(d8,SP)	d8 is zero-extended.	–	–	–	–	4
mov Rm,(d24,SP)	d24 is zero-extended.	–	–	–	–	6
mov Rm,(d32,SP)		–	–	–	–	7
mov SP,(d8,An)	d8 is sign-extended.	–	–	–	–	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the Mem address is not a multiple of 4, a system exception (address misalignment exception) occurs.

MOV (Rm+,imm), Rn

Operation	mem32(Rm) -> Rn Rm + imm -> Rn This instruction loads the data specified by Rm from Mem, and stores them in Rn. Moreover, it adds Rm to imm, and stores the result in Rm.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov (Rm+, imm8), Rn	imm8 is sign-extended.	-	-	-	-	4
mov (Rm+, imm24), Rn	imm24 is sign-extended.	-	-	-	-	6
mov (Rm+, imm32), Rn		-	-	-	-	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the Mem address is not a multiple of 4, a system exception (address misalignment exception) occurs.



When Rm=Rn is specified, the operations are not guaranteed.

MOV (Rm+), Rn

Operation	mem32(Rm) -> Rn Rm + 4 -> Rm This instruction loads the data specified by Rm from Mem, and stores them in Rn. Moreover, it adds Rm to 4, and stores the result in Rm.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov (Rm+), Rn		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the Mem address is not a multiple of 4, a system exception (address misalignment exception) occurs.



When Rm=Rn is specified, the operations are not guaranteed.

MOV Rm, (Rn+,imm)						
Operation	Rm -> mem32(Rn) Rn + imm -> Rn This instruction stores the contents of Rm in the memory specified by Rn. Moreover, it adds Rn to imm, and stores the result in Rn.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Rm, (Rn+, imm8)	imm8 is sign-extended.	-	-	-	-	4
mov Rm, (Rn+, imm24)	imm24 is sign-extended.	-	-	-	-	6
mov Rm, (Rn+, imm32)		-	-	-	-	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the Mem address is not a multiple of 4, a system exception (address misalignment exception) occurs.

MOV Rm, (Rn+)						
Operation	Rm -> mem32(Rn) Rn + 4 -> Rn This instruction stores the contents of Rn in the memory specified by Rn. Moreover, it adds Rn to 4, and stores the result in Rn.					
Assembler mnemonic	Note	V	C	N	Z	Size
mov Rm, (Rn+)		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the Mem address is not a multiple of 4, a system exception (address misalignment exception) occurs.

MOVU

Unsigned word transfer

MOVU imm, Reg						
Operation	imm -> Reg This instruction transfers imm to Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
movu imm8, Rn	imm8 is zero-extended.	–	–	–	–	4
movu imm24, Rn	imm24 is zero-extended.	–	–	–	–	6
movu imm32, Rn		–	–	–	–	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOVHU

Unsigned half-word transfer

MOVHU Mem, Reg						
Operation	<p>MOVHU (Reg1), Reg2 mem16(Reg1) -> Reg2[15:0] 0x0000 -> Reg2[31:16]</p> <p>MOVHU (disp, Reg1), Reg2 mem16(Reg1+disp) -> Reg2[15:0] 0x0000 -> Reg2[31:16]</p> <p>MOVHU (Reg1, Reg2), Reg3 mem16(Reg1+Reg2) -> Reg3[15:0] 0x0000 -> Reg3[31:16]</p> <p>MOVHU (abs), Reg1 mem16(abs) -> Reg1[15:0] 0x0000 -> Reg1[31:16]</p> <p>MOVHU (disp, SP), Reg1 mem16(SP+disp) -> Reg1[15:0] 0x0000 -> Reg1[31:16]</p> <p>]</p> <p>This instruction performs the half-word transfer of the content of Mem to Reg. (16 bits -> 32 bits: Zero-extension)</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movhu (Am),Dn		-	-	-	-	2
movhu (Rm),Rn		-	-	-	-	3
movhu (d8,Am),Dn	d8 is sign-extended.	-	-	-	-	3
movhu (d16,Am),Dn	d16 is sign-extended.	-	-	-	-	4
movhu (d32,Am),Dn		-	-	-	-	6
movhu (d8,Rm),Rn	d8 is sign-extended.	-	-	-	-	4
movhu (d24,Rm),Rn	d24 is sign-extended.	-	-	-	-	6
movhu (d32,Rm),Rn		-	-	-	-	7
movhu (Di,Am),Dn		-	-	-	-	2
movhu (Ri,Rm),Rn		-	-	-	-	4
movhu (abs16),Dn	abs16 is zero-extended.	-	-	-	-	3
movhu (abs32),Dn		-	-	-	-	6
movhu (abs8),Rn	abs8 is zero-extended.	-	-	-	-	4
movhu (abs24),Rn	abs24 is zero-extended.	-	-	-	-	6
movhu (abs32),Rn		-	-	-	-	7
movhu (d8,SP),Dn	d8 is zero-extended.	-	-	-	-	3
movhu (d16,SP),Dn	d16 is zero-extended.	-	-	-	-	4
movhu (d32,SP),Dn		-	-	-	-	6
movhu (SP),Rn		-	-	-	-	3
movhu (d8,SP),Rn	d8 is zero-extended.	-	-	-	-	4
movhu (d24,SP),Rn	d24 is zero-extended.	-	-	-	-	6
movhu (d32,SP),Rn		-	-	-	-	7



Flag change
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.



When the Mem address is not a multiple of 2, a system exception (address misalignment exception) occurs.

MOVHU Reg, Mem

Operation	<p>MOVHU Reg1 (Reg2) Reg1[15:0] -> mem16(Reg2)</p> <p>MOVHU Reg1, (disp, Reg2) Reg1[15:0] -> mem16(Reg2+disp)</p> <p>MOVHU Reg1, (Reg2, Reg3) Reg1[15:0] -> mem16(Reg2+Reg3)</p> <p>MOVHU Reg1, (abs) Reg1[15:0] -> mem16(abs)</p> <p>MOVHU Reg1, (disp, SP) Reg1[15:0] -> mem16(SP+disp)</p> <p>This instruction performs the half-word transfer of the content of Reg to Mem. The upper 16 bits of Reg are ignored.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movhu Dm,(An)		-	-	-	-	2
movhu Rm,(Rn)		-	-	-	-	3
movhu Dm,(d8,An)	d8 is sign-extended.	-	-	-	-	3
movhu Dm,(d16,An)	d16 is sign-extended.	-	-	-	-	4
movhu Dm,(d32,An)		-	-	-	-	6
movhu Rm,(d8,Rn)	d8 is sign-extended.	-	-	-	-	4
movhu Rm,(d24,Rn)	d24 is sign-extended.	-	-	-	-	6
movhu Rm,(d32,Rn)		-	-	-	-	7
movhu Dm,(Di,An)		-	-	-	-	2
movhu Rm,(Ri,Rn)		-	-	-	-	4
movhu Dm,(abs16)	abs16 is zero-extended.	-	-	-	-	3
movhu Dm,(abs32)		-	-	-	-	6
movhu Rm,(abs8)	abs8 is zero-extended.	-	-	-	-	4
movhu Rm,(abs24)	abs24 is zero-extended.	-	-	-	-	6
movhu Rm,(abs32)		-	-	-	-	7
movhu Dm,(d8,SP)	d8 is zero-extended.	-	-	-	-	3
movhu Dm,(d16,SP)	d16 is zero-extended.	-	-	-	-	4
movhu Dm,(d32,SP)		-	-	-	-	6
movhu Rm,(SP)		-	-	-	-	3
movhu Rm,(d8,SP)	d8 is zero-extended.	-	-	-	-	4
movhu Rm,(d24,SP)	d24 is zero-extended.	-	-	-	-	6
movhu Rm,(d32,SP)		-	-	-	-	7

Flag change
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.



When the Mem address is not a multiple of 2, a system exception (address misalignment exception) occurs.

MOVHU (Rm+, imm), Rn

Operation	<p>mem16(Rm) -> Rn[15:0] 0x0000 -> Rn[31:16] Rm + imm -> Rm</p> <p>This instruction loads the 16-bit data from Mem specified by Rm, and transfers them in Rn. Moreover, it adds Rm to imm, and stores the result in Rm.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movhu (Rm+, imm8), Rn	imm8 is sign-extended.	-	-	-	-	4
movhu (Rm+, imm24), Rn	imm24 is sign-extended.	-	-	-	-	6
movhu (Rm+, imm32), Rn		-	-	-	-	7
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						



When the Mem address is not a multiple of 2, a system exception (address misalignment exception) occurs.

MOVHU (Rm+), Rn

Operation	<p>mem16(Rm) -> Rn[15:0] 0x0000 -> Rn[31:16] Rm + 2 -> Rm</p> <p>This instruction loads the data specified by Rm from Mem, and stores them in Rn. Moreover, it adds Rm to 4, and stores the result in Rm.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movhu (Rm+), Rn		-	-	-	-	3
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						



When the Mem address is not a multiple of 2, a system exception (address misalignment exception) occurs.

MOVHU Rm, (Rn+, imm)

Operation	<p>Rm[15:0] -> mem16(Rn) Rn + imm -> Rn</p> <p>This instruction stores the lower 16 bits of the Rm in the Mem specified by Rn. Moreover, it adds Rn to imm, and stores the result in Rn.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movhu Rm(Rn+, imm8)	imm8 is sign-extended.	–	–	–	–	4
movhu Rm(Rn+, imm24)	imm24 is sign-extended.	–	–	–	–	6
movhu Rm(Rn+, imm32)		–	–	–	–	7
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						



When the Mem address is not a multiple of 2, a system exception (address misalignment exception) occurs.

MOVHU (Rm+), Rn

Operation	<p>Rm[15:0] -> mem16(Rm) Rn + 2 -> Rn</p> <p>This instruction stores the lower 16 bits of Rn in the Mem specified by Rn. Moreover, it adds Rm to 2, and stores the result in Rn.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movhu Rm, (Rn+)		–	–	–	–	3
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						



When the Mem address is not a multiple of 2, a system exception (address misalignment exception) occurs.

MOVBU

Unsigned byte transfer

MOVBU Mem, Reg						
Operation	<p>MOVBU (Reg1), Reg2 mem8(Reg1) -> Reg2[7:0] 0x000000 -> Reg2[31:8]</p> <p>MOVBU (disp, Reg1), Reg2 mem8(Reg1+disp) -> Reg2[7:0] 0x000000 -> Reg2[31:8]</p> <p>MOVBU (Reg1, Reg2), Reg3 mem8(Reg1+Reg2) -> Reg3[7:0] 0x000000 -> Reg3[31:8]</p> <p>MOVBU (abs), Reg1 mem8(abs) -> Reg1[7:0] 0x000000 -> Reg1[31:8]</p> <p>MOVBU (disp, SP), Reg1 mem8(SP+disp) -> Reg1[7:0] 0x000000 -> Reg1[31:8]</p> <p>This instruction performs the byte-data transfer of the content of Mem to Reg. (8 bits -> 32 bits: Zero-extension)</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movbu (Am),Dn		-	-	-	-	2
movbu (Rm),Rn		-	-	-	-	3
movbu (d8,Am),Dn	d8 is sign-extended.	-	-	-	-	3
movbu (d16,Am),Dn	d16 is sign-extended.	-	-	-	-	4
movbu (d32,Am),Dn		-	-	-	-	6
movbu (d8,Rm),Rn	d8 is sign-extended.	-	-	-	-	4
movbu (d24,Rm),Rn	d24 is sign-extended.	-	-	-	-	6
movbu (d32,Rm),Rn		-	-	-	-	7
movbu (Di,Am),Dn		-	-	-	-	2
movbu (Ri,Rm),Rn		-	-	-	-	4
movbu (abs16),Dn	abs16 is zero-extended.	-	-	-	-	3
movbu (abs32),Dn		-	-	-	-	6
movbu (abs8),Rn	abs8 is zero-extended.	-	-	-	-	4
movbu (abs24),Rn	abs24 is zero-extended.	-	-	-	-	6
movbu (abs32),Rn		-	-	-	-	7
movbu (d8,SP),Dn	d8 is zero-extended.	-	-	-	-	3
movbu (d16,SP),Dn	d16 is zero-extended.	-	-	-	-	4
movbu (d32,SP),Dn		-	-	-	-	6
movbu (SP),Rn		-	-	-	-	3
movbu (d8,SP),Rn	d8 is zero-extended.	-	-	-	-	4
movbu (d24,SP),Rn	d24 is zero-extended.	-	-	-	-	6
movbu (d32,SP),Rn		-	-	-	-	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOVBW Reg, Mem

Operation	MOVBW Reg1 (Reg2) Reg1[7:0] -> mem8(Reg2) MOVBW Reg1, (disp, Reg2) Reg1[7:0] -> mem8(Reg2+disp) MOVBW Reg1, (Reg2, Reg3) Reg1[7:0] -> mem8(Reg2+Reg3) MOVBW Reg1, (abs) Reg1[7:0] -> mem8(abs) MOVBW Reg1, (disp, SP) Reg1[7:0] -> mem8(SP+disp) This instruction performs the byte-data transfer of the content of Reg to Mem. (32 bits -> 8 bits: The upper bits are rounded down.)					
Assembler mnemonic	Note	V	C	N	Z	Size
movbu Dm,(An)		-	-	-	-	2
movbu Rm,(Rn)		-	-	-	-	3
movbu Dm,(d8,An)	d8 is sign-extended.	-	-	-	-	3
movbu Dm,(d16,An)	d16 is sign-extended.	-	-	-	-	4
movbu Dm,(d32,An)		-	-	-	-	6
movbu Rm,(d8,Rn)	d8 is sign-extended.	-	-	-	-	4
movbu Rm,(d24,Rn)	d24 is sign-extended.	-	-	-	-	6
movbu Rm,(d32,Rn)		-	-	-	-	7
movbu Dm,(Di,An)		-	-	-	-	2
movbu Rm,(Ri,Rn)		-	-	-	-	4
movbu Dm,(abs16)	abs16 is zero-extended.	-	-	-	-	3
movbu Dm,(abs32)		-	-	-	-	6
movbu Rm,(abs8)	abs8 is zero-extended.	-	-	-	-	4
movbu Rm,(abs24)	abs24 is zero-extended.	-	-	-	-	6
movbu Rm,(abs32)		-	-	-	-	7
movbu Dm,(d8,SP)	d8 is zero-extended.	-	-	-	-	3
movbu Dm,(d16,SP)	d16 is zero-extended.	-	-	-	-	4
movbu Dm,(d32,SP)		-	-	-	-	6
movbu Rm,(SP)		-	-	-	-	3
movbu Rm,(d8,SP)	d8 is zero-extended.	-	-	-	-	4
movbu Rm,(d24,SP)	d24 is zero-extended.	-	-	-	-	6
movbu Rm,(d32,SP)		-	-	-	-	7
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

MOVM

Transfer between multiple registers and memory

MOVM (SP), regs (SP relative)						
Operation	<p>When all registers are specified, regs=[ALL] mem32(SP+92) -> E2, mem32(SP+88) -> E3, mem32(SP+84) -> E4 mem32(SP+80) -> E5, mem32(SP+76) -> E6, mem32(SP+72) -> E7 mem32(SP+68) -> E0, mem32(SP+64) -> E1, mem32(SP+60) -> MDRQ mem32(SP+56) -> MCRH, mem32(SP+52) -> MCRL, mem32(SP+48) -> MCVF mem32(SP+44) -> D2, mem32(SP+40) -> D3, mem32(SP+36) -> A2 mem32(SP+32) -> A3, mem32(SP+28) -> D0, mem32(SP+24) -> D1 mem32(SP+20) -> A0, mem32(SP+16) -> A1, mem32(SP+12) -> MDR mem32(SP+8) -> LIR, mem32(SP+4) -> LAR, SP+96 -> SP</p> <p>This instruction restores the multiple registers saved in the memory indicated by SP. The multiple registers to transfer data are specified by regs, and D2, D3, A2, A3 and other registers are specified as followings. OTHER : D0,D1,A0,A1,MDR,LIR,LAR EXREG0 : E2,E3 EXREG1 : E4,E5,E6,E7 EXOTHER : E0,E1,MDRQ,MCRH,MCRL,MCVF</p> <p>When all of the above-mentioned registers are specified, ALL is used. ALL : D2,D3,A2,A3,OTHER,EXREG0,EXREG1,EXOTHER</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movm (SP), [reg1, reg2,,,regn]		-	-	-	-	2
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						



This instruction can be executed at any level, and refers to SP at the present. That is, it refers to USP when the present level is the user level, SSP when it is the supervisor one, and MSP when it is the monitor one.

Bit allocation of the multiple registers (regs) is shown below. When you specify all of these, use ALL.

7	6	5	4	3	2	1	0
D2	D3	A2	A3	OTHER	EXREG0	EXREG1	EXOTHER

Data is transferred from the higher addresses in the order of E2, E3, E4, E5, E6, E7, E0,

E1, MDRQ, MCRH, MCRL, MCVF, D2, D3, A2, A3, D0, D1, A0, A1, MDR, LIR and LAR.

Note that when “OTHER” is specified, a dummy area (4 bytes) is allocated (transfer operation is not actually performed) at the end in order to simplify calculation of the transfer area (4 bytes x 8).

When “OTHER” is not specified, this dummy area is not allocated.

When reg=0 is specified (when “movm (SP),[]” is executed), nothing is transferred.

When the Mem address indicated by the stack pointer (SP) is not a multiple of 4, system exception (Address misalignment exception) occurs.

The memory status when all registers are specified is shown below.

	Low address	Restore Order	Offset from SP
SP before execution ->	Dummy area	(-)	0
	LAR	(23)	+4
	LIR	(22)	+8
	MDR	(21)	+12
	A1	(20)	+16
	A0	(19)	+20
	D1	(18)	+24
	D0	(17)	+28
	A3	(16)	+32
	A2	(15)	+36
	D3	(14)	+40
	D2	(13)	+44
	MCVF	(12)	+48
	MCRL	(11)	+52
	MCRH	(10)	+56
	MDRQ	(9)	+60
	E1	(8)	+64
	E0	(7)	+68
	E7	(6)	+72
	E6	(5)	+76
	E5	(4)	+80
	E4	(3)	+84
	E3	(2)	+88
	E2	(1)	+92
			+96
	High address		
SP after execution ->			

MOVM regs, (SP) (SP relative)						
Operation	<p>When all registers are specified, regs=[ALL] E2 -> mem32(SP-4), E3 -> mem32(SP-8), E4 -> mem32(SP-12), E5 -> mem32(SP-16), E6 -> mem32(SP-20), E7 -> mem32(SP-24), E0 -> mem32(SP-28), E1-> mem32(SP-32), MDRQ -> mem32(SP-36), MCRH -> mem32(SP-40), MCRL-> mem32(SP-44),MCVF -> mem32(SP-48), D2 -> mem32(SP-52), D3 -> mem32(SP-56), A2 -> mem32(SP-60), A3 -> mem32(SP-64),D0 -> mem32(SP-68), D1 -> mem32(SP-72), A0 -> mem32(SP-76),A1 -> mem32(SP-80),MDR -> mem32(SP-84), LIR-> mem32(SP-88),LAR -> mem32(SP-92), SP-96 -> SP</p> <p>This instruction restores the multiple registers saved in the memory indicated by SP. The multiple registers to transfer data are specified by regs, and D2, D3, A2, A3 and other registers are specified as followings. OTHER : D0,D1,A0,A1,MDR,LIR,LAR EXREG0 : E2,E3 EXREG1 : E4,E5,E6,E7 EXOTHER : E0,E1,MDRQ,MCRH,MCRL,MCVF</p> <p>When all of the above-mentioned registers are specified, ALL is used. ALL : D2,D3,A2,A3,OTHER,EXREG0,EXREG1,EXOTHER</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movm [reg1, reg2,,,regn], (SP)		-	-	-	-	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction can be executed at any level, and refers to SP at the present. That is, it refers to USP when the present level is the user level, SSP when it is the supervisor one, and MSP when it is the monitor one.

Bit allocation of the multiple registers (regs) is shown below. When you specify all of these, use ALL.

7	6	5	4	3	2	1	0
D2	D3	A2	A3	OTHER	EXREG0	EXREG1	EXOTHER

Data is transferred from the higher addresses in the order of E2, E3, E4, E5, E6, E7, E0, E1, MDRQ, MCRH, MCRL, MCVF, D2, D3, A2, A3, D0, D1, A0, A1, MDR, LIR and LAR.

Note that when “OTHER” is specified, a dummy area (4 bytes) is allocated (transfer operation is not actually performed) at the end in order to simplify calculation of the transfer area (4 bytes x 8). When “OTHER” is not specified, this dummy area is not allocated.

When reg=0 is specified (when “movm ,[], (SP)” is executed), nothing is transferred.

When the Mem address indicated by the stack pointer (SP) is not a multiple of 4, system exception (Address misalignment exception) occurs.

The memory status when all registers are specified is shown below.

	Low address	Restore Order	Offset from SP
SP after execution ->	Dummy area	(-)	-96
	LAR	(23)	-94
	LIR	(22)	-88
	MDR	(21)	-84
	A1	(20)	-80
	A0	(19)	-76
	D1	(18)	-72
	D0	(17)	-68
	A3	(16)	-64
	A2	(15)	-60
	D3	(14)	-56
	D2	(13)	-52
	MCVF	(12)	-48
	MCRL	(11)	-44
	MCRH	(10)	-40
	MDRQ	(9)	-36
	E1	(8)	-32
	E0	(7)	-28
	E7	(6)	-24
	E6	(5)	-20
	E5	(4)	-16
	E4	(3)	-12
	E3	(2)	-8
	E2	(1)	-4
SP before execution ->			-0
	High address		

MOVM (USP), regs (USP relative<Privileged instruction>)

Operation	<p>When all registers are specified, regs=[ALL] mem32(USP+92) -> E2, mem32(USP+88) -> E3, mem32(USP+84) -> E4 mem32(USP+80) -> E5, mem32(USP+76) -> E6, mem32(USP+72) -> E7 mem32(USP+68) -> E0, mem32(USP+64) -> E1, mem32(USP+60) -> MDRQ mem32(USP+56) -> MCRH, mem32(USP+52) -> MCRL, mem32(USP+48) -> MCVF mem32(USP+44) -> D2, mem32(USP+40) -> D3, mem32(USP+36) -> A2 mem32(USP+32) -> A3, mem32(USP+28) -> D0, mem32(USP+24) -> D1 mem32(USP+20) -> A0, mem32(USP+16) -> A1, mem32(USP+12) -> MDR mem32(USP+8) -> LIR, mem32(USP+4) -> LAR, USP+96 -> USP</p> <p>This instruction restores the multiple registers saved in the memory indicated by USP. The multiple registers to transfer data are specified by regs, and D2, D3, A2, A3 and other registers are specified as followings. OTHER : D0,D1,A0,A1,MDR,LIR,LAR EXREG0 : E2,E3 EXREG1 : E4,E5,E6,E7 EXOTHER : E0,E1,MDRQ,MCRH,MCRL,MCVF</p> <p>When all of the above-mentioned registers are specified, ALL is used. ALL : D2,D3,A2,A3,OTHER,EXREG0,EXREG1,EXOTHER</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movm (USP),[reg1, reg2,,,regn]		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



This instruction can be executed at the monitor and supervisor levels, and refers to USP in any level. When an instruction is to be carried out in the user level, a system exception (privileged instruction execution exception) occurs.

Bit allocation of the multiple registers (regs) is shown below. When you specify all of these, use ALL.

7	6	5	4	3	2	1	0
D2	D3	A2	A3	OTHER	EXREG0	EXREG1	EXOTHER

Data is transferred from the higher addresses in the order of E2, E3, E4, E5, E6, E7, E0, E1, MDRQ, MCRH, MCRL, MCVF, D2, D3, A2, A3, D0, D1, A0, A1, MDR, LIR and LAR.

Note that when “OTHER” is specified, a dummy area (4 bytes) is allocated (transfer operation is not actually performed) at the end in order to simplify calculation of the transfer area (4 bytes x 8). When “OTHER” is not specified, this dummy area is not allocated.

When reg=0 is specified (when “movm (USP),[]” is executed), nothing is transferred.

When the Mem address indicated by the stack pointer (USP) is not a multiple of 4, system exception (Address misalignment exception) occurs.

Although MOVm using a stack pointer (SP) is a 2-byte instruction, this instruction is a 3-byte one.

The memory status when all registers are specified is shown below.

	Low address	Restore Order	Offset from USP
USP before execution ->	Dummy area	(-)	0
	LAR	(23)	+4
	LIR	(22)	+8
	MDR	(21)	+12
	A1	(20)	+16
	A0	(19)	+20
	D1	(18)	+24
	D0	(17)	+28
	A3	(16)	+32
	A2	(15)	+36
	D3	(14)	+40
	D2	(13)	+44
	MCVF	(12)	+48
	MCRL	(11)	+52
	MCRH	(10)	+56
	MDRQ	(9)	+60
	E1	(8)	+64
	E0	(7)	+68
	E7	(6)	+72
	E6	(5)	+76
	E5	(4)	+80
	E4	(3)	+84
	E3	(2)	+88
	E2	(1)	+92
			+96
	High address		
USP after execution ->			

MOVM regs,(USP)(USP relative<Privileged instruction>)						
Operation	<p>When all registers are specified, regs=[ALL] E2 -> mem32(USP-4), E3 -> mem32(USP-8), E4 -> mem32(USP-12), E5 -> mem32(USP-16), E6 -> mem32(USP-20), E7 -> mem32(USP-24), E0 -> mem32(USP-28), E1-> mem32(USP-32), MDRQ -> mem32(USP-36), MCRH -> mem32(USP-40), MCRL-> mem32(USP-44),MCVF -> mem32(USP-48), D2 -> mem32(USP-52), D3 -> mem32(USP-56), A2 -> mem32(USP-60), A3 -> mem32(USP-64),D0 -> mem32(USP-68), D1 -> mem32(USP-72), A0 -> mem32(USP-76),A1 -> mem32(USP-80),MDR -> mem32(USP-84), LIR-> mem32(USP-88),LAR -> mem32(USP-92), USP-96 -> USP</p> <p>This instruction saves the multiple registers in the memory indicated by USP. The multiple registers to transfer data are specified by regs, and D2, D3, A2, A3 and other registers are specified as followings. OTHER : D0,D1,A0,A1,MDR,LIR,LAR EXREG0 : E2,E3 EXREG1 : E4,E5,E6,E7 EXOTHER : E0,E1,MDRQ,MCRH,MCRL,MCVF</p> <p>When all of the above-mentioned registers are specified, ALL is used. ALL : D2,D3,A2,A3,OTHER,EXREG0,EXREG1,EXOTHER</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
movm [reg1, reg2,,,regn], (USP)		-	-	-	-	3
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						



This instruction can be executed at the monitor and supervisor levels, and refers to USP in any level. When an instruction is to be carried out in the user level, a system exception (privileged instruction execution exception) occurs.

Bit allocation of the multiple registers (regs) is shown below. When you specify all of these, use ALL.

7	6	5	4	3	2	1	0
D2	D3	A2	A3	OTHER	EXREG0	EXREG1	EXOTHER

Data is transferred from the higher addresses in the order of E2, E3, E4, E5, E6, E7, E0, E1, MDRQ, MCRH, MCRL, MCVF, D2, D3, A2, A3, D0, D1, A0, A1, MDR, LIR and LAR.

Note that when “OTHER” is specified, a dummy area (4 bytes) is allocated (transfer operation is not actually performed) at the end in order to simplify calculation of the transfer area (4 bytes x 8).

When “OTHER” is not specified, this dummy area is not allocated.

When reg=0 is specified (when “movm [], (USP)” is executed), nothing is transferred.

When the Mem address indicated by the stack pointer (USP) is not a multiple of 4, system exception (Address misalignment exception) occurs.

Although MOVm using a stack pointer (SP) is a 2-byte instruction, this instruction is a 3-byte one.

The memory status when all registers are specified is shown below.

	Low address	Restore Order	Offset from USP
SP after execution ->	Dummy area	(-)	-96
	LAR	(23)	-94
	LIR	(22)	-88
	MDR	(21)	-84
	A1	(20)	-80
	A0	(19)	-76
	D1	(18)	-72
	D0	(17)	-68
	A3	(16)	-64
	A2	(15)	-60
	D3	(14)	-56
	D2	(13)	-52
	MCVF	(12)	-48
	MCRL	(11)	-44
	MCRH	(10)	-40
	MDRQ	(9)	-36
	E1	(8)	-32
	E0	(7)	-28
	E7	(6)	-24
	E6	(5)	-20
	E5	(4)	-16
	E4	(3)	-12
	E3	(2)	-8
	E2	(1)	-4
			-0
	High address		
SP before execution ->			

EXT

64-bit sign-extension of word data

EXT Reg						
Operation	<p>When Reg[31]=0, 0x00000000 -> MDR When Reg[31]=1, 0xFFFFFFFF -> MDR</p> <p>This instruction sign-extends Reg to 64 bits, and transfers the upper 32 bits to MDR. The contents of Reg are not changed.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
ext Dn		–	–	–	–	2
ext Rn		–	–	–	–	3
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						

EXTH

32-bit sign-extension of half-word data

EXTH Reg						
Operation	When Reg[15]=0, Reg & 0x0000FFFF -> Reg When Reg[15]=1, Reg 0xFFFF0000 -> Reg This instruction sign-extends the lower 16 bits of Reg to 32 bits, and stores them in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
exth Dn		-	-	-	-	1
exth Rn		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the instruction has an operand register consisting of other than data register (Dn) for 1 operand operation, it is implemented by instruction swapping and the assembler constitutes the following instruction. `exth Rn, Rn`

EXTH Reg1, Reg2						
Operation	When Reg1[15]=0, Reg1 & 0x0000FFFF -> Reg2 When Reg1[15]=1, Reg1 0xFFFF0000 -> Reg2 This instruction sign-extends the lower 16 bits of Reg1 to 32 bits, and stores them into Reg2. The contents of Reg1 are not changed.					
Assembler mnemonic	Note	V	C	N	Z	Size
exth Rm, Rn		-	-	-	-	-
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

EXTHU

32-bit zero-extension of half-word data

EXTHU Reg						
Operation	When Reg[15]=0, Reg & 0x0000FFFF -> Reg When Reg[15]=1, Reg 0xFFFF0000 -> Reg This instruction sign-extends the lower 16 bits of Reg to 32 bits, and stores them in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
exthu Dn		-	-	-	-	1
exthu Rn		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the instruction has an operand register consisting of other than data register (Dn) for 1 operand operation, it is implemented by instruction swapping and the assembler constitutes the following instruction. `exthu Rn, Rn`

EXTHU Reg1, Reg2						
Operation	When Reg1[15]=0, Reg1 & 0x0000FFFF -> Reg2 When Reg1[15]=1, Reg1 0xFFFF0000 -> Reg2 This instruction zero-extends the lower 16 bits of Reg1 to 32 bits, and stores them into Reg2. The contents of Reg1 are not changed.					
Assembler mnemonic	Note	V	C	N	Z	Size
exthu Rm, Rn		-	-	-	-	-
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

EXTB

32-bit sign-extension of byte data

EXTB Reg						
Operation	When Rn[7]=0, Reg & 0x000000FF -> Reg When Rn[7]=1, Reg 0xFFFFFFF0 -> Reg This instruction sign-extends the lower 8 bits of Reg to 32 bits, and stores them in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
extb Dn		-	-	-	-	1
extb Rn		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the instruction has an operand register consisting of other than data register (Dn) for 1 operand operation, it is implemented by instruction swapping and the assembler constitutes the following instruction. `extb Rn, Rn`

EXTB Reg1, Reg2						
Operation	When Reg1[7]=0, Reg1 & 0x000000FF -> Reg2 When Reg1[7]=1, Reg1 0xFFFFFFF0 -> Reg2 This instruction zero-extends the lower 8 bits of Reg1 to 32 bits, and stores them into Reg2. The contents of Reg1 are not changed.					
Assembler mnemonic	Note	V	C	N	Z	Size
extb Rm, Rn		-	-	-	-	-
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

EXTBU

32-bit zero-extension of byte data

EXTBU Reg						
Operation	Reg & 0x000000FF -> Rn This instruction zero-extends the lower 8 bits of Reg to 32 bits, and stores them in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
extbu Dn		-	-	-	-	1
extbu Rn		-	-	-	-	3
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



When the instruction has an operand register consisting of other than data register (D) for 1 operand operation, it is implemented by instruction swapping and the assembler constitutes the following instruction. `extbu Rn, Rn`

EXTBU Reg1, Reg2						
Operation	Reg1 & 0x000000FF -> Reg2 This instruction zero-extends the lower 8 bits of Reg1 to 32 bits, and stores them into Reg2. The contents of Reg1 are not changed.					
Assembler mnemonic	Note	V	C	N	Z	Size
extbu Rm, Rn		-	-	-	-	-
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						




CLR

Data clear

EXTBU Reg						
Operation	<p>Reg & 0x000000FF -> Rn</p> <p>This instruction clears the contents of Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
clr Dn		-	-	-	1	1
clr Rn		-	-	-	1	3
Flag change						
<p>VF: Always 0.</p> <p>CF: Always 0.</p> <p>NF: Always 0.</p> <p>ZF: Always 1.</p>						

DCPF

Data cache pre-fetch

DCPF Mem										
Operation	DCPF (Reg) mem_cline(Reg) -> [Data Cache] DCPF (Ri, Rm) mem_cline(Ri + Rm) -> [Data Cache] DCPF (disp, Reg) mem_cline(Reg + disp) -> [Data Cache] DCPF (SP) mem_cline(SP) -> [Data Cache]									
Assembler mnemonic		Note				V	C	N	Z	Size
dcpf (Rm)						-	-	-	-	3
dcpf (Ri, Rm)						-	-	-	-	4
dcpf (d8, Rm)		d8 is sign-extended.				-	-	-	-	4
dcpf (d24, Rm)		d24 is sign-extended.				-	-	-	-	6
dcpf (d32, Rm)						-	-	-	-	7
dcpf (SP)						-	-	-	-	3
Flag change										
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.										



The contents of the register are not updated through executing DCPF instruction.



DCPF instruction do not cause illegal memory access exception, address alignment exception and MMU exception.

ADD

Addition

ADD Reg1, Reg2						
Operation	Reg1 + Reg2 -> Reg2 This instruction adds the contents of Reg1 and Reg2 to each other, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
add Dm, Dn		Δ	Δ	Δ	Δ	1
add Dm, An		Δ	Δ	Δ	Δ	2
add Am, Dn		Δ	Δ	Δ	Δ	2
add Am, An		Δ	Δ	Δ	Δ	2
add Rm, Rn		Δ	Δ	Δ	Δ	3
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the operation results are 0; 0 in all other cases						

ADD Reg1, Reg2, Reg3						
Operation	Reg1 + Reg2 -> Reg3 This instruction adds the contents of Reg1 and Reg2 to each other, and stores the result in Reg3.					
Assembler mnemonic	Note	V	C	N	Z	Size
add Rm, Rn, Rd		Δ	Δ	Δ	Δ	4
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the operation results are 0; 0 in all other cases						

ADD imm, Reg

Operation	<p>$(\text{sign_ext})\text{imm} + \text{Reg} \rightarrow \text{Reg}$</p> <p>This instruction adds imm and Reg to each other, and stores the result in Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add imm8, Dn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	2
add imm16, Dn	imm16 is sign-extended.	Δ	Δ	Δ	Δ	4
add imm32, Dn		Δ	Δ	Δ	Δ	6
add imm8, An	imm8 is sign-extended.	Δ	Δ	Δ	Δ	2
add imm16, An	imm16 is sign-extended.	Δ	Δ	Δ	Δ	4
add imm32, An		Δ	Δ	Δ	Δ	6
add imm8, Rn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	4
add imm24, Rn	imm24 is sign-extended.	Δ	Δ	Δ	Δ	6
add imm32, Rn		Δ	Δ	Δ	Δ	7
Flag change						
<p>VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases</p> <p>CF: 1 when carry from bit 31 occurs; 0 in all other cases</p> <p>NF: 1 when bit 31 of the operation results is 1; 0 in all other cases</p> <p>ZF: 1 when the operation results are 0; 0 in all other cases</p>						

ADD imm, SP

Operation	<p>$(\text{sign_ext})\text{imm} + \text{SP} \rightarrow \text{SP}$</p> <p>This instruction adds imm and SP to each other, and stores the result in SP.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add imm8, SP	imm8 is sign-extended.	Δ	Δ	Δ	Δ	3
add imm16, SP	imm16 is sign-extended.	Δ	Δ	Δ	Δ	4
add imm32, SP		Δ	Δ	Δ	Δ	6
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

ADDC

Addition with a carry

ADDC Reg1, Reg2						
Operation	Reg1 + Reg2 + EPSW.C -> Reg2 This instruction adds the contents of Reg1 and Reg2 to the C flag, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
addc Dm, Dn		Δ	Δ	Δ	Δ	2
addc Rm, Rn		Δ	Δ	Δ	Δ	3
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the Z flag before the operation is 1 and the operation results are 0; 0 in all other cases.						

ADDC Reg1, Reg2, Reg3						
Operation	Reg1 + Reg2 + EPSW.C -> Reg3 This instruction adds the contents of Reg1 and Reg2 to the C flag, and stores the result in Reg3.					
Assembler mnemonic	Note	V	C	N	Z	Size
addc Rm, Rn, Rd		Δ	Δ	Δ	Δ	4
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the Z flag before the operation is 1 and the operation results are 0; 0 in all other cases.						

ADDC imm, Reg						
Operation	<p>$(\text{sign_ext})\text{imm} + \text{Reg} + \text{EPSW.C} \rightarrow \text{Reg}$</p> <p>This instruction adds imm and Reg to the C flag, and stores the result in Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
addc imm8, Rn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	2
addc imm24, Rn	imm24 is sign-extended.	Δ	Δ	Δ	Δ	3
addc imm32, Rn		Δ	Δ	Δ	Δ	1
Flag change						
<p>VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases</p> <p>CF: 1 when carry from bit 31 occurs; 0 in all other cases</p> <p>NF: 1 when bit 31 of the operation results is 1; 0 in all other cases</p> <p>ZF: 1 when the Z flag before the operation is 1 and the operation results are 0; 0 in all other cases.</p>						

SUB

Subtraction

SUB Reg1, Reg2						
Operation	Reg2 - Reg1 -> Reg2 This instruction subtracts the contents of Reg1 from Reg2 to each other, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
sub Dm, Dn		Δ	Δ	Δ	Δ	2
sub Dm, An		Δ	Δ	Δ	Δ	2
sub Am, Dn		Δ	Δ	Δ	Δ	2
sub Am, An		Δ	Δ	Δ	Δ	2
sub Rm, Rn		Δ	Δ	Δ	Δ	3
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the operation results are 0; 0 in all other cases						

SUB Reg1, Reg2, Reg3						
Operation	Reg2 - Reg1 -> Reg3 This instruction subtracts the contents of Reg1 from Reg2, and stores the result in Reg3.					
Assembler mnemonic	Note	V	C	N	Z	Size
sub Rm, Rn, Rd		Δ	Δ	Δ	Δ	4
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the operation results are 0; 0 in all other cases						

SUB imm, Reg						
Operation	<p>Reg - (sign_ext)imm -> Reg</p> <p>This instruction subtracts the sign-extended imm from Reg, and stores the result in Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
sub imm32, Dn		Δ	Δ	Δ	Δ	6
sub imm32, An		Δ	Δ	Δ	Δ	6
sub imm8, Dn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	4
sub imm24, An	imm24 is sign-extended.	Δ	Δ	Δ	Δ	6
sub imm32, Rn		Δ	Δ	Δ	Δ	7
Flag change						
<p>VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases</p> <p>CF: 1 when carry from bit 31 occurs; 0 in all other cases</p> <p>NF: 1 when bit 31 of the operation results is 1; 0 in all other cases</p> <p>ZF: 1 when the operation results are 0; 0 in all other cases</p>						

SUBC

Subtraction with a carry

SUBC Reg1, Reg2						
Operation	Reg2 - Reg1 - EPSW.C -> Reg2 This instruction subtracts the contents of Reg1 including the C flag from Reg2, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
subc Dm, Dn		Δ	Δ	Δ	Δ	2
subc Rm, Rn		Δ	Δ	Δ	Δ	3
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the Z flag before the operation is 1 and the operation results are 0; 0 in all other cases.						

SUBC Reg1, Reg2, Reg3						
Operation	Reg3 - Reg2 - EPSW.C -> Reg3 This instruction subtracts the contents of Reg2 including the C flag from Reg3, and stores the result in Reg3.					
Assembler mnemonic	Note	V	C	N	Z	Size
subc Rm, Rn, Rd		Δ	Δ	Δ	Δ	4
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the Z flag before the operation is 1 and the operation results are 0; 0 in all other cases.						

SUBC imm, Reg						
Operation	<p>Reg - (sign_ext)imm - EPSW.C -> Reg</p> <p>This instruction subtracts the sign-extended imm and the C flag from Reg, and stores the result in Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
subc imm8, Rn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	4
subc imm24, Rn	imm24 is sign-extended.	Δ	Δ	Δ	Δ	6
subc imm32, Rn		Δ	Δ	Δ	Δ	7
Flag change						
<p>VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases</p> <p>CF: 1 when carry from bit 31 occurs; 0 in all other cases</p> <p>NF: 1 when bit 31 of the operation results is 1; 0 in all other cases</p> <p>ZF: 1 when the Z flag before the operation is 1 and the operation results are 0;0 in all other cases.</p>						

MUL

Signed multiplication

MUL Reg1, Reg2						
Operation	<p>$\text{Reg1} * \text{Reg2} \rightarrow \{\text{MDR}, \text{Reg2}\}$</p> <p>The contents of Reg1 (signed 32-bit integer: multiplicand) and Reg2 (signed 32-bit integer: multiplier) are multiplied, and the upper 32 bits of the result (64 bits) are written into MDR and the lower 32 bits into Reg2.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mul Dm, Dn		?	?	Δ	Δ	2
mul Rm, Rn		?	?	Δ	Δ	3
Flag change						
<p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: 1 when bit 31 of the lower32 bits of the operation result is 1; 0 in all other cases</p> <p>ZF: 1 when the lower 32 bits of the operation result is 0 and the operation results are 0; 0 in all other cases</p>						

MUL Reg1, Reg2, Reg3, Reg4						
Operation	<p>$\text{Reg1} * \text{Reg2} \rightarrow \{\text{Reg3}, \text{Reg4}\}$</p> <p>The contents of Reg1 (signed 32-bit integer: multiplicand) and Reg2 (signed 32-bit integer: multiplier) are multiplied, and the upper 32 bits of the result (64 bits) are written into Reg3 and the lower 32 bits into Reg4.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mul Rm, Rn, Rd1, Rd2	Rd1=Rd2 cannot be specified.	Δ	Δ	Δ	Δ	4
Flag change						
<p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: 1 when the MSB of the operation result (64 bits) is 1; 0 in all other cases</p> <p>ZF: 1 when the operation result (64 bits) is 0; 0 in all other cases</p>						



When Rd1=Rd2 is specified, the operation result is not guaranteed.

MUL imm, Reg						
Operation	<p>$(\text{sign_ext})\text{imm} * \text{Reg} \rightarrow \{ \text{MDR}, \text{Reg} \}$</p> <p>imm (signed 32-bit integer: multiplicand) and the contents of Reg (signed 32-bit integer: multiplier) are multiplied, and the upper 32 bits of the result (64 bits) are written into MDR and the lower 32 bits into Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mul imm8, Rn	imm8 is sign-extended.	?	?	Δ	Δ	4
mul imm24, Rn	imm24 is sign-extended.	?	?	Δ	Δ	6
mul imm32, Rn		?	?	Δ	Δ	7
Flag change						
<p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: 1 when bit 31 of the lower32 bits of the operation result is 1; 0 in all other cases</p> <p>ZF: 1 when the lower 32 bits of the operation result is 0 ; 0 in all other cases</p>						

MULU

Unsigned multiplication

MULU Reg1, Reg2						
Operation	<p>$\text{Reg1} * \text{Reg2} \rightarrow \{\text{MDR}, \text{Reg2}\}$</p> <p>The contents of Reg1 (unsigned 32-bit integer: multiplicand) and Reg2 (unsigned 32-bit integer: multiplier) are multiplied, and the upper 32 bits of the result (64 bits) are written into MDR and the lower 32 bits into Reg2.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mulu Dm, Dn		?	?	Δ	Δ	2
mulu Rm, Rn		?	?	Δ	Δ	3
Flag change						
<p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: 1 when bit 31 of the lower32 bits of the operation result is 1; 0 in all other cases</p> <p>ZF: 1 when the lower 32 bits of the operation result is 0 and the operation results are 0; 0 in all other cases</p>						

MULU Reg1, Reg2, Reg3, Reg4						
Operation	<p>$\text{Reg1} * \text{Reg2} \rightarrow \{\text{Reg3}, \text{Reg4}\}$</p> <p>The contents of Reg1 (signed 32-bit integer: multiplicand) and Reg2 (signed 32-bit integer: multiplier) are multiplied, and the upper 32 bits of the result (64 bits) are written into Reg3 and the lower 32 bits into Reg4.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mulu Rm, Rn, Rd1, Rd2	Rd1=Rd2 cannot be specified.	?	?	Δ	Δ	4
Flag change						
<p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: 1 when the MSB of the operation result (64 bits) is 1; 0 in all other cases</p> <p>ZF: 1 when the operation result (64 bits) is 0; 0 in all other cases</p>						



When Rd1=Rd2 is specified, the operation result is not guaranteed.

MULU imm, Reg						
Operation	<p>$(\text{sign_ext})\text{imm} * \text{Reg} \rightarrow \text{Reg}$</p> <p>imm (unsigned 32-bit integer: multiplicand) and Reg (unsigned 32-bit integer: multiplier) are multiplied, and the upper 32 bits of the result (64 bits) are written into MDR and the lower 32 bits into Reg.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mulu imm8, Rn	imm8 is sign-extended.	?	?	Δ	Δ	4
mulu imm24, Rn	imm24 is sign-extended.	?	?	Δ	Δ	6
mulu imm32, Rn		?	?	Δ	Δ	7
Flag change						
<p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: 1 when bit 31 of the lower32 bits of the operation result is 1; 0 in all other cases</p> <p>ZF: 1 when the lower 32 bits of the operation result is 0 ; 0 in all other cases</p>						

DIV

Signed divide

DIV Reg1, Reg2						
Operation	<p>{ MDR , Reg2 / Reg1 -> Reg2 { MDR , Reg2 } % Reg1 -> MDR</p> <p>The signed 64-bit integer obtained by linking MDR (upper 32 bits of the dividend) and Reg2 (lower 32 bits of the dividend) is divided by Reg1 (signed 32-bit integer: divisor), and the remainder (32 bits) is written into MDR and the quotient (32 bits) into Reg2.</p> <p>When the quotient cannot be expressed as a 32-bit signed number, the V flag is set to 1 and MDR and Reg2 are undefined.</p> <p>In addition, the V flag also is set to 1 when zero division (divisor = 0) is performed</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
div Dm, Dn		?	?	Δ	Δ	2
div Rm, Rn		?	?	Δ	Δ	3
Flag change						
<p>Normal (Operation finishes normally)</p> <p>VF: Always 0. This indicates that the quotient is a 32-bit signed integer without overflow.</p> <p>CF: Undefined</p> <p>NF: 1 when MSB of the quotient (32 bits) is 1; 0 in all other cases</p> <p>ZF: 1 when the quotient (32 bits) is 0; 0 in all other cases</p> <p>When the quotient cannot be indicated as a 32-bit signed value, or when in zero divide.</p> <p>VF: Always 1.</p> <p>CF: Undefined</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						



This instruction is a multi-cycle instruction, and the number of cycles differs depending on the significant number of bytes of the dividend (64bits). The significant number of bytes from the LSB of the dividend is judged (note that MDR is judged in word units), and the operation is only performed for the range containing these significant values. Consequently the fewer significant number of bytes of the dividend, the faster operation result can be obtained.

Refer to the instruction list for the number of cycles.

[Example]

the dividend	significant number of bytes
0x000000000000001112 bytes	
0xFFFFFFFFFFFF80000	3 bytes

!

Refer to the instruction list for the number of cycles.

the dividend significant number of bytes

significant number of bytes

3 bytes

INC

Increment by 1

INC Reg						
Operation	<div>Reg + 1 -> Reg</div> <div>1 is added to Reg, and the result is stored in Reg.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
inc Dn		Δ	Δ	Δ	Δ	1
inc An		–	–	–	–	1
inc Rn		Δ	Δ	Δ	Δ	3
Flag change						
<div>inc An</div> <div>VF: This is not changed.</div> <div>CF: This is not changed.</div> <div>NF: This is not changed.</div> <div>ZF: This is not changed.</div> <div>All other cases</div> <div>VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases</div> <div>CF: 1 when carry from bit 31 occurs; 0 in all other cases</div> <div>NF: 1 when bit 31 of the operation results is 1; 0 in all other cases</div> <div>ZF: 1 when the operation results are 0; 0 in all other cases</div>						

INC4

Increment by 4

INC4 Reg						
Operation	<div>Reg + 4 -> Reg</div> <div>4 is added to Reg, and the result id stored in Reg.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
inc4 An		–	–	–	–	1
inc4 Rn		Δ	Δ	Δ	Δ	3
Flag change						
<div>inc An</div> <div>VF: This is not changed.</div> <div>CF: This is not changed.</div> <div>NF: This is not changed.</div> <div>ZF: This is not changed.</div> <div>All other cases</div> <div>VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases</div> <div>CF: 1 when carry from bit 31 occurs; 0 in all other cases</div> <div>NF: 1 when bit 31 of the operation results is 1; 0 in all other cases</div> <div>ZF: 1 when the operation results are 0; 0 in all other cases</div>						

CMP Comparison

CMP Reg1, Reg2						
Operation	Reg2 - Reg1: EPSW This instruction subtracts the contents of Reg1 from Reg2, and reflects the result to the flag. The contents of Reg1 and Reg2 are not changes before and after execution.					
Assembler mnemonic	Note	V	C	N	Z	Size
cmp Dm, Dn	m=n cannot be specified.	Δ	Δ	Δ	Δ	1
cmp Dm, An		Δ	Δ	Δ	Δ	2
cmp Am, Dn		Δ	Δ	Δ	Δ	2
cmp Am, An	m=n cannot be specified.	Δ	Δ	Δ	Δ	1
cmp Rm, Rn		Δ	Δ	Δ	Δ	3
Flag change						
VF: 1 when overflow occurs as a 32-bit signed number; 0 in all other cases CF: 1 when carry from bit 31 occurs; 0 in all other cases NF: 1 when bit 31 of the operation results is 1; 0 in all other cases ZF: 1 when the operation results are 0; 0 in all other cases						

CMP imm, Reg						
Operation	Rn2 - imm : EPSW This instruction subtracts imm from Reg, and reflects the result to the flag. The contents of Reg1 and Reg2 are not changes before and after execution.					
Assembler mnemonic	Note	V	C	N	Z	Size
cmp imm8, Dn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	2
cmp imm16, Dn	imm16 is sign-extended.	Δ	Δ	Δ	Δ	4
cmp imm32, Dn		Δ	Δ	Δ	Δ	6
cmp imm8, An	imm8 is sign-extended.	Δ	Δ	Δ	Δ	2
cmp imm16, An	imm16 is sign-extended.	Δ	Δ	Δ	Δ	4
cmp imm32, An		Δ	Δ	Δ	Δ	6
cmp imm8, Rn	imm8 is sign-extended.	Δ	Δ	Δ	Δ	4
cmp imm24, Rn	imm24 is sign-extended.	Δ	Δ	Δ	Δ	6
cmp imm32, Rn		Δ	Δ	Δ	Δ	7
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all ther cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

AND

Logical product

AND Reg1, Reg2						
Operation	Reg1 & Reg2 -> Reg2 This instruction takes the logical product between Reg1 and Reg2, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
and Dm, Dn		0	0	Δ	Δ	2
and Rm, Rn		0	0	Δ	Δ	3
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

AND Reg1, Reg2, Reg3						
Operation	Reg1 & Reg2 -> Reg2 This instruction takes the logical product between Reg1 and Reg2, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
and Rm, Rn, Rd		0	0	Δ	Δ	4
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

AND imm, Reg						
Operation	(zero_ext)imm & Reg -> Reg This instruction takes the logical product between zero-extended imm and Reg, and stores the result in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
and imm8, Dn	imm8 is zero-extended.	0	0	Δ	Δ	3
and imm16, Dn	imm16 is zero-extended.	0	0	Δ	Δ	4
and imm32, Dn		0	0	Δ	Δ	6
and imm8, Rn	imm8 is zero-extended.	0	0	Δ	Δ	4
and imm24, Rn	imm24 is zero-extended.	0	0	Δ	Δ	6
and imm32, Rn		0	0	Δ	Δ	7
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

Logical instruction

AND imm16, PSW (Privileged instruction)

Operation	imm16 & EPSW[15:0] -> EPSW[15:0] This instruction takes the logical product between imm16 and the lower 16 bits of EPSW, and stores the result in the lower 16 bits of EPSW. The upper 16 bits of EPSW (EPSW[32:16]) are not changed. This instruction is a privileged instruction, and can be executed at the supervisor and monitor levels.					
Assembler mnemonic	Note	V	C	N	Z	Size
and imm16, PSW		Δ	Δ	Δ	Δ	4
Flag change						
VF: The bit 3 of the operation result is set. CF: The bit 2 of the operation result is set. NF: The bit 1 of the operation result is set. ZF: The bit 0 of the operation result is set.						



If this instruction is executed at user level, system exception (privileged instruction execution exception) occurs.

AND imm32, EPSW (Privileged instruction)

Operation	imm32(SP) & EPSW -> EPSW This instruction takes the logical product between imm32 and EPSW, and stores the result in EPSW. This instruction is a privileged instruction, and can be executed at the supervisor and monitor levels.					
Assembler mnemonic	Note	V	C	N	Z	Size
and imm32, EPSW		Δ	Δ	Δ	Δ	6
Flag change						
VF: The bit 3 of the operation result is set. CF: The bit 2 of the operation result is set. NF: The bit 1 of the operation result is set. ZF: The bit 0 of the operation result is set.						



If this instruction is executed at user level, system exception (privileged instruction execution exception) occurs.

OR

Logical sum

OR Reg1, Reg2						
Operation	Reg1 Reg2 -> Reg2 This instruction takes the logical sum between Reg1 and Reg2, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
or Dm, Dn		0	0	Δ	Δ	2
or Rm, Rn		0	0	Δ	Δ	3
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

OR Reg1, Reg2, Reg3						
Operation	Reg1 Reg2 -> Reg2 This instruction takes the logical sum between Reg1 and Reg2, and stores the result in Reg3.					
Assembler mnemonic	Note	V	C	N	Z	Size
and Rm, Rn, Rd		0	0	Δ	Δ	4
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

OR imm, Reg

Operation	(zero_ext)imm Reg -> Reg This instruction takes the logical sum between zero-extended imm and Reg, and stores the result in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
or imm8, Dn	imm8 is zero-extended.	0	0	Δ	Δ	3
or imm16, Dn	imm16 is zero-extended.	0	0	Δ	Δ	4
or imm32, Dn		0	0	Δ	Δ	6
or imm8, Rn	imm8 is zero-extended.	0	0	Δ	Δ	4
or imm24, Rn	imm24 is zero-extended.	0	0	Δ	Δ	6
or imm32, Rn		0	0	Δ	Δ	7
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

OR imm16, PSW (Privileged instruction)

Operation	imm16 EPSW[15:0] -> EPSW[15:0] This instruction takes the logical sum between imm16 and the lower 16 bits of EPSW, and stores the result in the lower 16 bits of EPSW. The upper 16 bits of EPSW (EPSW[32:16]) are not changed. This instruction is a privileged instruction, and can be executed at the supervisor and monitor levels.					
Assembler mnemonic	Note	V	C	N	Z	Size
or imm16, PSW		Δ	Δ	Δ	Δ	4
Flag change						
VF: The bit 3 of the operation result is set. CF: The bit 2 of the operation result is set. NF: The bit 1 of the operation result is set. ZF: The bit 0 of the operation result is set.						



If this instruction is executed at user level, system exception (privileged instruction execution exception) occurs.

OR imm32, EPSW (Privileged instruction)

Operation	imm32(SP) EPSW -> EPSW This instruction takes the logical sum between imm32 and EPSW, and stores the result in EPSW. This instruction is a privileged instruction, and can be executed at the supervisor and monitor levels.					
Assembler mnemonic	Note	V	C	N	Z	Size
or imm32, EPSW		Δ	Δ	Δ	Δ	6
Flag change						
VF: The bit 3 of the operation result is set. CF: The bit 2 of the operation result is set. NF: The bit 1 of the operation result is set. ZF: The bit 0 of the operation result is set.						



If this instruction is executed at user level, system exception (privileged instruction execution exception) occurs.

XOR

Exclusive-OR

XOR Reg1, Reg2						
Operation	Reg1 ^ Reg2 -> Reg2 This instruction takes the exclusive logical sum between Reg1 and Reg2, and stores the result in Reg2.					
Assembler mnemonic	Note	V	C	N	Z	Size
xor Dm, Dn		0	0	Δ	Δ	2
xor Rm, Rn		0	0	Δ	Δ	3
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

XOR Reg1, Reg2, Reg3						
Operation	Reg1 ^ Reg2 -> Reg2 This instruction takes the exclusive logical sum between Reg1 and Reg2, and stores the result in Reg3.					
Assembler mnemonic	Note	V	C	N	Z	Size
xor Rm, Rn, Rd		0	0	Δ	Δ	4
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

XOR imm, Reg						
Operation	(zero_ext)imm ^ Reg -> Reg This instruction takes the exclusive logical sum between zero-extended imm and Reg, and stores the result in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
xor imm16, Dn	imm16 is zero-extended.	0	0	Δ	Δ	4
xor imm32, Dn		0	0	Δ	Δ	6
xor imm8, Rn	imm8 is zero-extended.	0	0	Δ	Δ	4
xor imm24, Rn	imm24 is zero-extended.	0	0	Δ	Δ	6
xor imm32, Rn		0	0	Δ	Δ	7
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

NOT

Reversion of all bits

NOT Reg						
Operation	Reg ^ 0xFFFFFFFF -> Reg This instruction reverses all bits of Reg, and stores the result in Reg.					
Assembler mnemonic	Note	V	C	N	Z	Size
not Dn		0	0	Δ	Δ	2
not Rn		0	0	Δ	Δ	3
Flag change						
VF: Always 0. CF: Always 0. NF: 1 when bit 31 of the operation result is 1, 0 in all other cases. ZF: 1 when the operation result is 0, 0 in all other cases.						

BTST

Multiple bit test

BTST imm, Reg						
Operation	(zero_ext)imm & Reg : EPSW The logical product of the zero-extended imm and Reg is executed, and the result is reflected to the flag.					
Assembler mnemonic	Note	V	C	N	Z	Size
btst imm8, Dn	imm8 is zero-extended.	0	0	Δ	Δ	3
btst imm16, Dn	imm16 is zero-extended.	0	0	Δ	Δ	4
btst imm32, Dn		0	0	Δ	Δ	6
btst imm8, Rn	imm8 is zero-extended.	0	0	Δ	Δ	4
btst imm24, Rn	imm24 is zero-extended.	0	0	Δ	Δ	6
btst imm32, Rn		0	0	Δ	Δ	7
Flag change						
VF : Always "0" CF : Always "0" NF : 1 when bit 31 of the operation result is 1; 0 in all other cases ZF : 1 when the operation result is 0; 0 in all other cases						

Bit manipulate instruction

BTST imm, Mem						
Operation	BTST imm, (disp, Reg) Mem8(disp, Reg) -> TMP[7:0] 0x000000 -> TMP[31:8] imm & TMP : EPSW BTST imm, (abs) Mem8(abs) -> TMP[7:0] 0x000000 -> TMP[31:8] imm & TMP : EPSW The logical product of imm and the contents (byte data) of Mem zero-extended to 32 bits is executed, and the result is reflected to the flag.					
Assembler mnemonic	Note	V	C	N	Z	Size
btst imm8, (d8, An)	imm8 is zero-extended, and d8 is sign-extended.	0	0	Δ	Δ	4
btst imm8, (abs16)	imm8 is zero-extended, and abs16 is zero-extended.	0	0	Δ	Δ	5
btst imm8, (abs32)	imm8 is zero-extended.	0	0	Δ	Δ	7
Flag change						
VF : Always "0" CF : Always "0" NF : 1 when bit 31 of the operation result is 1; 0 in all other cases ZF : 1 when the operation result is 0; 0 in all other cases						



btst imm8, (abs16) implements AM33-2, AM33-2A, AM34-1.

BSET


Multiple bit tests and sets

BSET Reg, Mem						
Operation	BSET Reg1, (Reg2) Mem8(Reg2) -> TMP[7:0] 0x000000 -> TMP[31:8] TMP & Reg1 : EPSW (TMP Reg1)[7:0] -> Mem8(Reg2) 1. The contents (byte data) of Mem are zero-extended to 32 bits and loaded into the internal temporary register (TMP). 2. The logical product of the contents of the temporary register (TMP) and the contents of Reg is taken and the operation flag results are reflected to EPSW. 3. The logical product of the contents of the temporary register (TMP) and the contents of Reg is taken and the lower 8 bits of the result are stored in Mem.					
Assembler mnemonic	Note	V	C	N	Z	Size
bset Dm, (An)		0	0	0	Δ	2
Flag change						
VF : Always 0. CF : Always 0. NF : Always 0. ZF : "1" when the operation result is "0", "0" in all other cases.						



All operations for this instruction are executed in the bus-lock and interrupt prohibited condition.

BSET imm, Mem						
Operation	<div>BTST imm, (disp, Reg) Mem8(Reg + disp) -> TMP[7:0] 0x000000 -> TMP[31:8] TMP & imm : EPSW (TMP imm) [7:0] -> mem8(Reg + disp)</div> <div>BTST imm, (abs) Mem8(abs) -> TMP[7:0] 0x000000 -> TMP[31:8] TMP & imm : EPSW (TMP imm) [7:0] -> mem8(abs)</div> <div>1. The contents (byte data) of Mem are zero-extended to 32 bits and loaded into the internal temporary register (TMP). 2. The logical product of the contents of the temporary register (TMP) and imm is taken and the operation flag results are reflected to PSW. 3. The logical product of the contents of the temporary register (TMP) and imm is taken and the lower 8 bits of the result are stored in Mem.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
bset imm8, (d8, An)	imm8 is zero-extended, and d8 is sign-extended.	0	0	0	Δ	4
bset imm8, (abs16)	imm8 is zero-extended, and abs16 is zero-extended.	0	0	0	Δ	5
bset imm8, (abs32)	imm8 is zero-extended.	0	0	0	Δ	7
Flag change						
VF : Always "0" CF : Always "0" NF : Always "0" ZF : 1 when the operation result is 0; 0 in all other cases						



All operations for this instruction are executed in the bus-lock and interrupt prohibited condition.



bset imm8, (abs16) implements AM33-2, AM33-2A, AM34-1.

BCLR

Multiple bit tests and clearing

BCLR Reg, Mem								
Operation	<div>BCLR Reg1, (Reg2) mem8(Reg2) -> TMP[7:0] 0x000000 -> TMP[31:8] TMP & Reg1 : EPSW (TMP &(Reg ^ 0xFFFFFFFF))[7:0] -> mem8(Reg2)</div> <div><div>1. The contents (byte data) of Mem are zero-extended to 32 bits and loaded into the internal temporary register (TMP).</div><div>2. The logical product of the contents of the temporary register (TMP) and the contents of Reg1 is taken and the operation flag results are reflected to EPSW.</div><div>3. The logical product of the contents of the temporary register (TMP) and the logically inverted contents of Reg1 is taken and the lower 8 bits of the result are stored in Mem.</div></div>							
	Assembler mnemonic		Note		V	C	N	Z
bclr Dm, (An)				0	0	0	Δ	2
		Flag change						
<div>VF : Always 0.</div> <div>CF : Always 0.</div> <div>NF : Always 0.</div> <div>ZF : "1" when the operation result is "0", "0" in all other cases.</div>								



All operations for this instruction are executed in the bus-lock and interrupt prohibited condition.

BCLR imm, Mem

Operation	<p> BCLR imm, (disp, Reg) mem8(Reg + disp) -> TMP[7:0] 0x000000 -> TMP[31:8] TMP & imm : EPSW (TMP & (imm ^ 0xFFFFFFFF)) -> mem8(Reg + disp) </p> <p> BCLR imm, (abs) mem8(abs2) -> TMP[7:0] 0x000000 -> TMP[31:8] TMP & imm : EPSW (TMP & (imm ^ 0xFFFFFFFF)) -> mem8(abs) </p> <ol style="list-style-type: none"> 1. The contents (byte data) of Mem are zero-extended to 32 bits and loaded into the internal temporary register (TMP). 2. The logical product of the contents of the temporary register (TMP) and imm is taken and the operation flag results are reflected to EPSW. 3. The logical product of the contents of the temporary register (TMP) and the logically inverted data of imm is taken and the lower 8 bits of the result are stored in Mem. 					
Assembler mnemonic	Note	V	C	N	Z	Size
bclr imm8, (d8, An)	imm8 is zero-extended, and d8 is sign-extended.	0	0	0	Δ	4
bclr imm8, (abs16)	imm8 is zero-extended, and abs16 is zero-extended.	0	0	0	Δ	5
bclr imm8, (abs32)	imm8 is zero-extended.	0	0	0	Δ	7
<p>Flag change</p> <p> VF : Always "0" CF : Always "0" NF : Always "0" ZF : 1 when the operation result is 0; 0 in all other cases </p>						



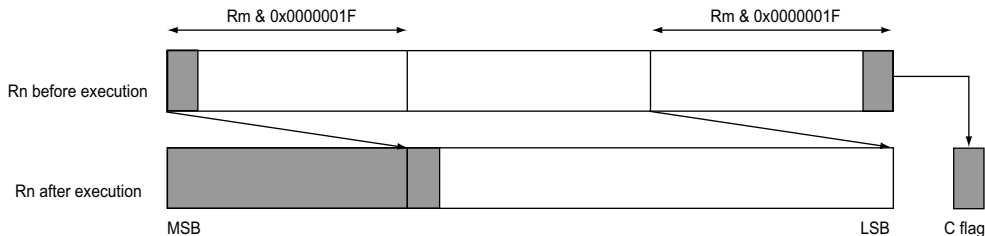
All operations for this instruction are executed in the bus-lock and interrupt prohibited condition.



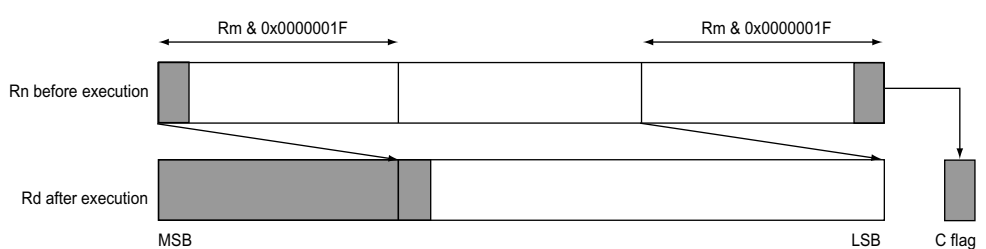
bclr imm8, (abs16) implements AM33-2, AM33-2A, AM34-1.

ASR

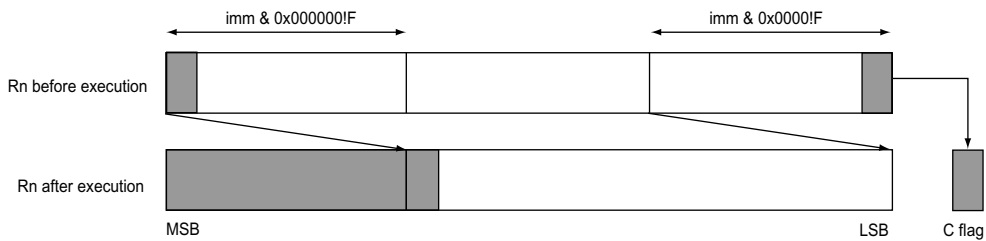
Arbitrary-bit arithmetic shift right

ASR Reg1, Reg2						
Operation	If (Reg1 & 0x0000001F) is not "0" Reg2[0] -> EPSW.C (sign-ext)(Reg2 >> (Reg1 & 0x0000001F)) -> Reg2 If (Reg1 & 0x0000001F) is "0" PC + Code Size -> PC					
	The contents of Reg2 are arithmetically shifted to the right by the number of bits specified by Reg1, and the result is written into Reg2. When (Reg1 & 0x0000001F) are 0, shift operation is not performed. Only the lower 5 bits of Reg1 are effective and the upper bits are ignored.					
						
Assembler mnemonic	Note	V	C	N	Z	Size
asr Dm, Dn	The contents of the lower 5 bits in Dm are other than "0"	?	Δ	Δ	Δ	2
	The contents of the lower 5 bits in Dm are "0"	?	?	Δ	Δ	
asr Rm, Rn	The contents of the lower 5 bits in Rm are other than "0"	?	Δ	Δ	Δ	3
	The contents of the lower 5 bits in Rm are "0"	?	?	Δ	Δ	
Flag change						
When (Reg1 & 0x00001F) is not "0"						
VF: Undefined						
CF: Reflects the value of the bit initially shifted out						
NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases.						
ZF: "1" when the operation result is "0", and "0" in all other cases.						
When (Reg1 & 0x00001F) is "0"						
VF: Undefined						
CF: Undefined						
NF: "1" when bit 31 of the register (Reg2) is "1", and "0" in all other cases.						
ZF: "1" when the register (Reg2) is "0", and "0" in all other cases.						

ASR Reg1, Reg2, Reg3

Operation	<p>If (Reg1 & 0x0000001F) is not "0"</p> <p>Reg2[0] -> EPSW.C</p> <p>(sign-ext)(Reg2 >> (Reg1 & 0x0000001F)) -> Reg2</p> <p>If (Reg1 & 0x0000001F) is "0"</p> <p>PC + Code Size -> PC</p> <p>The contents of Reg2 are arithmetically shifted to the right by the number of bits specified by Reg1, and the result is written into Reg2.</p> <p>When (Reg1 & 0x0000001F) are 0, shift operation is not performed.</p> <p>Only the lower 5 bits of Reg1 are effective and the upper bits are ignored.</p> 					
Assembler mnemonic	Note	V	C	N	Z	Size
asr Rm, Rn, Rd	The contents of the lower 5 bits in Rm are other than "0"	?	Δ	Δ	Δ	4
	The contents of the lower 5 bits in Rm are "0"	?	?	Δ	Δ	
Flag change						
<p>When (Reg1 & 0x0000001F) is not "0"</p> <p>VF: Undefined</p> <p>CF: Reflects the value of the bit initially shifted out</p> <p>NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases.</p> <p>ZF: "1" when the operation result is "0", and "0" in all other cases.</p> <p>When (Reg1 & 0x0000001F) is "0"</p> <p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: "1" when bit 31 of the register (Reg2) is "1", and "0" in all other cases.</p> <p>ZF: "1" when the register (Reg2) is "0", and "0" in all other cases.</p>						

ASR imm, Reg

Operation	<p>If (imm & 0x0000001F) is not "0"</p> <p>Reg2[0] -> EPSW.C</p> <p>(sign-ext)(Reg2 >> (imm & 0x0000001F)) -> Reg</p> <p>If (imm & 0x0000001F) is "0"</p> <p>PC + Code Size -> PC</p> <p>The contents of Reg are arithmetically shifted to the right by the number of bits specified by the lower 5 bits of imm, and the result is written into Reg.</p> <p>When the imm lower 5 bits are 0, shift operation is not performed.</p> <p>Only the lower 5 bits in imm are effective and the upper bits are ignored.</p> 
-----------	---

Assembler mnemonic	Note	V	C	N	Z	Size
asr imm8, Dn	When (imm& 0x0000001F) is not "0"	?	Δ	Δ	Δ	3
	When (imm& 0x0000001F) is "0"	?	?	Δ	Δ	
asr imm8, Rn	When (imm& 0x0000001F) is not "0"	?	Δ	Δ	Δ	4
	When (imm& 0x0000001F) is "0"	?	?	Δ	Δ	
asr imm24, Rn	When (imm& 0x0000001F) is not "0"	?	Δ	Δ	Δ	6
	When (imm& 0x0000001F) is "0"	?	?	Δ	Δ	
asr imm32, Rn	When (imm& 0x0000001F) is not "0"	?	Δ	Δ	Δ	7
	When (imm& 0x0000001F) is "0"	?	?	Δ	Δ	

Flag change

When (imm & 0x0000001F) is not "0"

VF: Undefined

CF: Reflects the value of the bit initially shifted out

NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases.

ZF: "1" when the operation result is "0", and "0" in all other cases.

When (imm & 0x0000001F) is "0"

VF: Undefined

CF: Undefined

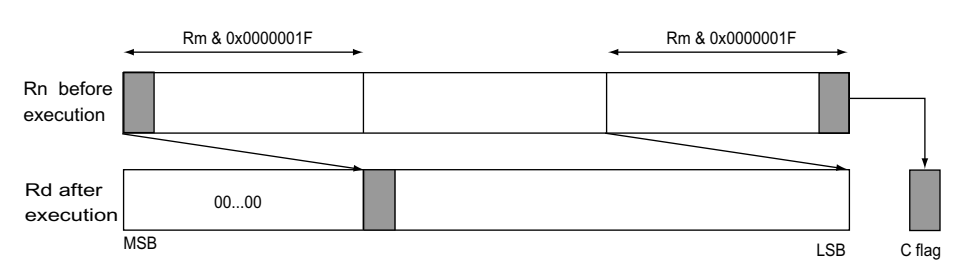
NF: "1" when bit 31 of the register (Reg) is "1", and "0" in all other cases.

ZF: "1" when the register (Reg) is "0", and "0" in all other cases.

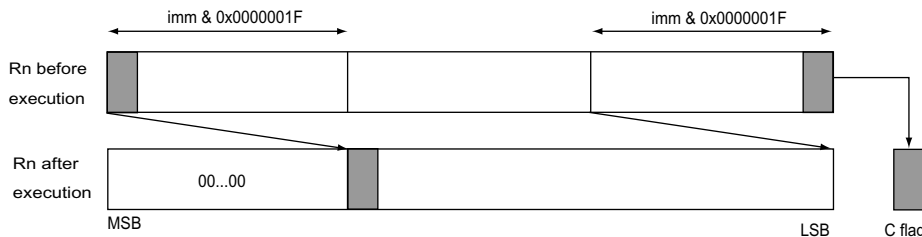
LSR

Arbitrary-bit logical shift right

LSR Reg1, Reg2						
Operation	<div>If (Reg1 & 0x0000001F)is not "0" Reg2[0] -> EPSW.C (zero-ext)(Reg2 >> (Reg1 & 0x0000001F)) -> Reg2 If (Reg1 & 0x0000001F) is "0" PC + Code Size -> PC</div> <div>The contents of Reg2 are logically shifted to the right by the number of bits specified by the lower 5 bits of Reg1, and the result is written into Reg2. When the contents of the lower 5 bits in Reg1 are 0, shift operation is not performed. Only the lower 5 bits of Reg1 are effective and the upper bits are ignored.</div> <div></div>					
Assembler mnemonic	Note	V	C	N	Z	Size
lsr Dm, Dn	The contents of the lower 5 bits in Dm are other than "0"	?	Δ	Δ	Δ	2
	The contents of the lower 5 bits in Dm are "0"	?	?	Δ	Δ	
lsr Rm, Rn	The contents of the lower 5 bits in Rm are other than "0"	?	Δ	Δ	Δ	3
	The contents of the lower 5 bits in Rm are "0"	?	?	Δ	Δ	
Flag change						
<div>When (Reg1 & 0x0000001F) is not "0"</div> <div>VF: Undefined CF: Reflects the value of the bit initially shifted out NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases. ZF: "1" when the operation result is "0", and "0" in all other cases.</div> <div>When (Reg1 & 0x0000001F) is "0"</div> <div>VF: Undefined CF: Undefined NF: "1" when bit 31 of the register (Reg2) is "1", and "0" in all other cases. ZF: "1" when the register (Reg2) is "0", and "0" in all other cases.</div>						

LSR Reg1, Reg2, Reg3						
Operation	<div>If (Reg1 & 0x0000001F) is not "0" Reg2[0] -> EPSW.C (zero-ext)(Reg2 >> (Reg1 & 0x0000001F)) -> Reg3 If (Reg1 & 0x0000001F) is "0" PC + Code Size -> PC</div> <div>The contents of Reg2 are logically shifted to the right by the number of bits specified by the lower 5 bits of Reg1, and the result is written into Reg3. When the lower 5 bits of Reg 1 are 0, shift operation is not performed. Only the lower 5 bits of Reg1 are effective and the upper bits are ignored.</div> <div></div>					
	Assembler mnemonic	Note	V	C	N	Z
lsr Rm, Rn, Rd	The contents of the lower 5 bits in Rm are other than "0"	?	Δ	Δ	Δ	4
	The contents of the lower 5 bits in Rm are "0"	?	?	Δ	Δ	
Flag change						
<div>When (Reg1 & 0x0000001F) is not "0" VF: Undefined CF: Reflects the value of the bit initially shifted out NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases. ZF: "1" when the operation result is "0", and "0" in all other cases.</div> <div>When (Reg1 & 0x0000001F) is "0" VF: Undefined CF: Undefined NF: "1" when bit 31 of the register (Reg2) is "1", and "0" in all other cases. ZF: "1" when the register (Reg2) is "0", and "0" in all other cases.</div>						

LSR imm, Reg

Operation	<div>If (imm & 0x0000001F) is not "0" Reg[0] -> EPSW.C (zero-ext)(Reg2 >> (imm & 0x0000001F)) -> Reg If (imm & 0x0000001F) is "0" PC + Code Size -> PC</div> <div>The contents of Reg are arithmetically shifted to the right by the number of bits specified by the lower 5 bits of imm, and the result is written into Reg. When the imm lower 5 bits are 0, shift operation is not performed. Only the lower 5 bits in imm are effective and the upper bits are ignored.</div> <div></div>																																																						
Assembler mnemonic	<table><tr><th>Note</th><th>V</th><th>C</th><th>N</th><th>Z</th><th>Size</th></tr><tr><td rowspan="2">lsr imm8, Dn</td><td>When the imm lower 5 bits are not "0"</td><td>?</td><td>Δ</td><td>Δ</td><td>Δ</td><td rowspan="2">3</td></tr><tr><td>When the imm lower 5 bits are "0"</td><td>?</td><td>?</td><td>Δ</td><td>Δ</td></tr><tr><td rowspan="2">lsr imm8, Rn</td><td>When the imm lower 5 bits are not "0"</td><td>?</td><td>Δ</td><td>Δ</td><td>Δ</td><td rowspan="2">4</td></tr><tr><td>When the imm lower 5 bits are "0"</td><td>?</td><td>?</td><td>Δ</td><td>Δ</td></tr><tr><td rowspan="2">lsr imm24, Rn</td><td>When the imm lower 5 bits are not "0"</td><td>?</td><td>Δ</td><td>Δ</td><td>Δ</td><td rowspan="2">6</td></tr><tr><td>When the imm lower 5 bits are "0"</td><td>?</td><td>?</td><td>Δ</td><td>Δ</td></tr><tr><td rowspan="2">lsr imm32, Rn</td><td>When the imm lower 5 bits are not "0"</td><td>?</td><td>Δ</td><td>Δ</td><td>Δ</td><td rowspan="2">7</td></tr><tr><td>When the imm lower 5 bits are "0"</td><td>?</td><td>?</td><td>Δ</td><td>Δ</td></tr></table>	Note	V	C	N	Z	Size	lsr imm8, Dn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	3	When the imm lower 5 bits are "0"	?	?	Δ	Δ	lsr imm8, Rn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	4	When the imm lower 5 bits are "0"	?	?	Δ	Δ	lsr imm24, Rn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	6	When the imm lower 5 bits are "0"	?	?	Δ	Δ	lsr imm32, Rn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	7	When the imm lower 5 bits are "0"	?	?	Δ	Δ
Note	V	C	N	Z	Size																																																		
lsr imm8, Dn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	3																																																	
	When the imm lower 5 bits are "0"	?	?	Δ	Δ																																																		
lsr imm8, Rn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	4																																																	
	When the imm lower 5 bits are "0"	?	?	Δ	Δ																																																		
lsr imm24, Rn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	6																																																	
	When the imm lower 5 bits are "0"	?	?	Δ	Δ																																																		
lsr imm32, Rn	When the imm lower 5 bits are not "0"	?	Δ	Δ	Δ	7																																																	
	When the imm lower 5 bits are "0"	?	?	Δ	Δ																																																		
Flag change																																																							
<div>When (imm & 0x0000001F) is not "0"</div> <div>VF: Undefined CF: Reflects the value of the bit initially shifted out NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases. ZF: "1" when the operation result is "0", and "0" in all other cases.</div> <div>When (imm & 0x0000001F) is "0"</div> <div>VF: Undefined CF: Undefined NF: "1" when bit 31 of the register (Reg) is "1", and "0" in all other cases. ZF: "1" when the register (Reg) is "0", and "0" in all other cases.</div>																																																							

ASL

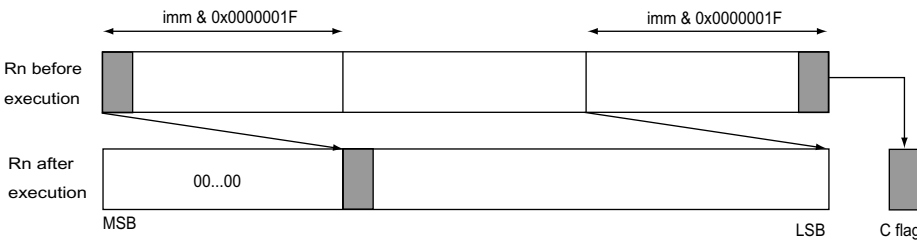
Arbitrary-bit arithmetic shift left

ASL Reg1, Reg2						
Operation	<div>If (Reg1 & 0x0000001F)is not "0" (Reg2 << (Reg1 & 0x0000001F)) -> Reg2 If (Reg1 & 0x0000001F) is "0" PC + Code Size -> PC</div> <div>The contents of Reg 2 are arithmetically shifted to the left by the number of bits specified by the lower 5 bits of Reg1, and the result is written into Reg 2. When the contents of the lower 5 bits in Reg 1 are 0, shift operation is not performed. Only the lower 5 bits of Reg1 are effective and the upper bits are ignored. "0" is input to the LSB side.</div> <div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div></div>					

ASL Reg1, Reg2, Reg3

Operation	<p>If (Reg1 & 0x0000001F) is not "0" (Reg2 << (Reg1 & 0x0000001F)) -> Reg3</p> <p>If (Reg1 & 0x0000001F) is "0" PC + Code Size -> PC</p> <p>The contents of Reg 2 are arithmetically shifted to the left by the number of bits specified by the lower 5 bits of Reg1, and the result is written into Reg 3.</p> <p>When the contents of the lower 5 bits in Reg 1 are 0, shift operation is not performed.</p> <p>Only the lower 5 bits of Reg1 are effective and the upper bits are ignored.</p> <p>"0" is input to the LSB side.</p> <div><div><div>Rm & 0x0000001F</div><div>Rm & 0x0000001F</div></div><div><div>Rn before execution</div><div>Rd after execution</div></div><div><div>MSB</div><div>LSB</div></div><div><div>00...00</div></div></div>																	
Assembler mnemonic	<table><tr><th>Note</th><th>V</th><th>C</th><th>N</th><th>Z</th><th>Size</th></tr><tr><td>The contents of the lower 5 bits in Rm are other than "0"</td><td>?</td><td>?</td><td>Δ</td><td>Δ</td><td rowspan="2">4</td></tr><tr><td>The contents of the lower 5 bits in Rm are "0"</td><td>?</td><td>?</td><td>Δ</td><td>Δ</td></tr></table>	Note	V	C	N	Z	Size	The contents of the lower 5 bits in Rm are other than "0"	?	?	Δ	Δ	4	The contents of the lower 5 bits in Rm are "0"	?	?	Δ	Δ
Note	V	C	N	Z	Size													
The contents of the lower 5 bits in Rm are other than "0"	?	?	Δ	Δ	4													
The contents of the lower 5 bits in Rm are "0"	?	?	Δ	Δ														
Flag change																		
<p>When (Reg1 & 0x0000001F) is not "0"</p> <p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases.</p> <p>ZF: "1" when the operation result is "0", and "0" in all other cases.</p> <p>When (Reg1 & 0x0000001F) is "0"</p> <p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: "1" when bit 31 of the register (Reg2) is "1", and "0" in all other cases.</p> <p>ZF: "1" when the register (Reg2) is "0", and "0" in all other cases.</p>																		

ASL imm, Reg

Operation	<p>If (imm & 0x0000001F) is not "0" (Reg << (imm & 0x0000001F)) -> Reg</p> <p>If (imm & 0x0000001F) is "0" PC + Code Size -> PC</p> <p>The contents of Reg are arithmetically shifted to the left by the number of bits specified by the lower 5 bits of imm, and the result is written into Reg 3.</p> <p>When the imm lower 5 bits are 0, shift operation is not performed.</p> <p>Only the imm lower 5 bits are effective and the upper bits are ignored.</p> <p>"0" is input to the LSB side.</p> 					
Assembler mnemonic	Note	V	C	N	Z	Size
asl imm8, Dn	When the imm lower 5 bits are not "0"	?	?	Δ	Δ	3
	When the imm lower 5 bits are "0"	?	?	Δ	Δ	
asl imm8, Rn	When the imm lower 5 bits are not "0"	?	?	Δ	Δ	4
	When the imm lower 5 bits are "0"	?	?	Δ	Δ	
asl imm24, Rn	When the imm lower 5 bits are not "0"	?	?	Δ	Δ	6
	When the imm lower 5 bits are "0"	?	?	Δ	Δ	
asl imm32, Rn	When the imm lower 5 bits are not "0"	?	?	Δ	Δ	7
	When the imm lower 5 bits are "0"	?	?	Δ	Δ	
Flag change						
<p>When (imm & 0x0000001F) is not "0"</p> <p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: "1" when bit 31 of the operation result is "1", and "0" in all other cases.</p> <p>ZF: "1" when the operation result is "0", and "0" in all other cases.</p> <p>When (imm & 0x0000001F) is "0"</p> <p>VF: Undefined</p> <p>CF: Undefined</p> <p>NF: "1" when bit 31 of the register (Reg) is "1", and "0" in all other cases.</p> <p>ZF: "1" when the register (Reg) is "0", and "0" in all other cases.</p>						

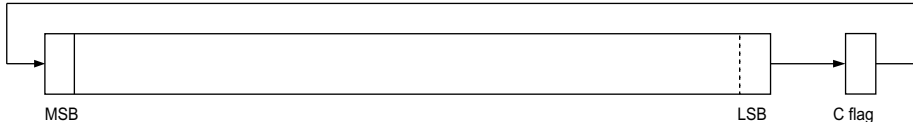
ASL2

2-bit arithmetic shift left

ASL2 Reg						
Operation	Reg << 2 -> Reg					
	The contents of Reg are arithmetically shifted to the left by only 2 bits, and the result is written into Reg .					
<div><div><div>2 bit</div><div><div></div><div></div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div>						

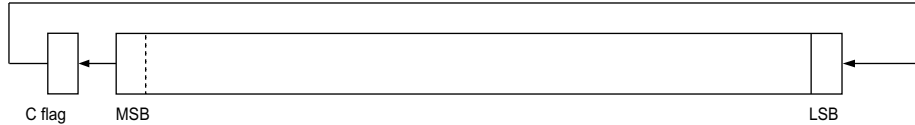
ROR

1-bit rotate right

ROR Reg									
Operation	{EPSW.C , Reg[31:0] } -> {Reg[31:0] , EPSW.C}								
	Reg and the C flag are linked and rotated 1 bit to the right, and the result is written into Reg. C flag previously set reflects to MSB.								
									
Assembler mnemonic		Note			V	C	N	Z	Size
ror Dn					0	Δ	Δ	Δ	2
ror Rn					0	Δ	Δ	Δ	3
Flag change									
VF : Always 0. CF : Reflects the value of the bit shifted out by the operation NF : 1 when bit 31 of the operation result is 1; 0 in all other cases ZF : 1 when the operation result is 0; 0 in all other cases									

ROL

1-bit rotate left

ROL Reg							
Operation	{ Reg[31:0] , EPSW.C} -> { EPSW.C , Reg[31:0]}						
	Reg and the C flag are linked and rotated 1 bit to the left, and the result is written into Reg. C flag previously set reflects to LSB.						
							
Assembler mnemonic		Note	V	C	N	Z	Size
rol Dn			0	Δ	Δ	Δ	2
rol Rn			0	Δ	Δ	Δ	3
Flag change							
VF : Always 0. CF : Reflects the value of the bit shifted out by the operation NF : 1 when bit 31 of the operation result is 1; 0 in all other cases ZF : 1 when the operation result is 0; 0 in all other cases							

Bcc

Conditional branch

Bcc (d8, PC)

Operation	<p>When branch is taken, PC (current instruction address) + (sign_ext) d8 -> nPC (next instruction PC)</p> <p>8-bit displacement (d8) is sign-extended and added to the PC, and the result is written into the PC. Even if the addition result overflows, this overflow is ignored and the result is written into the PC.</p> <p>When branch is not taken, PC (current instruction address) + CodeSize -> nPC (next instruction PC)</p> <p>The next instruction is executed.</p>
-----------	--

Assembler mnemonic	Note	V	C	N	Z	Size
beq label	Z Branch when Z = 1 or Z flag set	–	–	–	–	2
bne label	~Z Branch when Z ≠ 1 or Z flag cleared	–	–	–	–	2
bgt label	~(Z (N ^ V)) Branch when < (signed)	–	–	–	–	2
bge label	~(N ^ V) Branch when ≤ (signed)	–	–	–	–	2
ble label	Z (N ^ V) Branch when ≥ (signed)	–	–	–	–	2
blt label	N ^ V Branch when > (signed)	–	–	–	–	2
bhi label	~(C Z) Branch when < (unsigned)	–	–	–	–	2
bcc label	~C Branch when ≤ or C flag cleared (unsigned)	–	–	–	–	2
blsq label	C Z Branch when ≥ (unsigned)	–	–	–	–	2
bcs label	C Branch when > or C flag set (unsigned)	–	–	–	–	2
bvc label	~V Branch when V flag cleared	–	–	–	–	3
bvs label	V Branch when V flag set	–	–	–	–	3
bnc label	~N Branch when N flag cleared	–	–	–	–	3
bns label	N Branch when N flag set	–	–	–	–	3
bra label	None Unconditional branch	–	–	–	–	2

Flag change

VF: This is not changed.
 CF: This is not changed.
 NF: This is not changed.
 ZF: This is not changed.

Lcc

Loop-only conditional branch

Lcc						
Operation	<p>When branch is taken</p> <p>LAR - 4 -> nPC (next instruction PC)</p> <p>The instruction loaded from the loop instruction register (LIR) is executed and instruction fetch starts for the address loaded to the loop address register (LAR).</p> <p>At the same time, 4 is subtracted from the loop address register (LAR) and the result is written into the PC. Even if the subtract result overflows, this overflow is ignored and the result is written into the PC. When Lcc is not coordinated with SETLB, execution cannot be assured.</p> <p>This is used with SETLB in order to speed up the loop executionl, and performs conditional branch to the top of the loop set by SETLB.</p> <p>When branch is not taken</p> <p>PC (current instruction address) + 1 -> nPC (next instruction PC)</p> <p>The next instruction is executed.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
leq label	Z Branch when Z = 1 or Z flag set	—	—	—	—	1
lne label	~Z Branch when Z ≠ 1 or Z flag cleared	—	—	—	—	1
lgt label	~(Z (N ^ V)) Branch when < (signed)	—	—	—	—	1
lge label	~(N ^ V) Branch when ≤ (signed)	—	—	—	—	1
lle label	Z (N ^ V) Branch when ≥ (signed)	—	—	—	—	1
llt label	N ^ V Branch when > (signed)	—	—	—	—	1
lhi label	~(C Z) Branch when < (unsigned)	—	—	—	—	1
lcc label	~C Branch when ≤ or C flag cleared (unsigned)	—	—	—	—	1
lls label	C Z Branch when ≥ (unsigned)	—	—	—	—	1
lcs label	C Branch when > or C flag set (unsigned)	—	—	—	—	1
lra label	None Unconditional flag	—	—	—	—	1
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						

SETLB

Set loop buffer

SETLB					
Operation	<p>mem32(PC+1) -> LIR, PC + 5 -> LAR</p> <p>The 4-byte instruction string and 5th-byte address following to SETLB are stored to LIR and LAR respectively.</p> <p>SETLB is used together with Lcc in order to speed up the loop (the inner most loop) execution. The top of the loop is set by SETLB just before the loop entrance.</p>				
Assembler mnemonic	Note	V	C	N	Z
setlb		-	-	-	-
Flag change					
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>					



When the instruction row subsequent to the SETLB instruction is as follows:
SETLB

A
B
C
D

:A,B,C,D are 1-byte instructions.

These are stored in the LIR as shown below.

	LIR[31]				[0]
When LAR = 4n	D	C	B	A	
When LAR = 4n+1	C	B	A	D	
When LAR = 4n+2	B	A	D	C	
When LAR = 4n+3	A	D	C	B	

JMP

Unconditional branch

JMP (An)						
Operation	<p>An -> nPC (next instruction PC)</p> <p>This stores the contents of the An register in PC.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
jmp (An)		–	–	–	–	2
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

JMP (d16, PC)						
Operation	<p>When the displacement from PC to label can be represented within 16 bits, PC (current instruction address) + ((sign-extended) d16) -> nPC (next instruction PC)</p> <p>This sign-extends the 16-bit displacement (d16) and adds it to PC, and stores the result in PC. Even if the additon result overflows, the overflow is ignored and the result is stored in PC.</p>					
Assembler mnemonic	Branch condition	V	C	N	Z	Size
jmp label	When the displacement from PC to label can be represented within 16 bits	–	–	–	–	3
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

JMP (d32, PC)

Operation	<p>When the displacement from PC to label can be represented within 32 bits, PC (current instruction address) + ((sign-extended) d32) -> nPC (next instruction PC)</p> <p>This sign-extends the 32-bit displacement (d32) and adds it to PC, and stores the result in PC. Even if the addition result overflows, the overflow is ignored and the result is stored in PC.</p>					
Assembler mnemonic	Branch condition	V	C	N	Z	Size
jmp label	When the displacement from PC to label can be represented within 32 bits	–	–	–	–	5
Flag change						
<p>VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.</p>						

CALL

Subroutine call

CALL (d16, PC), regs, imm8						
Operation	PC (current instruction address) + (sign-ext) d16 -> nPC (next instruction PC)					
	PC (current instruction address) + CodeSize -> mem32(SP)					
	PC (current instruction address) + CodeSize -> MDR					
	Multiple registers specified by regs -> Lower address memory following (SP - 4)					
	SP - (zero-ext) imm8 -> SP					
	<p>This instruction branches to the specified address after saving PC and the multiple registers to the stack and allocating the stack area. d16 is sign-extended and added to the PC, and the result is written into the PC. Even if the addition result overflows, this overflow is ignored and the result is written into the PC. The multiple registers to be saved are specified by regs, and the area to be allocated (number of bytes) by imm8 (zero-extended). (See MOVN for a detailed description of regs.)</p>					
	<p>CALL is used together with RET or RETF in order to save and restore registers and allocate and deallocate the stack area quickly during subroutine call.</p>					
	<p>The stack frame status after executing the CALL instruction is shown below.</p>					
	<div><div>SP after execution →</div><div>SP before execution →</div><div><div>Address low</div><div>Stack area reserved in imm8</div><div>Multiple registers specified by regs</div><div>PC</div><div>Address high</div></div><div>imm8</div></div>					
Assembler mnemonic		Note			V	C
call label, regs, imm8					–	–
					N	Z
					–	–
					Size	
					2	
Flag change						
VF: This is not changed.						
CF: This is not changed.						
NF: This is not changed.						
ZF: This is not changed.						



The three operands of d16, regs and imm8 are used for the bit assignment. Normally, the multiple registers to be saved and the amount of area to be allocated are not specified (regs, imm8) directly by the assembler. Instead, these items are specified indirectly with a pseudo instruction in the callee and ultimately resolved by the linker. For details, refer to MN10300-series cross assembler user's manual.



When the memory address (SP) is not a multiple of 4, a system exception (address misalignment exception) occurs.

CALL (d32, PC), regs, imm8

Operation	<p>PC (current instruction address) + (sign-ext) d32 -> nPC (next instruction PC)</p> <p>PC (current instruction address) + CodeSize -> mem32(SP)</p> <p>PC (current instruction address) + CodeSize -> MDR</p> <p>Multiple registers specified by regs -> Lower address memory following (SP - 4)</p> <p>SP - (zero-ext) imm8 -> SP</p> <p>This instruction branches to the specified address after saving PC and the multiple registers to the stack and allocating the stack area. d32 is sign-extended and added to the PC, and the result is written into the PC. Even if the addition result overflows, this overflow is ignored and the result is written into the PC. The multiple registers to be saved are specified by regs, and the area to be allocated (number of bytes) by imm8 (zero-extended). (See MOVN for a detailed description of regs.)</p> <p>CALL is used together with RET or RETF in order to save and restore registers and allocate and deallocate the stack area quickly during subroutine call. The stack frame status after executing the CALL instruction is shown below.</p> <div><div>SP after execution →</div><div>Address low</div><div>Stack area reserved in imm8</div><div>Multiple registers specified by regs</div><div>PC</div><div>Address high</div><div>imm8</div><div>SP before execution →</div></div>					
Assembler mnemonic	Note	V	C	N	Z	Size
call label, regs, imm8		–	–	–	–	5
Flag change						
VF: This is not changed.						
CF: This is not changed.						
NF: This is not changed.						
ZF: This is not changed.						



The three operands of d16, regs and imm8 are used for the bit assignment. Normally, the multiple registers to be saved and the amount of area to be allocated are not specified (regs, imm8) directly by the assembler. Instead, these items are specified indirectly with a pseudo instruction in the callee and ultimately resolved by the linker. For details, refer to MN10300-series cross assembler user's manual.



When the memory address (SP) is not a multiple of 4, a system exception (address misalignment exception) occurs.

CALLS

Subroutine call

CALLS (An)						
Operation	<p>An -> nPC (next instruction PC)</p> <p>PC (current instruction address) + CodeSize -> mem32(SP)</p> <p>PC (current instruction address) + CodeSize -> MDR</p> <p>This instruction branches to the specified address after saving the return address (PC (current instruction address) + CodeSize) to the stack.</p> <p>CALLS is used together with RETS in order to maintain compatibility in the case of registers to be saved and the stack area to be allocated are unclear.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
calls (An)		–	–	–	–	2
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

CALLS (d16, PC)

Operation	<p>PC (current instruction address) + (sign-ext) d16 -> nPC (next instruction PC)</p> <p>PC (current instruction address) + CodeSize -> mem32(SP)</p> <p>PC (current instruction address) + CodeSize -> MDR</p> <p>This instruction branches to the specified address after saving the return address (PC (current instruction address) + CodeSize) to the stack.</p> <p>d16 is sign-extended and added to the PC(current instruction address), and the result is written into the nPC (next instruction PC).</p> <p>Even if the addition result overflows, this overflow is ignored and the result is written into the nPC (next instruction PC).</p> <p>CALLS is used together with RETS in order to maintain compatibility in the case of registers to be saved and the stack area to be allocated are unclear.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
calls label		–	–	–	–	4
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

CALLS (d32, PC)

Operation	<p>PC (current instruction address) + (sign-ext) d32 -> nPC (next instruction PC)</p> <p>PC (current instruction address) + CodeSize -> mem32(SP)</p> <p>PC (current instruction address) + CodeSize -> MDR</p> <p>This instruction branches to the specified address after saving the return address (PC (current instruction address) + CodeSize) to the stack.</p> <p>d32 is sign-extended and added to the PC(current instruction address), and the result is written into the nPC (next instruction PC).</p> <p>Even if the addition result overflows, this overflow is ignored and the result is written into the nPC (next instruction PC).</p> <p>CALLS is used together with RETS in order to maintain compatibility in the case of registers to be saved and the stack area to be allocated are unclear.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
calls label		–	–	–	–	6
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

RET

Subroutine call

RET regs, imm8						
Operation	<p>mem32(SP + (zero-ext) imm8) -> nPC (next instruction PC)</p> <p>Lower address memory following mem32(SP + (zero-ext) imm8 - 4)</p> <p>-> Multiple registers specified by regs</p> <p>SP + (zero-ext) imm8 -> SP</p> <p>This instruction branches to the return address stored in the stack after restoring the saved multiple registers from the stack and deallocating the stack area.</p> <p>The multiple registers to be restored are specified by regs, and the area to be deallocated (number of bytes) are specified by imm8 (zero-extended).</p> <p>(Refer to MOVN for a detailed description of regs.)</p> <p>RET is used together with CALL to save and restore registers and allocate and deallocate the stack area quickly during returning from subroutine.</p> <p>When a subroutine doesn't update MDR, RETF makes returning operation from the subroutine faster.</p>					
Assembler mnemonic		V	C	N	Z	Size
ret		-	-	-	-	3
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						



The two operands of regs and imm8 are used for the bit assignment. Normally, the multiple registers to be restored and the amount of area to be deallocated are not specified (regs, imm8) directly by the assembler. Instead, these items are specified indirectly with a pseudo instruction in the callee and ultimately resolved by the linker. For details, refer to MN103-series cross assembler user's manual.



When the Mem address (SP) is not a multiple of 4, system exception (Address misalignment exception) occurs.

RETF

Return from subroutine

RETF					
Operation	<p>MDR -> nPC (next instruction PC)</p> <p>Lower address memory following $\text{mem32}(\text{SP} + (\text{zero-ext}) \text{imm8} - 4)$ -> Multiple registers specified by regs</p> <p>$\text{SP} + (\text{zero-ext}) \text{imm8} \rightarrow \text{SP}$</p> <p>This instruction branches to the return address stored in MDR after restoring the saved multiple registers to the stack and deallocating the stack area.</p> <p>The multiple registers to be restored are specified by regs, and the area to be deallocated (number of bytes) are specified by imm8 (zero-extended). (Refer to MOVN for a detailed description of regs.)</p> <p>RETF is used together with CALL in order to save and restore registers and allocate and deallocate the stack area quickly during returning from subroutine.</p> <p>When a subroutine doesn't update MDR, return operation from the subroutine cannot be assured (use RET).</p>				
Assembler mnemonic		V	C	N	Z
retf		-	-	-	-
Flag change					
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>					



The two operands of regs and imm8 are used for the bit assignment. Normally, the multiple registers to be restored and the amount of area to be deallocated are not specified (regs, imm8) directly by the assembler. Instead, these items are specified indirectly with a pseudo instruction in the callee and ultimately resolved by the linker. For details, refer to MN103-series cross assembler user's manual.

RETS

Return from subroutine

RETS						
Operation	<p>mem32(SP) -> nPC (next instruction PC)</p> <p>This branches to the return address stored in the stack.</p> <p>RETS is used together with CALLS. This is also used in order to maintain compatibility (used in RTS).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
rets		–	–	–	–	2
Flag change						
<p>VF: This is not changed.</p> <p>CF: This is not changed.</p> <p>NF: This is not changed.</p> <p>ZF: This is not changed.</p>						

RTI

Return from interrupt

RTI (Privileged instruction)						
Operation	mem32(SP) -> EPSW mem32(SP + 4) -> nPC (next instruction PC) SP + 8 -> SP This instruction returns from the interrupt, and branches to the return address stored in the stack after restoring the EPSW contained in the stack.					
Assembler mnemonic	Note	V	C	N	Z	Size
rti		Δ	Δ	Δ	Δ	2
Flag change						
VF: The V flag of the saved EPSW.V CF: The C flag of the saved EPSW.C NF: The N flag of the saved EPSW.N ZF: The Z flag of the saved EPSW.Z						



Interrupt control register TBR is set to 0x40000000 at reset.



This instruction is a privileged instruction. If this instruction is executed at user level, system exception (privileged instruction execution exception) occurs.



Access to EPSW[31:16] is possible only with EPSW at privilege level. EPSW.ML is set to 1 by restoring EPSW, when EPSW.ML=1 before restoring.

TRAP

Subroutine call to a specified address

TRAP						
Operation	<div>{ TBR[31:24], 0x000010 } -> nPC (next instruction PC)</div> <div>PC (current instruction address) + CodeSize -> mem32(SP)</div> <div>This instruction branches to the specified address (Base address set in TBR + 0x10) after saving the PC of the next instruction to the stack.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
trap		–	–	–	–	2
Flag change						
<div>VF: This is not changed.</div> <div>CF: This is not changed.</div> <div>NF: This is not changed.</div> <div>ZF: This is not changed.</div>						



Interrupt control register TBR is set to 0x40000000 at reset.



The privileged level has no change in this instruction.



The privileged level has no change in this instruction. When the privileged level is changed from the user to supervisor levels, refer to SYSCALL.

NOP

No operation

NOP									
Operation	PC + CodeSize -> PC								
	No operations are performed, and then the next instruction is executed.								
Assembler mnemonic	Note				V	C	N	Z	Size
nop					-	-	-	-	1
Flag change									
VF: This is not changed.									
CF: This is not changed.									
NF: This is not changed.									
ZF: This is not changed.									

SYSCALL

System call instruction

SYSCALL imm4						
Operation	PC (current instruction address) + CodeSize -> mem32(SP-4) EPSW -> mem32(SP-8) SP - 8 -> SP { TBR[31:24], (0x000300 + imm4 x 8) } -> nPC (next instruction PC)Starts After nPC (the next instruction address) and EPSW are stored on to stack, it branches to specified address (Base address set in TBR + 0x300 +imm4 x 8). This instruction is used as system call (OS and library calling).					
Assembler mnemonic	Note	V	C	N	Z	Size
syscall imm4		-	-	-	-	2
Flag change						
VF: This is not changed. CF: This is not changed. NF: This is not changed. ZF: This is not changed.						



Interrupt control register TBR is set to 0x40000000 at reset.



The level is changed to the supervisor level after executing this instruction.



Access to EPSW [31:16] is possible only with EPSW at the privileged level.

PI

Debug instruction

PI						
Operation	<div>PC (current instruction address) + CodeSize -> mem32(SP-4)</div> <div>EPSW -> mem32(SP-8)</div> <div>SP - 8 -> SP</div> <div>{ TBR[31:24], 0x000008 } -> nPC (next instruction PC)</div> <div>This instruction is reserved by a debugger.</div> <div>Normally, an unimplemented instruction exception occurs when this instruction is executed.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
pi		-	-	-	-	1
Flag change						
<div>VF: This is not changed.</div> <div>CF: This is not changed.</div> <div>NF: This is not changed.</div> <div>ZF: This is not changed.</div>						



Interrupt control register TBR is set to 0x40000000 at reset.



The level is changed to the supervisor level after executing this instruction.



Access to EPSW [31:16] is possible only with EPSW at the privileged level.

DMULH

Signed dual multiplication
operation

DMULH Rm, Rn						
Operation	(sign_ext)Rm[31:16] * (sign_ext)Rn[31:16] -> MDRQ (sign_ext)Rm[15:0] * (sign_ext)Rn[15:0] -> Rn This instruction multiplies the upper 16 bits of Rm (signed) by the upper 16 bits of Rn (signed), and the lower 16 bits of Rm(signed) by the lower 16bits of Rn (signed), and stores the result (32 bits) of multiplication of the upper 16 bits into MDRQ and the result (32 bits) of multiplication of the lower 16 bits into Rn.					
Assembler mnemonic	Note	V	C	N	Z	Size
dmulh Rm, Rn		-	-	-	-	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMULH Rm, Rn, Rd1, Rd2						
Operation	(sign_ext)Rm[31:16] * (sign_ext)Rn[31:16] -> Rd1 (sign_ext)Rm[15:0] * (sign_ext)Rn[15:0] -> Rd2 This instruction multiplies the upper 16 bits of Rm (signed) by the upper 16 bits of Rn (signed), and the lower 16 bits of Rm(signed) by the lower 16bits of Rn (signed), and stores the result (32 bits) of multiplication of the upper 16 bits into Rd1 and the result (32 bits) of multiplication of the lower 16 bits into Rd2.					
Assembler mnemonic	Note	V	C	N	Z	Size
dmulh Rm, Rn, Rd1, Rd2	Rd1=Rd2 can not be specified.	-	-	-	-	4
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

Extension
operation
instruction



When Rd1=Rd2 is specified, the operation result is undefined.

DMULH imm, Rn						
Operation	$(\text{sign_ext})\text{imm}[31:16] * (\text{sign_ext})\text{Rn}[31:16] \rightarrow \text{MDRQ}$ $(\text{sign_ext})\text{imm}[15:0] * (\text{sign_ext})\text{Rn}[15:0] \rightarrow \text{Rn}$ This instruction multiplies the upper 16 bits of imm (signed) by the upper 16 bits of Rn (signed), and the lower 16 bits of imm(signed) by the lower 16bits of Rn (signed), and stores the result (32 bits) of multiplication of the upper 16 bits into MDRQ and the result (32 bits) of multiplication of the lower 16 bits into Rn.					
Assembler mnemonic	Note	V	C	N	Z	Size
dmulh imm32, Rn		–	–	–	–	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMULHU Unsigned dual multiplication operation

DMULHU Rm, Rn						
Operation	(zero_ext)Rm[31:16] * (zero_ext)Rn[31:16] -> MDRQ (zero_ext)Rm[15:0] * (zero_ext)Rn[15:0] -> Rn This instruction multiplies the upper 16 bits of Rm (unsigned) by the upper 16 bits of Rn (unsigned), and the lower 16 bits of Rm(unsigned) by the lower 16bits of Rn (unsigned), and stores the result (32 bits) of multiplication of the upper 16 bits into MDRQ and the result (32 bits) of multiplication of the lower 16 bits into Rn.					
Assembler mnemonic	Note	V	C	N	Z	Size
dmulhu Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMULHU Rm, Rn, Rd1, Rd2						
Operation	(zero_ext)Rm[31:16] * (zero_ext)Rn[31:16] -> Rd1 (zero_ext)Rm[15:0] * (zero_ext)Rn[15:0] -> Rd2 This instruction multiplies the upper 16 bits of Rm (unsigned) by the upper 16 bits of Rn (unsigned), and the lower 16 bits of Rm(unsigned) by the lower 16bits of Rn (unsigned), and stores the result (32 bits) of multiplication of the upper 16 bits into Rd1 and the result (32 bits) of multiplication of the lower 16 bits into Rd2.					
Assembler mnemonic	Note	V	C	N	Z	Size
dmulhu Rm, Rn, Rd1, Rd2	Rd1=Rd2 can not be specified.	–	–	–	–	4
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						



When Rd1=Rd2 is specified, the operation result is undefined.

DMULHU imm, Rn						
Operation	(zero_ext)imm[31:16] * (zero_ext)Rn[31:16] -> MDRQ (zero_ext)imm[15:0] * (zero_ext)Rn[15:0] -> Rn This instruction multiplies the upper 16 bits of imm (unsigned) by the upper 16 bits of Rn (unsigned), and the lower 16 bits of imm(unsigned) by the lower 16bits of Rn (unsigned), and stores the result (32 bits) of multiplication of the upper 16 bits into MDRQ and the result (32 bits) of multiplication of the lower 16 bits into Rn.					
Assembler mnemonic	Note	V	C	N	Z	Size
dmulhu imm32, Rn		–	–	–	–	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMACH

Signed dual multiply-and-accumulate operation

DMACH Rm, Rn						
Operation	$(\text{sign_ext})Rm[31:16] * (\text{sign_ext})Rn[31:16]$ $+ (\text{sign_ext})Rm[15:0] * (\text{sign_ext})Rn[15:0] + MCRL \rightarrow MCRL$ <p>This instruction multiplies the upper 16 bits of Rm (signed) by the upper 16 bits of Rn (signed), and the lower 16 bits of Rm (signed) by the lower 16bits of Rn (signed). And then, it adds the result (32 bits) of multiplication of the upper 16 bits, the result (32 bits) of the lower 16 bits and the cumulative sum (32 bits) which is stored in MCRL, and stores the operation result into MCRL. When that operation result overflows, MCVF is set to 1.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmach Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMACH Rm, Rn, Rd						
Operation	$(\text{sign_ext})Rm[31:16] * (\text{sign_ext})Rn[31:16]$ $+(\text{sign_ext})Rm[15:0] * (\text{sign_ext})Rn[15:0] \& 0x0000FFFF \rightarrow Rd$ <p>This instruction multiplies the upper 16bits of Rm (signed) by the upper 16 bits of Rn (signed), and the lower 16 bits of Rm (signed) by the lower 16bits of Rn (signed). And then, it adds the result (32 bits) of multiplication of the upper 16 bits, the result (32 bits) of the lower 16 bits and the cumulative sum (32 bits) which is stored in Rd, and the operation result into Rd.</p> <p>When that operation result overflows , MCVF is set to 1.</p>					
	Assembler mnemonic	Note	V	C	N	Z
dmach Rm, Rn, Rd		Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, this is set to 1. In all other cases, it keeps the previous condition.						
CF : This is not changed.						
NF : This is not changed.						
ZF : This is not changed.						

DMACH imm, Rn

Operation	$(\text{sign_ext})\text{imm32}[31:16] * (\text{sign_ext})\text{Rn}[31:16] + (\text{sign_ext})\text{imm32}[15:0] * (\text{sign_ext})\text{Rn}[15:0] + \text{MCRL} \rightarrow \text{MCRL}$ <p>This instruction multiplies the upper 16bits of imm(signed) by the upper 16 bits of Rn (signed), and the lower 16 bits of imm (signed) by the lower 16bits of Rn (signed). And then, it adds the result (32 bits) of multiplication of the upper 16 bits, the result (32 bits) of the lower 16 bits and the cumulative sum (32 bits) which is stored in MCRL, and the operation result into MCRL.</p> <p>When that operation result overflows , MCVF is set to 1.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmach imm32, Rn		–	–	–	–	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMACHU

Unsigned dual multiply-
and-accumulate operation

DMACHU Rm, Rn						
Operation	$(\text{zero_ext})Rm[31:16] * (\text{zero_ext})Rn[31:16]$ $+ (\text{zero_ext})Rm[15:0] * (\text{zero_ext})Rn[15:0] + MCRL \rightarrow MCRL$ <p>This instruction multiplies the upper 16 bits of Rm (unsigned) by the upper 16 bits of Rn (unsigned), and the lower 16 bits of Rm (unsigned) by the lower 16 bits of Rn (unsigned). And then, it adds the result (32 bits) of multiplication of the upper 16 bits, the result (32 bits) of the lower 16 bits and the cumulative sum (32 bits) which is stored in MCRL, and stores the operation result into MCRL.</p> <p>When that operation result overflows, MCVF is set to 1.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmachu Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMACHU Rm, Rn, Rd						
Operation	$(\text{zero_ext})Rm[31:16] * (\text{zero_ext})Rn[31:16]$ $+ (\text{zero_ext})Rm[15:0] * (\text{zero_ext})Rn[15:0] + Rd \rightarrow Rd$ <p>This instruction multiplies the upper 16 bits of Rm (unsigned) by the upper 16 bits of Rn (unsigned), and the lower 16 bits of Rm (unsigned) by the lower 16 bits of Rn (unsigned). And then, it adds the result (32 bits) of multiplication of the upper 16 bits, the result (32 bits) of the lower 16 bits and the cumulative sum (32 bits) which is stored in Rd, and the operation result into Rd.</p> <p>When that operation result overflows, MCVF is set to 1.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmachu Rm, Rn, Rd		Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, this is set to 1. In all other cases, it keeps the previous condition. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

DMACHU imm, Rn

Operation	$(\text{zero_ext})\text{imm32}[31:16] * (\text{zero_ext})\text{Rn}[31:16]$ $+ (\text{zero_ext})\text{imm32}[15:0] * (\text{zero_ext})\text{Rn}[15:0] + \text{MCRL} \rightarrow \text{MCRL}$ <p>This instruction multiplies the upper 16bits of imm(unsigned) by the upper 16 bits of Rn (unsigned), and the lower 16 bits of imm (unsigned) by the lower 16bits of Rn (unsigned). And then, it adds the result (32 bits) of multiplication of the upper 16 bits, the result (32 bits) of the lower 16 bits and the cumulative sum (32 bits) which is stored in MCRL, and the operation result into MCRL.</p> <p>When that operation result overflows , MCVF is set to 1.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmachu imm32, Rn		–	–	–	–	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MAC

Signed multiply-and-accumulate
operation

MAC Rm, Rn						
Operation	<div>$(Rm * Rn) + \{MCRH, MCRL\} \rightarrow \{MCRH, MCRL\}$</div> <p>This instruction multiplies the contents of Rm (signed) and Rn (signed) , and adds the result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. And then the upper 32 bits of the operation result are stored into MCRH, and the lower 32 bits of that result are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF is set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mac Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MAC Rm, Rn, Rd1, Rd2						
Operation	<div>$(Rm * Rn) + \{Rd1, Rd2\} \rightarrow \{Rd1, Rd2\}$</div> <div>This instruction multiplies the contents of Rm (signed) and Rn (signed) , and adds the result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in Rd1 and Rd2 respectively. And then the upper 32 bits of the operation result are stored into Rd1, and the lower 32 bits of that result are stored into Rd2.</div> <div>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF is set to 1. In all other cases, it keeps the previous condition.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
mac Rm, Rn, Rd1, Rd2	Rd1=Rd2 can not be specified	Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, this is set to 1.						
CF : This is not changed.						
NF : This is not changed.						
ZF : This is not changed.						

MAC imm, Rn

Operation	$((\text{sign_ext})\text{imm} * \text{Rn}) + \{\text{MCRH}, \text{MCRL}\} \rightarrow \{\text{MCRH}, \text{MCRL}\}$ <p>This instruction multiplies the contents of the 32 bit-data which is obtained by sign-extending imm by the contents of Rn (signed), and adds the result to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits of the result are stored into MCRH and MCRL respectively. And then, the upper 32 bits of the operation result are stored into MCRH, and the lower 32 bits of that result are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF is set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mac imm8, Rn	imm8 is sign-extended.	-	-	-	-	4
mac imm24, Rn	imm24 is sign-extended.	-	-	-	-	6
mac imm32, Rn		-	-	-	-	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACU

Unsigned multiply-and-accumulate operation

MACU Rm, Rn						
Operation	<div>$(Rm * Rn) + \{MCRH, MCRL\} \rightarrow \{MCRH, MCRL\}$<p>This instruction multiplies the contents of Rm (unsigned) and Rn (unsigned) , and adds the result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. And then the upper 32 bits of the operation result are stored into MCRH, and the lower 32 bits of that result are stored into MCRL.</p><p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF is set to 1. In all other cases, it keeps the previous condition.</p></div>					
Assembler mnemonic	Note	V	C	N	Z	Size
macu Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACU Rm, Rn, Rd1, Rd2						
Operation	<div>$(Rm * Rn) + \{Rd1, Rd2\} \rightarrow \{Rd1, Rd2\}$</div> <div>This instruction multiplies the contents of Rm (unsigned) and Rn (unsigned) , and adds the result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in Rd1 and Rd2 respectively. And then, the upper 32 bits of the operation result are stored into Rd1, and the lower 32 bits of that result are stored into Rd2.</div> <div>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF is set to 1. In all other cases, it keeps the previous condition.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
macu Rm, Rn, Rd1, Rd2	Rd1=Rd2 can not be specified	Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, this is set to 1.						
CF : This is not changed.						
NF : This is not changed.						
ZF : This is not changed.						



When Rd1=Rd2 is specified, the operation result is undefined.

MACU imm, Rn

Operation	$((\text{zero_ext})\text{imm} * Rn) + \{\text{MCRH}, \text{MCRL}\} \rightarrow \{\text{MCRH}, \text{MCRL}\}$ <p>This instruction multiplies the contents of the 32 bit-data which is obtained by zero-extending imm by the contents of Rn (unsigned), and adds the result to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits of the result are stored into MCRH and MCRL respectively. And then, the upper 32 bits of the operation result are stored into MCRH, and the lower 32 bits of that result are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF is set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
macu imm8, Rn	imm8 is sign-extended.	-	-	-	-	4
macu imm24, Rn	imm24 is sign-extended.	-	-	-	-	6
macu imm32, Rn		-	-	-	-	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACH

Signed half-word multiply-and-ac-cumulate operation

MACH Rm, Rn						
Operation	<div>$((\text{sign_ext})Rm[15:0] * (\text{sign_ext})Rn[15:0]) + \{MCRH, MCRL\} \rightarrow \{MCRH, MCRL\}$</div> <div>This instruction multiplies the 32-bit data which is obtained by sign-extending the lower16 bits of Rm by the 32-bit data which is obtained by sign-extending the lower 16 bits of Rn, and adds that result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. The upper 32 bits of operation result (64 bits) are stored into MCRH and the lower 32 bits are stored into MCRL.</div> <div>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
mach Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACH Rm, Rn, Rd1, Rd2						
Operation	$((\text{sign_ext})Rm[15:0] * (\text{sign_ext})Rn[15:0]) + \{Rd1, Rd2\} \rightarrow \{Rd1, Rd2\}$ <p>This instruction multiplies the 32-bit data which is obtained by sign-extending the lower16 bits of Rm and the 32-bit data which is obtained by sign-extending the lower 16 bits of Rn, and adds that result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in Rd1 and Rd2 respectively. The upper 32 bits of operation result (64 bits) are stored into Rd1 and the lower 32 bits are stored into Rd2.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mach Rm, Rn, Rd1, Rd2	Rd1=Rd2 can not be specified	Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, this is set to 1.						
CF : This is not changed.						
NF : This is not changed.						
ZF : This is not changed.						



When Rd1=Rd2 is specified, the operation result is undefined.

MACH imm, Rn

Operation	$((\text{sign_ext})\text{imm}[15:0] * (\text{sign_ext})\text{Rn}[15:0]) + \{\text{MCRH}, \text{MCRL}\} \rightarrow \{\text{MCRH}, \text{MCRL}\}$ <p>This instruction multiplies the 16-bit data which is obtained by sign-extending imm and the 32-bit data which is obtained by sign-extending the lower 16 bits of Rn, and adds that result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. The upper 32 bits of operation result (64 bits) are stored into MCRH and the lower 32 bits are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mach imm8, Rn	imm8 is sign-extended.	-	-	-	-	4
mach imm24, Rn		-	-	-	-	6
mach imm32, Rn		-	-	-	-	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACHU

Unsigned half-word multiply-
and-accumulate operation

MACHU Rm, Rn						
Operation	<div>$((\text{zero_ext})Rm[15:0] * (\text{zero_ext})Rn[15:0]) + \{MCRH, MCRL\} \rightarrow \{MCRH, MCRL\}$</div> <div>This instruction multiplies the 32-bit data which is obtained by zero-extending the lower16 bits of Rm by the 32-bit data which is obtained by zero-extending the lower 16 bits of Rn, and adds that result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. The upper 32 bits of operation result (64 bits) are stored into MCRH and the lower 32 bits are stored into MCRL.</div> <div>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
machu Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACHU Rm, Rn, Rd1, Rd2						
Operation	$((\text{zero_ext})Rm[15:0] * (\text{zero_ext})Rn[15:0]) + \{Rd1, Rd2\} \rightarrow \{Rd1, Rd2\}$ <p>This instruction multiplies the 32-bit data which is obtained by zero-extending the lower16 bits of Rm and the 32-bit data which is obtained by zero-extending the lower 16 bits of Rn, and adds that result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in Rd1 and Rd2 respectively. The upper 32 bits of operation result (64 bits) are stored into Rd1 and the lower 32 bits are stored into Rd2.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
	Assembler mnemonic	Note	V	C	N	Z
machu Rm, Rn, Rd1, Rd2	Rd1=Rd2 can not be specified	Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, this is set to 1.						
CF : This is not changed.						
NF : This is not changed.						
ZF : This is not changed.						



When Rd1=Rd2 is specified, the operation result is undefined.

MACHU imm, Rn

Operation	$((\text{zero_ext})\text{imm}[15:0] * (\text{zero_ext})\text{Rn}[15:0]) + \{\text{MCRH} , \text{MCRL}\} \rightarrow \{\text{MCRH} , \text{MCRL}\}$ <p>This instruction multiplies the 16-bit data which is obtained by zero-extending imm and the 32-bit data which is obtained by zero-extending the lower 16 bits of Rn, and adds that result (64 bits) to the cumulative sum (64 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. The upper 32 bits of operation result (64 bits) are stored into MCRH and the lower 32 bits are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 64 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>						
Assembler mnemonic		Note	V	C	N	Z	Size
machu imm8, Rn		imm8 is sign-extended.	-	-	-	-	4
machu imm24, Rn			-	-	-	-	6
machu imm32, Rn			-	-	-	-	7
Flag change							
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.							

MACB

Signed byte multiply-and-accumulate operation

MACB Rm, Rn								
Operation	$(\text{sign_ext})Rm[7:0] * (\text{sign_ext})Rn[7:0]) + MCRL \rightarrow MCRL\}$							
	This instruction multiplies the 32-bit data which is obtained by sign-extending the lower 8 bits of Rm by the 32-bit data which is obtained by sign-extending the lower 8 bits of Rn, and adds that result (32 bits) to the cumulative sum (32 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRL respectively. The operation result (32 bits) are stored into MCRL and the lower 32 bits are stored into MCRL.							
	If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.							
Assembler mnemonic		Note		V	C	N	Z	Size
macb Rm, Rn				–	–	–	–	3
Flag change								
VF : This is not changed.								
CF : This is not changed.								
NF : This is not changed.								
ZF : This is not changed.								

MACB Rm, Rn, Rd						
Operation	$(\text{sign_ext})Rm[7:0] * (\text{sign_ext})Rn[7:0]) + Rd \rightarrow Rd$ <p>This instruction multiplies the 32-bit data which is obtained by sign-extending the lower 8 bits of Rm and the 32-bit data which is obtained by sign-extending the lower 8 bits of Rn, and adds that result (32 bits) to the cumulative sum (32 bits) stored in Rd. The operation result (32 bits) are stored into Rd.</p> <p>If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
macb Rm, Rn, Rd	Rd1=Rd2 can not be specified	Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, this is set to 1.						
CF : This is not changed.						
NF : This is not changed.						
ZF : This is not changed.						

MACB imm, Rn						
Operation	$((\text{sign_ext})\text{imm}[7:0] * (\text{sign_ext})\text{Rn}[7:0]) + \text{MCRL} \rightarrow \text{MCRL}$ <p>This instruction multiplies the 8-bit data of imm and the 32-bit data which is obtained by sign-extending the lower 8 bits of Rn, and adds that result (64 bits) to the cumulative sum (32 bits) of which the upper 32 bits and the lower 32 bits are stored in MCRH and MCRL respectively. The operation result (32 bits) are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
macb imm8, Rn		-	-	-	-	4
macb imm24, Rn		-	-	-	-	6
macb imm32, Rn		-	-	-	-	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACBU

Unsigned byte-data multiply-
and-accumulate operation

MACBU Rm, Rn						
Operation	$(\text{zero_ext})Rm[7:0] * (\text{zero_ext})Rn[7:0] + MCRL \rightarrow MCRL$ <p>This instruction multiplies the 32-bit data which is obtained by zero-extending the lower 8 bits of Rm by the 32-bit data which is obtained by zero-extending the lower 8 bits of Rn, and adds that result (32 bits) to the cumulative sum (32 bits) stored in MCRL. The operation result (32 bits) are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
macbu Rm, Rn		–	–	–	–	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACBU Rm, Rn, Rd						
Operation	$((\text{zero_ext})Rm[7:0] * (\text{zero_ext})Rn[7:0]) + Rd \rightarrow Rd$ <p>This instruction multiplies the 32-bit data which is obtained by zero-extending the lower 8 bits of Rm and the 32-bit data which is obtained by zero-extending the lower 8 bits of Rn, and adds that result (32 bits) to the cumulative sum (32 bits) stored in Rd. The operation result (32 bits) are stored into Rd.</p> <p>If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
macbu Rm, Rn, Rd	Rd1=Rd2 can not be specified	Δ	–	–	–	4
Flag change						
VF : If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, this is set to 1. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

MACBU imm, Rn

Operation	$((\text{zero_ext})\text{imm}[7:0] * (\text{zero_ext})\text{Rn}[7:0]) + \text{MCRL} \rightarrow \text{MCRL}$ <p>This instruction multiplies the 8-bit data of imm and the 32-bit data which is obtained by zero-extending the lower 8 bits of Rn, and adds that result (32 bits) to the cumulative sum (32 bits) are stored in MCRL respectively. The operation result (32 bits) are stored into MCRL.</p> <p>If the cumulative sum data overflows out of 32 bits when adding the product and the cumulative sum, MCVF are set to 1. In all other cases, it keeps the previous condition.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
macbu imm8, Rn		-	-	-	-	4
macbu imm24, Rn		-	-	-	-	6
macbu imm32, Rn		-	-	-	-	7
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

SWHW

Data swapping instruction between
the upper 2 bytes and lower 2 bytes

SWHW Rm, Rn						
Operation	<div>Rm[31:16] -> Rn[15:0] Rm[15:0] -> Rn[31:16]</div> <div>This instruction swaps the contents of the upper 16 bits and lower 16 bits of Rm, and stores the operation result (32 bits) in Rn.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
swhw Rm, Rn		–	–	–	–	3
Flag change						
<div>VF : This is not changed.</div> <div>CF : This is not changed.</div> <div>NF : This is not changed.</div> <div>ZF : This is not changed.</div>						

SWAP

Data swapping instruction between the upper and lower of the 4-byte data in byte unit

SWAP Rm, Rn						
Operation	<p> Rm[31:24] -> Rn[7:0] Rm[23:16] -> Rn[15:8] Rm[15:8] -> Rn[23:16] Rm[7:0] -> Rn[31:24] </p> <p>This instruction swaps the upper and lower 16 bits of Rm after swapping the upper 8 bits and lower 8 bits both in the upper and lower 16 bits of Rm, and then stores the result in Rn.</p> <p>Execution example: Before execution Rm=0x12345678 -> After execution Rn=0x78563412</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
swap Rm, Rn		-	-	-	-	3
Flag change						
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.						

SWAPH

Data swapping instruction between the upper and lower of the 2-byte data in byte unit

SWAPH Rm, Rn									
Operation	Rm[31:24] -> Rn[23:16]								
	Rm[23:16] -> Rn[31:24]								
	Rm[15:8] -> Rn[7:0]								
	Rm[7:0] -> Rn[15:8]								
	This instruction swaps the upper 8 bits and lower 8 bits both in the upper and lower 16 bits of Rm, and then stores the result in Rn.								
Execution example: Before execution Rm=0x12345678 -> After execution Rn=0x34127856									
Assembler mnemonic		Note			V	C	N	Z	Size
swaph Rm, Rn					-	-	-	-	3
Flag change									
VF : This is not changed.									
CF : This is not changed.									
NF : This is not changed.									
ZF : This is not changed.									

SAT16

16-bit saturation operation instruction

SAT16 Rm, Rn									
Operation	1) equal or bigger than maximum positive value (0x00007FFFF) for 16-bit signed number, 0x00007FFFF -> Rn								
	2) equal or smaller than minimum negative value (0xFFFF8000) for 16-bit signed number, 0xFFFF8000 -> Rn								
	3) other than 1) or 2), Rm -> Rn								
	1) equal or bigger than maximum positive value (0x00007FFFF) for 16-bit signed number, the maximum positive value (0x00007FFFF) is stored into Rn.								
	2) equal or smaller than minimum negative value (0xFFFF8000) for 16-bit signed number, the minimum negative value (0xFFFF8000) is stored into Rn.								
	3) other than 1) or 2), the contents of Rm are stored into Rn.								
Assembler mnemonic		Note			V	C	N	Z	Size
sat16 Rm, Rn					?	?	Δ	Δ	3
Flag change									
VF : This is not changed. CF : This is not changed. NF : "1" when the MSB of the operation result is "1", "0" in all other cases. ZF : "1" when the operation result is "0", "0" in all other cases.									

SAT24

24-bit saturation operation instruction

SAT24 Rm, Rn						
Operation	<div>1) equal or bigger than maximum positive value (0x007FFFFFFF) for 24-bit signed number, 0x007FFFFFFF -> Rn</div> <div>2) equal or smaller than minimum negative value (0xFF800000) for 24-bit signed number, 0xFF800000 -> Rn</div> <div>3) other than 1) or 2), Rm -> Rn</div> <div>1) equal or bigger than maximum positive value (0x007FFFFFFF) for 16-bit signed number, the maximum positive value (0x007FFFFFFF) is stored into Rn.</div> <div>2) equal or smaller than minimum negative value (0xFF800000) for 16-bit signed number, the minimum negative value (0xFF800000) is stored into Rn.</div> <div>3) other than 1) or 2), the contents of Rm are stored into Rn.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
sat24 Rm, Rn		?	?	Δ	Δ	4
Flag change						
<div>VF : This is not changed.</div> <div>CF : This is not changed.</div> <div>NF : "1" when the MSB of the operation result is "1", "0" in all other cases.</div> <div>ZF : "1" when the operation result is "0", "0" in all other cases.</div>						

MCSTE

Instruction of the saturation operation
for the multiply-and-accumulate result

MCSTE Rm, Rn	
Operation	<p>This instruction sets the value of the multiply-and-accumulate operation overflow, and stores the result into the V flag. In addition, depending on the value of Rm, the following instructions are performed.</p> <p>1) When the set value is 32 (0x00000020)</p> <p>When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 32-bit signed number (0x000000007FFFFFFF), the maximum positive value (0x7fffffff) is stored in Rn.</p> <p style="text-align: center;">0x7fffffff-> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 32-bit signed number (0xffffffff80000000), the maximum negative value (0x80000000) is stored in Rn.</p> <p style="text-align: center;">0x80000000 ->Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p> <p>2) When the set value is 16 (0x00000010)</p> <p>When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 16-bit signed number (0x0000000000007FFF), the maximum positive value (0x7fff) is stored in Rn.</p> <p style="text-align: center;">0x7fff -> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 16-bit signed number (0xffffffffffff8000), the maximum negative value (0xFFFF8000) is stored in Rn.</p> <p style="text-align: center;">0xffff8000 -> Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p> <p>3) When the set value is 8 (0x00000008)</p> <p>When the 32-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for an 8-bit signed number (0x0000007F), the maximum positive value (0x0000007f) is stored in Rn.</p> <p style="text-align: center;">0x0000007f -> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for an 8-bit signed number (0xffffffff80), the maximum negative value (0xfffff80) is stored in Rn.</p> <p style="text-align: center;">0xfffff80 -> Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p>

Operation	<p>4) When the set value is 9(0x00000009),</p> <p>When the 32-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 9-bit signed number (0x000000ff), the maximum positive value (0xff) is stored in Rn.</p> <p style="text-align: center;">0xff-> Rn</p> <p>If the value stored in the lower 31bits of the multiply-and-accumulate register MCRL is equal to or less than 0, 0(0x00000000) is stored in Rn.</p> <p style="text-align: center;">0x00000000 ->Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p> <p>4) When the set value is 48 (0x00000030)</p> <p>When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 48-bit signed number (0x00007fffffff), the bits from 47 to 16 are stored in Rn as the operation result.</p> <p style="text-align: center;">0x7fffffff -> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 48-bit signed number (0xffff800000000000), the bits from 47 to 16 are stored in Rn as the operation result.</p> <p style="text-align: center;">0x80000000 -> Rn</p> <p>In all other cases, the values from the bits 16 to 47 of MCRH and MCRL are stored in Rn.</p> <p style="text-align: center;">{MCRH[15:0], MCRL[31:16] -> Rn[31:0]}</p> <p>5) When the set value is other than the above-mentioned values, Rn is undefined.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mcste Rm, Rn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	3
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	3
Flag change						
<p>When multiply-and-accumulate operation overflow was not detected</p> <p>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</p> <p>CF: Always "0"</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When multiply-and-accumulate operation overflow was detected</p> <p>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</p> <p>CF: Always "0"</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

MCSTE imm, Rn

Operation	<p>This instruction sets the value of the multiply-and-accumulate operation overflow, and stores the result into the V flag. In addition, depending on the value of imm, the following instructions are performed.</p> <p>1) When the set value is 32 (0x00000020)</p> <p>When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 32-bit signed number (0x000000007FFFFFFF), the maximum positive value (0x7fffffff) is stored in Rn.</p> <p style="text-align: center;">0x7fffffff-> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 32-bit signed number (0xffffffff80000000), the maximum negative value (0x80000000) is stored in Rn.</p> <p style="text-align: center;">0x80000000 ->Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p> <p>2) When the set value is 16 (0x00000010)</p> <p>When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 16-bit signed number (0x0000000000007FFF), the maximum positive value (0x7fff) is stored in Rn.</p> <p style="text-align: center;">0x7fff -> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 16-bit signed number (0xffffffffffff8000), the maximum negative value (0xFFFF8000) is stored in Rn.</p> <p style="text-align: center;">0xffff8000 -> Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p> <p>3) When the set value is 8 (0x00000008)</p> <p>When the 32-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for an 8-bit signed number (0x0000007F), the maximum positive value (0x0000007f) is stored in Rn.</p> <p style="text-align: center;">0x0000007f -> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for an 8-bit signed number (0xffffffff80), the maximum negative value (0xffff80) is stored in Rn.</p> <p style="text-align: center;">0xffff80 -> Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p>
-----------	---

Operation	<p>4) When the set value is 9(0x00000009),</p> <p>When the 32-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 9-bit signed number (0x000000ff), the maximum positive value (0xff) is stored in Rn.</p> <p style="text-align: center;">0xff-> Rn</p> <p>If the value stored in the multiply-and-accumulate register MCRL is equal to or less than 0, the maximum negative value 0(0x00000000) is stored in Rn.</p> <p style="text-align: center;">0x00000000 ->Rn</p> <p>In all other cases, the lower 31 bits of MCRL are stored in Rn.</p> <p style="text-align: center;">MCRL -> Rn</p> <p>4) When the set value is 48 (0x00000030)</p> <p>When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 48-bit signed number (0x00007fffffffff), the bits from 47 to 16 are stored in Rn as the operation result.</p> <p style="text-align: center;">0x7fffffff -> Rn</p> <p>If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 48-bit signed number (0xffff800000000000), the bits from 47 to 16 are stored in Rn as the operation result.</p> <p style="text-align: center;">0x80000000 -> Rn</p> <p>In all other cases, the values from the bits 16 to 47 of MCRH and MCRL are stored in Rn.</p> <p style="text-align: center;">{MCRH[15:0], MCRL[31:16] -> Rn[31:0]}</p> <p>5) When the set value is other than the above-mentioned values, Rn is undefined.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
mcste imm8, Rn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	4
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	4
Flag change						
<p>When multiply-and-accumulate operation overflow was not detected</p> <p>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</p> <p>CF: Always "0"</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When multiply-and-accumulate operation overflow was detected</p> <p>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</p> <p>CF: Always "0"</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

BSCH

Bit search instruction

BSCH Rm, Rn						
Operation	<p>This instruction conducts a bit search within the 32-bit bit string stored in Rm, starting from the bit position of the bit number indicated by the contents of (Rn - 1) in the direction of descending bit numbers. The bit number of the first bit position where a “1” is found is then stored in Rn.</p> <p>When the least significant five bits of Rm are zeroes, the bit search is conducted from bit 31 in the direction of descending bit numbers.</p> <p>If the bit search reaches bit 0 without finding a “1”, the “C” flag is set, 0 is written in Rn.</p> <p>When execution of this instruction starts, the upper 27 bits of Rn are ignored.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
bsch Rm, Rn	When the search operation was succeeded ("1" is found)	?	0	?	?	3
	When the search operation was not succeeded ("1" is not found)	?	1	?	?	3
Flag change						
<p>When the search operation was succeeded</p> <p>VF: Undefined</p> <p>CF: This is "0" and indicates that the search operation is succeeded.</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When the search operation was not succeeded</p> <p>VF: Undefined</p> <p>CF: This is "1" and indicates that the search operation is not succeeded.</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

BSCH Rm, Rn, Rd

Operation	<p>This instruction conducts a bit search within the 32-bit bit string stored in Rm, starting from the bit position of the bit number indicated by the contents of (Rn - 1) in the direction of descending bit numbers. The bit number of the first bit position where a “1” is found is then stored in Rn.</p> <p>When the least significant five bits of Rm are zeroes, the bit search is conducted from bit 31 in the direction of descending bit numbers.</p> <p>If the bit search reaches bit 0 without finding a “1”, the “C” flag is set, 0 is written in Rn.</p> <p>When execution of this instruction starts, the upper 27 bits of Rn are ignored.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
bsch Rm, Rn, Rd	When the search operation was succeeded ("1" is found)	?	0	?	?	4
	When the search operation was not succeeded ("1" is not found)	?	1	?	?	4
Flag change						
<p>When the search operation was succeeded</p> <p>VF: Undefined</p> <p>CF: This is "0" and indicates that the search operation is succeeded.</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When the search operation was not succeeded</p> <p>VF: Undefined</p> <p>CF: This is "1" and indicates that the search operation is not succeeded.</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

ADD_OP2 Parallel execution of addition and OP2

ADD_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rm1 + Rn1 -> Rn1] with op2 This performs the parallel execution of the addition between the register (Rm1) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add Rm1, Rn1, Rm2, Rn2	Same as add_add of the OP1_ADD instruction	-	-	-	-	4
add_sub Rm1, Rn1, Rm2, Rn2	Same as add_sub of the OP1_SUB instruction	-	-	-	-	4
add_cmp Rm1, Rn1, Rm2, Rn2	Same as add_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
add_mov Rm1, Rn1, Rm2, Rn2	Same as add_mov of the OP1_MOV instruction	-	-	-	-	4
add_asr Rm1, Rn1, Rm2, Rn2	Same as add_asr of the OP1_ASR instruction	-	-	-	-	4
add_lsr Rm1, Rn1, Rm2, Rn2	Same as add_lsr of the OP1_LSR instruction	-	-	-	-	4
add_asl Rm1, Rn1, Rm2, Rn2	Same as add_asl of the OP1_ASX instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

ADD_OP2 Rm1, Rn1, imm, Rn2

Operation	[Rm1 + Rn1 -> Rn1] with op2 This performs the parallel execution of the addition between the register (Rm1) and the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add Rm1, Rn1, imm4, Rn2	Same as add_add of the OP1_ADD instruction	-	-	-	-	4
add_sub Rm1, Rn1, imm4, Rn2	Same as add_sub of the OP1_SUB instruction	-	-	-	-	4
add_cmp Rm1, Rn1, imm4, Rn2	Same as add_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
add_mov Rm1, Rn1, imm4, Rn2	Same as add_mov of the OP1_MOV instruction	-	-	-	-	4
add_asr Rm1, Rn1, imm4, Rn2	Same as add_asr of the OP1_ASR instruction	-	-	-	-	4
add_lsr Rm1, Rn1, imm4, Rn2	Same as add_lsr of the OP1_LSR instruction	-	-	-	-	4
add_asl Rm1, Rn1, imm4, Rn2	Same as add_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

ADD_OP2 imm, Rn1, Rm2, Rn2

Operation	<p>$[(\text{sign_ext})\text{imm4} + \text{Rn1} \rightarrow \text{Rn1}]$ with op2</p> <p>This performs the parallel execution of the addition between the sign-extended immediate value (imm4) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add imm4, Rn1, Rm2, Rn2	Same as add_add of the OP1_ADD instruction	-	-	-	-	4
add_sub imm4, Rn1, Rm2, Rn2	Same as add_sub of the OP1_SUB instruction	-	-	-	-	4
add_cmp imm4, Rn1, Rm2, Rn2	Same as add_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
add_mov imm4, Rn1, Rm2, Rn2	Same as add_mov of the OP1_MOV instruction	-	-	-	-	4
add_asr imm4, Rn1, Rm2, Rn2	Same as add_asr of the OP1_ASR instruction	-	-	-	-	4
add_lsr imm4, Rn1, Rm2, Rn2	Same as add_lsr of the OP1_LSR instruction	-	-	-	-	4
add_asl imm4, Rn1, Rm2, Rn2	Same as add_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

ADD_OP2 imm, Rn1, imm, Rn2

Operation	<p>[(sign_ext)imm4* + Rn1 -> Rn1] with op2 (* : 1st operand)</p> <p>This performs the parallel execution of the addition between the sign-extended immediate value (imm4) and the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add imm4, Rn1, imm4, Rn2	Same as add_add of the OP1_ADD instruction	-	-	-	-	4
add_sub imm4, Rn1, imm4, Rn2	Same as add_sub of the OP1_SUB instruction	-	-	-	-	4
add_cmp imm4, Rn1, imm4, Rn2	Same as add_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
add_mov imm4, Rn1, imm4, Rn2	Same as add_mov of the OP1_MOV instruction	-	-	-	-	4
add_asr imm4, Rn1, imm4, Rn2	Same as add_asr of the OP1_ASR instruction	-	-	-	-	4
add_lsr imm4, Rn1, imm4, Rn2	Same as add_lsr of the OP1_LSR instruction	-	-	-	-	4
add_asl imm4, Rn1, imm4, Rn2	Same as add_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

CMP_OP2 Parallel execution of comparison and OP2

CMP_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rn1 - Rm1] with op2 This performs the parallel execution of the comparison between the register (Rm1) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
cmp_add Rm1, Rn1, Rm2, Rn2	Same as cmp_add of the OP1_ADD instruction	Δ	Δ	Δ	Δ	4
cmp_sub Rm1, Rn1, Rm2, Rn2	Same as cmp_sub of the OP1_SUB instruction	Δ	Δ	Δ	Δ	4
cmp_mov Rm1, Rn1, Rm2, Rn2	Same as cmp_mov of the OP1_MOV instruction	Δ	Δ	Δ	Δ	4
cmp_asr Rm1, Rn1, Rm2, Rn2	Same as cmp_asr of the OP1_ASR instruction	Δ	Δ	Δ	Δ	4
cmp_lsr Rm1, Rn1, Rm2, Rn2	Same as cmp_lsr of the OP1_LSR instruction	Δ	Δ	Δ	Δ	4
cmp_asl Rm1, Rn1, Rm2, Rn2	Same as cmp_asl of the OP1_ASL instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

CMP_OP2 Rm1, Rn1, imm, Rn2						
Operation	[Rn1 - Rm1] with op2 This performs the parallel execution of the comparison between the register (Rm1) and the register (Rn1) and the op2 operation between the immediate value and the register (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
cmp_add Rm1, Rn1, imm4, Rn2	Same as cmp_add of the OP1_ADD instruction	Δ	Δ	Δ	Δ	4
cmp_sub Rm1, Rn1, imm4, Rn2	Same as cmp_sub of the OP1_SUB instruction	Δ	Δ	Δ	Δ	4
cmp_mov Rm1, Rn1, imm4, Rn2	Same as cmp_mov of the OP1_MOV instruction	Δ	Δ	Δ	Δ	4
cmp_asr Rm1, Rn1, imm4, Rn2	Same as cmp_asr of the OP1_ASR instruction	Δ	Δ	Δ	Δ	4
cmp_lsr Rm1, Rn1, imm4, Rn2	Same as cmp_lsr of the OP1_LSR instruction	Δ	Δ	Δ	Δ	4
cmp_asl Rm1, Rn1, imm4, Rn2	Same as cmp_asl of the OP1_ASL instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

CMP_OP2 imm, Rn1, Rm2, Rn2						
Operation	[Rn1 - (sign_ext)imm4] with op2 This performs the parallel execution of the comparison between the sign-extended immediate value (imm4) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
cmp_add imm4, Rn1, Rm2, Rn2	Same as cmp_add of the OP1_ADD instruction	Δ	Δ	Δ	Δ	4
cmp_sub imm4, Rn1, Rm2, Rn2	Same as cmp_sub of the OP1_SUB instruction	Δ	Δ	Δ	Δ	4
cmp_mov imm4, Rn1, Rm2, Rn2	Same as cmp_mov of the OP1_MOV instruction	Δ	Δ	Δ	Δ	4
cmp_asr imm4, Rn1, Rm2, Rn2	Same as cmp_asr of the OP1_ASR instruction	Δ	Δ	Δ	Δ	4
cmp_lsr imm4, Rn1, Rm2, Rn2	Same as cmp_lsr of the OP1_LSR instruction	Δ	Δ	Δ	Δ	4
cmp_asl imm4, Rn1, Rm2, Rn2	Same as cmp_asl of the OP1_ASL instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

CMP_OP2 imm, Rn1, imm, Rn2						
Operation	[Rn1 - (sign_ext)imm4*] with op2 (* : 1st operand) This performs the parallel execution of the comparison between the first sign-extended immediate value (imm4) and the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
cmp_add imm4, Rn1, imm4, Rn2	Same as cmp_add of the OP1_ADD instruction	Δ	Δ	Δ	Δ	4
cmp_sub imm4, Rn1, imm4, Rn2	Same as cmp_sub of the OP1_SUB instruction	Δ	Δ	Δ	Δ	4
cmp_mov imm4, Rn1, imm4, Rn2	Same as cmp_mov of the OP1_MOV instruction	Δ	Δ	Δ	Δ	4
cmp_asr imm4, Rn1, imm4, Rn2	Same as cmp_asr of the OP1_ASR instruction	Δ	Δ	Δ	Δ	4
cmp_lsr imm4, Rn1, imm4, Rn2	Same as cmp_lsr of the OP1_LSR instruction	Δ	Δ	Δ	Δ	4
cmp_asl imm4, Rn1, imm4, Rn2	Same as cmp_asl of the OP1_ASL instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

SUB_OP2

Parallel execution of subtraction and OP2

SUB_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rn1 - Rm1 -> Rn1] with op2 This performs the parallel execution of the subtraction of the register (Rm1) from the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
sub_add Rm1, Rn1, Rm2, Rn2	Same as sub_add of the OP1_ADD instruction	-	-	-	-	4
sub_sub Rm1, Rn1, Rm2, Rn2	Same as sub_sub of the OP1_SUB instruction	-	-	-	-	4
sub_cmp Rm1, Rn1, Rm2, Rn2	Same as sub_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
sub_mov Rm1, Rn1, Rm2, Rn2	Same as sub_mov of the OP1_MOV instruction	-	-	-	-	4
sub_asr Rm1, Rn1, Rm2, Rn2	Same as sub_asr of the OP1_ASR instruction	-	-	-	-	4
sub_lsr Rm1, Rn1, Rm2, Rn2	Same as sub_lsr of the OP1_LSR instruction	-	-	-	-	4
sub_asl Rm1, Rn1, Rm2, Rn2	Same as sub_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SUB_OP2 Rm1, Rn1, imm, Rn2

Operation	<p>[Rn1 - Rm1 -> Rn1] with op2</p> <p>This performs the parallel execution of the subtraction of the register (Rm1) from the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
sub_add Rm1, Rn1, imm4, Rn2	Same as sub_add of the OP1_ADD instruction	-	-	-	-	4
sub_sub Rm1, Rn1, imm4, Rn2	Same as sub_sub of the OP1_SUB instruction	-	-	-	-	4
sub_cmp Rm1, Rn1, imm4, Rn2	Same as sub_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
sub_mov Rm1, Rn1, imm4, Rn2	Same as sub_mov of the OP1_MOV instruction	-	-	-	-	4
sub_asr Rm1, Rn1, imm4, Rn2	Same as sub_asr of the OP1_ASR instruction	-	-	-	-	4
sub_lsr Rm1, Rn1, imm4, Rn2	Same as sub_lsr of the OP1_LSR instruction	-	-	-	-	4
sub_asl Rm1, Rn1, imm4, Rn2	Same as sub_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SUB_OP2 imm, Rn1, Rm2, Rn2

Operation	<p>[Rn1 - (sign_ext)imm4 -> Rn1] with op2</p> <p>This performs the parallel execution of the subtraction of the register (Rm1) from the sign-extended immediate value (imm4) and the op2 operation between the registers (Rm2 and Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
sub_add imm4, Rn1, Rm2, Rn2	Same as sub_add of the OP1_ADD instruction	-	-	-	-	4
sub_sub imm4, Rn1, Rm2, Rn2	Same as sub_sub of the OP1_SUB instruction	-	-	-	-	4
sub_cmp imm4, Rn1, Rm2, Rn2	Same as sub_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
sub_mov imm4, Rn1, Rm2, Rn2	Same as sub_mov of the OP1_MOV instruction	-	-	-	-	4
sub_asr imm4, Rn1, Rm2, Rn2	Same as sub_asr of the OP1_ASR instruction	-	-	-	-	4
sub_lsr imm4, Rn1, Rm2, Rn2	Same as sub_lsr of the OP1_LSR instruction	-	-	-	-	4
sub_asl imm4, Rn1, Rm2, Rn2	Same as sub_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SUB_OP2 imm, Rn1, imm, Rn2

Operation	<p>[Rn1 - (sign_ext)imm4* -> Rn1] with op2 (* : 1st operand)</p> <p>This performs the parallel execution of the subtraction of the register (Rn1) from the first sign-extended immediate value (imm4) and the op2 operation between the immediate value (imm4) and the register (Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
sub_add imm4, Rn1, imm4, Rn2	Same as sub_add of the OP1_ADD instruction	-	-	-	-	4
sub_sub imm4, Rn1, imm4, Rn2	Same as sub_sub of the OP1_SUB instruction	-	-	-	-	4
sub_cmp imm4, Rn1, imm4, Rn2	Same as sub_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
sub_mov imm4, Rn1, imm4, Rn2	Same as sub_mov of the OP1_MOV instruction	-	-	-	-	4
sub_asr imm4, Rn1, imm4, Rn2	Same as sub_asr of the OP1_ASR instruction	-	-	-	-	4
sub_lsr imm4, Rn1, imm4, Rn2	Same as sub_lsr of the OP1_LSR instruction	-	-	-	-	4
sub_asl imm4, Rn1, imm4, Rn2	Same as sub_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

MOV_OP2 Parallel execution of transfer and OP2

MOV_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rm1 -> Rn1] with op2 This performs the parallel execution of the transfer of the register (Rm1) to the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov_add Rm1, Rn1, Rm2, Rn2	Same as mov_add of the OP1_ADD instruction	-	-	-	-	4
mov_sub Rm1, Rn1, Rm2, Rn2	Same as mov_sub of the OP1_SUB instruction	-	-	-	-	4
mov_cmp Rm1, Rn1, Rm2, Rn2	Same as mov_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
mov_mov Rm1, Rn1, Rm2, Rn2	Same as mov_mov of the OP1_MOV instruction	-	-	-	-	4
mov_asr Rm1, Rn1, Rm2, Rn2	Same as mov_asr of the OP1_ASR instruction	-	-	-	-	4
mov_lsr Rm1, Rn1, Rm2, Rn2	Same as mov_lsr of the OP1_LSR instruction	-	-	-	-	4
mov_asl Rm1, Rn1, Rm2, Rn2	Same as mov_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

MOV_OP2 Rm1, Rn1, imm, Rn2						
Operation	[Rm1 -> Rn1] with op2 This performs the parallel execution of the transfer of the register (Rm1) to the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov_add Rm1, Rn1, imm4, Rn2	Same as mov_add of the OP1_ADD instruction	-	-	-	-	4
mov_sub Rm1, Rn1, imm4, Rn2	Same as mov_sub of the OP1_SUB instruction	-	-	-	-	4
mov_cmp Rm1, Rn1, imm4, Rn2	Same as mov_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
mov_mov Rm1, Rn1, imm4, Rn2	Same as mov_mov of the OP1_MOV instruction	-	-	-	-	4
mov_asr Rm1, Rn1, imm4, Rn2	Same as mov_asr of the OP1_ASR instruction	-	-	-	-	4
mov_lsr Rm1, Rn1, imm4, Rn2	Same as mov_lsr of the OP1_LSR instruction	-	-	-	-	4
mov_asl Rm1, Rn1, imm4, Rn2	Same as mov_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

MOV_OP2 imm, Rn1, Rm2, Rn2

Operation	[(sign-ext)imm4 -> Rn1] with op2 This performs the parallel execution of the transfer of the sign-extended immediate value (imm4) to the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov_add imm4, Rn1, Rm2, Rn2	Same as mov_add of the OP1_ADD instruction	-	-	-	-	4
mov_sub imm4, Rn1, Rm2, Rn2	Same as mov_sub of the OP1_SUB instruction	-	-	-	-	4
mov_cmp imm4, Rn1, Rm2, Rn2	Same as mov_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
mov_mov imm4, Rn1, Rm2, Rn2	Same as mov_mov of the OP1_MOV instruction	-	-	-	-	4
mov_asr imm4, Rn1, Rm2, Rn2	Same as mov_asr of the OP1_ASR instruction	-	-	-	-	4
mov_lsr imm4, Rn1, Rm2, Rn2	Same as mov_lsr of the OP1_LSR instruction	-	-	-	-	4
mov_asl imm4, Rn1, Rm2, Rn2	Same as mov_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

MOV_OP2 imm, Rn1, imm, Rn2						
Operation	[(sign_ext)imm4* -> Rn1] with op2 (* : 1st operand) This performs the parallel execution of the transfer of the first sign-extended immediate value (imm4) to the register (Rn1) and the op2 operation between the immediate value (imm) and the registers (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
mov_add imm4, Rn1, imm4, Rn2	Same as mov_add of the OP1_ADD instruction	-	-	-	-	4
mov_sub imm4, Rn1, imm4, Rn2	Same as mov_sub of the OP1_SUB instruction	-	-	-	-	4
mov_cmp imm4, Rn1, imm4, Rn2	Same as mov_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
mov_mov imm4, Rn1, imm4, Rn2	Same as mov_mov of the OP1_MOV instruction	-	-	-	-	4
mov_asr imm4, Rn1, imm4, Rn2	Same as mov_asr of the OP1_ASR instruction	-	-	-	-	4
mov_lsr imm4, Rn1, imm4, Rn2	Same as mov_lsr of the OP1_LSR instruction	-	-	-	-	4
mov_asl imm4, Rn1, imm4, Rn2	Same as mov_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

AND_OP2 Parallel execution of AND and OP2

AND_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rm1 & Rn1 -> Rn1] with op2 This performs the parallel execution of the AND between the register (Rm1) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
and_add Rm1, Rn1, Rm2, Rn2	Same as and_add of the OP1_ADD instruction	-	-	-	-	4
and_sub Rm1, Rn1, Rm2, Rn2	Same as and_sub of the OP1_SUB instruction	-	-	-	-	4
and_cmp Rm1, Rn1, Rm2, Rn2	Same as and_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
and_mov Rm1, Rn1, Rm2, Rn2	Same as and_mov of the OP1_MOV instruction	-	-	-	-	4
and_asr Rm1, Rn1, Rm2, Rn2	Same as and_asr of the OP1_ASR instruction	-	-	-	-	4
and_lsr Rm1, Rn1, Rm2, Rn2	Same as and_lsr of the OP1_LSR instruction	-	-	-	-	4
and_asl Rm1, Rn1, Rm2, Rn2	Same as and_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

AND_OP2 Rm1, Rn1, imm, Rn2

Operation	[Rm1 & Rn1 -> Rn1] with op2 This performs the parallel execution of the AND between the register (Rm1) and the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
and_add Rm1, Rn1, imm4, Rn2	Same as and_add of the OP1_ADD instruction	-	-	-	-	4
and_sub Rm1, Rn1, imm4, Rn2	Same as and_sub of the OP1_SUB instruction	-	-	-	-	4
and_cmp Rm1, Rn1, imm4, Rn2	Same as and_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
and_mov Rm1, Rn1, imm4, Rn2	Same as and_mov of the OP1_MOV instruction	-	-	-	-	4
and_asr Rm1, Rn1, imm4, Rn2	Same as and_asr of the OP1_ASR instruction	-	-	-	-	4
and_lsr Rm1, Rn1, imm4, Rn2	Same as and_lsr of the OP1_LSR instruction	-	-	-	-	4
and_asl Rm1, Rn1, imm4, Rn2	Same as and_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OR_OP2

Parallel execution of OR and
OP2

OR_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rm1 Rn1 -> Rn1] with op2 This performs the parallel execution of the OR between the register (Rm1) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
or_add Rm1, Rn1, Rm2, Rn2	Same as or_add of the OP1_ADD instruction	-	-	-	-	4
or_sub Rm1, Rn1, Rm2, Rn2	Same as or_sub of the OP1_SUB instruction	-	-	-	-	4
or_cmp Rm1, Rn1, Rm2, Rn2	Same as or_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
or_mov Rm1, Rn1, Rm2, Rn2	Same as or_mov of the OP1_MOV instruction	-	-	-	-	4
or_asr Rm1, Rn1, Rm2, Rn2	Same as or_asr of the OP1_ASR instruction	-	-	-	-	4
or_lsr Rm1, Rn1, Rm2, Rn2	Same as or_lsr of the OP1_LSR instruction	-	-	-	-	4
or_asl Rm1, Rn1, Rm2, Rn2	Same as or_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OR_OP2 Rm1, Rn1, imm, Rn2

Operation	[Rm1 Rn1 -> Rn1] with op2 This performs the parallel execution of the OR between the register (Rm1) and the register (Rn1) and the op2 operation between the immediate value (imm) and the register (Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
or_add Rm1, Rn1, imm4, Rn2	Same as or_add of the OP1_ADD instruction	-	-	-	-	4
or_sub Rm1, Rn1, imm4, Rn2	Same as or_sub of the OP1_SUB instruction	-	-	-	-	4
or_cmp Rm1, Rn1, imm4, Rn2	Same as or_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
or_mov Rm1, Rn1, imm4, Rn2	Same as or_mov of the OP1_MOV instruction	-	-	-	-	4
or_asr Rm1, Rn1, imm4, Rn2	Same as or_asr of the OP1_ASR instruction	-	-	-	-	4
or_lsr Rm1, Rn1, imm4, Rn2	Same as or_lsr of the OP1_LSR instruction	-	-	-	-	4
or_asl Rm1, Rn1, imm4, Rn2	Same as or_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

XOR_OP2 Parallel execution of XOR and OP2

XOR_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	[Rm1 ^ Rn1 -> Rn1] with op2 This performs the parallel execution of the XOR between the register (Rm1) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).					
Assembler mnemonic	Note	V	C	N	Z	Size
xor_add Rm1, Rn1, Rm2, Rn2	Same as xor_add of the OP1_ADD instruction	-	-	-	-	4
xor_sub Rm1, Rn1, Rm2, Rn2	Same as xor_sub of the OP1_SUB instruction	-	-	-	-	4
xor_cmp Rm1, Rn1, Rm2, Rn2	Same as xor_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
xor_mov Rm1, Rn1, Rm2, Rn2	Same as xor_mov of the OP1_MOV instruction	-	-	-	-	4
xor_asr Rm1, Rn1, Rm2, Rn2	Same as xor_asr of the OP1_ASR instruction	-	-	-	-	4
xor_lsr Rm1, Rn1, Rm2, Rn2	Same as xor_lsr of the OP1_LSR instruction	-	-	-	-	4
xor_asl Rm1, Rn1, Rm2, Rn2	Same as xor_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

XOR_OP2 Rm1, Rn1, imm, Rn2

Operation	<p>[Rm1 ^ Rn1 -> Rn1] with op2</p> <p>This performs the parallel execution of the XOR between the immediate value (imm) and the register (Rn1) and the op2 operation between the registers (Rm2 and Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
xor_add Rm1, Rn1, imm4, Rn2	Same as xor_add of the OP1_ADD instruction	-	-	-	-	4
xor_sub Rm1, Rn1, imm4, Rn2	Same as xor_sub of the OP1_SUB instruction	-	-	-	-	4
xor_cmp Rm1, Rn1, imm4, Rn2	Same as xor_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
xor_mov Rm1, Rn1, imm4, Rn2	Same as xor_mov of the OP1_MOV instruction	-	-	-	-	4
xor_asr Rm1, Rn1, imm4, Rn2	Same as xor_asr of the OP1_ASR instruction	-	-	-	-	4
xor_lsr Rm1, Rn1, imm4, Rn2	Same as xor_lsr of the OP1_LSR instruction	-	-	-	-	4
xor_asl Rm1, Rn1, imm4, Rn2	Same as xor_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

DMACH_OP2

Parallel execution of dual multiply-and-accumulate operation and OP2

DMACH_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	$[(\text{sign_ext}) (Rm1[31:16] * (\text{sign_ext})Rn1[31:16]) + ((\text{sign_ext})Rm1[15:0] * (\text{sign_ext})Rn1[15:0] + MCRL \rightarrow MCRL)] \text{ with op2}$ <p>This instruction adds the results of the multiplications between the upper 16 bits of the register (Rm1) and the upper 16 bits of the sign-extended register (Rn1) and between the lower 16 bits of the sign-extended register (Rm1) and the lower 16 bits of the sign-extended register (Rn1), and MCRL to each other. Then it performs the parallel execution of storing the operation result in MCRL and executing the op2 operation between the registers (Rm2 and Rn2). When that operation result overflows out of 32 bits, MCVF is set to 1 and the previous condition is kept.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmach_add Rm1, Rn1, Rm2, Rn2	Same as dmach_add of the OP1_ADD instruction	-	-	-	-	4
dmach_sub Rm1, Rn1, Rm2, Rn2	Same as dmach_sub of the OP1_SUB instruction	-	-	-	-	4
dmach_cmp Rm1, Rn1, Rm2, Rn2	Same as dmach_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
dmach_mov Rm1, Rn1, Rm2, Rn2	Same as dmach_mov of the OP1_MOV instruction	-	-	-	-	4
dmach_asr Rm1, Rn1, Rm2, Rn2	Same as dmach_asr of the OP1_ASR instruction	-	-	-	-	4
dmach_lsr Rm1, Rn1, Rm2, Rn2	Same as dmach_lsr of the OP1_LSR instruction	-	-	-	-	4
dmach_asl Rm1, Rn1, Rm2, Rn2	Same as dmach_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

DMACH_OP2 Rm1, Rn1, imm, Rn2

Operation	$[(\text{sign_ext})(\text{Rm1}[31:16]) * (\text{sign_ext})\text{Rn1}[31:16]) \\ + ((\text{sign_ext})\text{Rm1}[15:0] * (\text{sign_ext})\text{Rn1}[15:0] + \text{MCRL} \rightarrow \text{MCRL}) \text{ with op2}$ <p>This instruction adds the results of the multiplications between the upper 16 bits of the register (Rm1) and the upper 16 bits of the sign-extended register (Rn1) and between the lower 16 bits of the sign-extended register (Rm1) and the lower 16 bits of the sign-extended register (Rn1), and MCRL to each other. Then it performs the parallel execution of storing the operation result in MCRL and executing the op2 operation between the registers (imm and Rn2). When that operation result overflows out of 32 bits, MCVF is set to 1 and the previous condition is kept.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
dmach_add Rm1, Rn1, imm4, Rn2	Same as dmach_add of the OP1_ADD instruction	-	-	-	-	4
dmach_sub Rm1, Rn1, imm4, Rn2	Same as dmach_sub of the OP1_SUB instruction	-	-	-	-	4
dmach_cmp Rm1, Rn1, imm4, Rn2	Same as dmach_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
dmach_mov Rm1, Rn1, imm4, Rn2	Same as dmach_mov of the OP1_MOV instruction	-	-	-	-	4
dmach_asr Rm1, Rn1, imm4, Rn2	Same as dmach_asr of the OP1_ASR instruction	-	-	-	-	4
dmach_lsr Rm1, Rn1, imm4, Rn2	Same as dmach_lsr of the OP1_LSR instruction	-	-	-	-	4
dmach_asl Rm1, Rn1, imm4, Rn2	Same as dmach_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SWHW_OP2

Parallel execution of the data exchange between the upper 2bytes and lower 2 bytes and OP2

SWHW_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	<p>[Rm1[15:0] -> Rn1[31:16], Rm1[31: 16] -> Rn1[15:0]] with op2</p> <p>This instruction performs the parallel execution of storing the upper 16 bits of the register (Rm1) in the lower 16 bits of the register (Rn1), storing the lower 16 bits of the register (Rm1) in the upper 16 bits of the register (Rn1), and op2 operation between the registers (Rm2 and Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
swhw_add Rm1, Rn1, Rm2, Rn2	Same as swhw_add of the OP1_ADD instruction	-	-	-	-	4
swhw_sub Rm1, Rn1, Rm2, Rn2	Same as swhw_sub of the OP1_SUB instruction	-	-	-	-	4
swhw_cmp Rm1, Rn1, Rm2, Rn2	Same as swhw_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
swhw_mov Rm1, Rn1, Rm2, Rn2	Same as swhw_mov of the OP1_MOV instruction	-	-	-	-	4
swhw_asr Rm1, Rn1, Rm2, Rn2	Same as swhw_asr of the OP1_ASR instruction	-	-	-	-	4
swhw_lsr Rm1, Rn1, Rm2, Rn2	Same as swhw_lsr of the OP1_LSR instruction	-	-	-	-	4
swhw_asl Rm1, Rn1, Rm2, Rn2	Same as swhw_asl of the OP1_ASX instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all ther cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SWHW_OP2 Rm1, Rn1, imm, Rn2

Operation	<p>[Rm1[15:0] -> Rn1[31:16], Rm1[31: 16] -> Rn1[15:0]] with op2</p> <p>This instruction performs the parallel execution of storing the upper 16 bits of the register (Rm1) in the lower 16 bits of the register (Rn1), storing the lower 16 bits of the register (Rm1) in the upper 16 bits of the register (Rn1), and op2 operation between the immediate value (imm) and the register (Rn2).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
swhw_add Rm1, Rn1, imm4, Rn2	Same as swhw_add of the OP1_ADD instruction	-	-	-	-	4
swhw_sub Rm1, Rn1, imm4, Rn2	Same as swhw_sub of the OP1_SUB instruction	-	-	-	-	4
swhw_cmp Rm1, Rn1, imm4, Rn2	Same as swhw_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
swhw_mov Rm1, Rn1, imm4, Rn2	Same as swhw_mov of the OP1_MOV instruction	-	-	-	-	4
swhw_asr Rm1, Rn1, imm4, Rn2	Same as swhw_asr of the OP1_ASR instruction	-	-	-	-	4
swhw_lsr Rm1, Rn1, imm4, Rn2	Same as swhw_lsr of the OP1_LSR instruction	-	-	-	-	4
swhw_asl Rm1, Rn1, imm4, Rn2	Same as swhw_asl of the OP1_ASL instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all ther cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SAT16_OP2

Parallel execution of the saturation operation of the upper 16 bits and OP2

SAT16_OP2 Rm1, Rn1, Rm2, Rn2						
Operation	<p>[(SAT16 op.) Rm1 -> Rn1] with op2</p> <p>When source register Rm1 is</p> <p>1) equal or bigger than maximum positive value (0x00007FFF) for 16-bit signed number, the maximum positive value (0x00007FFF) is stored into the register (Rn1).</p> <p>2) equal or smaller than minimum negative value (0xFFFF8000) for 16-bit signed number, the minimum negative value (0xFFFF8000) is stored into the register (Rn1).</p> <p>3) other than 1) or 2), the contents of the register (Rm1) are stored into the register (Rn1).</p> <p>Parallel execution is carried out for op2 operations of SAT16 and Rm2, Rn2.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
sat16_add Rm1, Rn1, Rm2, Rn2	Same as sat16_add of the OP1_ADD instruction	-	-	-	-	4
sat16_sub Rm1, Rn1, Rm2, Rn2	Same as sat16_sub of the OP1_SUB instruction	-	-	-	-	4
sat16_cmp Rm1, Rn1, Rm2, Rn2	Same as sat16_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
sat16_mov Rm1, Rn1, Rm2, Rn2	Same as sat16_mov of the OP1_MOV instruction	-	-	-	-	4
sat16_asr Rm1, Rn1, Rm2, Rn2	Same as sat16_asr of the OP1_ASR instruction	-	-	-	-	4
sat16_lsr Rm1, Rn1, Rm2, Rn2	Same as sat16_lsr of the OP1_LSR instruction	-	-	-	-	4
sat16_asl Rm1, Rn1, Rm2, Rn2	Same as sat16_asl of the OP1_ASX instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

SAT16_OP2 Rm1, Rn1, imm, Rn2

Operation	[(SAT16 op.) Rm1 -> Rn1] with op2 When source register Rm1 is 1) equal or bigger than maximum positive value (0x00007FFF) for 16-bit signed number, the maximum positive value (0x00007FFF) is stored into the register (Rn1). 2) equal or smaller than minimum negative value (0xFFFF8000) for 16-bit signed number, the minimum negative value (0xFFFF8000) is stored into the register (Rn1). 3) other than 1) or 2), the contents of the register (Rm1) are stored into the register (Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
sat16_add Rm1, Rn1, imm4, Rn2	Same as sat16_add of the OP1_ADD instruction	-	-	-	-	4
sat16_sub Rm1, Rn1, imm4, Rn2	Same as sat16_sub of the OP1_SUB instruction	-	-	-	-	4
sat16_cmp Rm1, Rn1, imm4, Rn2	Same as sat16_cmp of the OP1_CMP instruction	Δ	Δ	Δ	Δ	4
sat16_mov Rm1, Rn1, imm4, Rn2	Same as sat16_mov of the OP1_MOV instruction	-	-	-	-	4
sat16_asr Rm1, Rn1, imm4, Rn2	Same as sat16_asr of the OP1_ASR instruction	-	-	-	-	4
sat16_lsr Rm1, Rn1, imm4, Rn2	Same as sat16_lsr of the OP1_LSR instruction	-	-	-	-	4
sat16_asl Rm1, Rn1, imm4, Rn2	Same as sat16_asl of the OP1_ASX instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP2 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ADD

Parallel execution of OP1 and addition

OP1_ADD Rm1, Rn1, Rm2, Rn2						
Operation	[Rm2 +Rn -> Rn2] with op1 This performs the parallel execution of the addition between the registers (Rm2 and Rn2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add Rm1, Rn1, Rm2, Rn2	Same as add_add of the ADD_OP2 instruction	-	-	-	-	4
cmp_add Rm1, Rn1, Rm2, Rn2	Same as cmp_add of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_add Rm1, Rn1, Rm2, Rn2	Same as sub_add of the SUB_OP2 instruction	-	-	-	-	4
mov_add Rm1, Rn1, Rm2, Rn2	Same as mov_add of the MOV_OP2 instruction	-	-	-	-	4
and_add Rm1, Rn1, Rm2, Rn2	Same as and_add of the AND_OP2 instruction	-	-	-	-	4
or_add Rm1, Rn1, Rm2, Rn2	Same as or_add of the OR_OP2 instruction	-	-	-	-	4
xor_add Rm1, Rn1, Rm2, Rn2	Same as xor_add of the XOR_OP2 instruction	-	-	-	-	4
dmach_add Rm1, Rn1, Rm2, Rn2	Same as dmach_add of the DMACH_OP2 instruction	-	-	-	-	4
swhw_add Rm1, Rn1, Rm2, Rn2	Same as swhw_add of the SWHW_OP2 instruction	-	-	-	-	4
sat16_add Rm1, Rn1, Rm2, Rn2	Same as sat16_add of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
When OP1 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ADD imm, Rn1, Rm2, Rn2						
Operation	[Rm2 +Rn -> Rn2] with op1 This performs the paralell execution of the addition between the registers (Rm2 and Rn2) and the op1 operation between the immediate value (imm) and the register (Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add imm4, Rn1, Rm2, Rn2	Same as add_add of the ADD_OP2 instruction	-	-	-	-	4
cmp_add imm4, Rn1, Rm2, Rn2	Same as cmp_add of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_add imm4, Rn1, Rm2, Rn2	Same as sub_add of the SUB_OP2 instruction	-	-	-	-	4
mov_add imm4, Rn1, Rm2, Rn2	Same as mov_add of the MOV_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all ther cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ADD Rm1, Rn1, imm, Rn2

Operation	[(sign_ext) imm4+Rn2 -> Rn2] with op1 This performs the parallel execution of the addition between the sign-extended immediate value (imm4) the register (Rn2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add Rm1, Rn1, imm4, Rn2	Same as add_add of the ADD_OP2 instruction	-	-	-	-	4
cmp_add Rm1, Rn1, imm4, Rn2	Same as cmp_add of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_add Rm1, Rn1, imm4, Rn2	Same as sub_add of the SUB_OP2 instruction	-	-	-	-	4
mov_add Rm1, Rn1, imm4, Rn2	Same as mov_add of the MOV_OP2 instruction	-	-	-	-	4
and_add Rm1, Rn1, imm4, Rn2	Same as and_add of the AND_OP2 instruction	-	-	-	-	4
or_add Rm1, Rn1, imm4, Rn2	Same as or_add of the OR_OP2 instruction	-	-	-	-	4
xor_add Rm1, Rn1, imm4, Rn2	Same as xor_add of the XOR_OP2 instruction	-	-	-	-	4
dmach_add Rm1,Rn1,imm4,Rn2	Same as dmach_add of the DMACH_OP2 instruction	-	-	-	-	4
swhw_add Rm1, Rn1, imm4, Rn2	Same as swhw_add of the SWHW_OP2 instruction	-	-	-	-	4
sat16_add Rm1, Rn1, imm4, Rn2	Same as sat16_add of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
When OP2 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ADD imm, Rn1, imm, Rn2						
Operation	[(sign_ext) imm4* + Rn2 -> Rn2] with op1 (* : 3rd operand) This performs the parallel execution of the addition between the sign-extended first immediate value (imm4) the register (Rn2) and the op1 operation between the immediate value (imm) and the register (Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_add imm4, Rn1, imm4, Rn2	Same as add_add of the ADD_OP2 instruction	-	-	-	-	4
cmp_add imm4, Rn1, imm4, Rn2	Same as cmp_add of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_add imm4, Rn1, imm4, Rn2	Same as sub_add of the SUB_OP2 instruction	-	-	-	-	4
mov_add imm4, Rn1, imm4, Rn2	Same as mov_add of the MOV_OP2 instruction	-	-	-	-	4
Flag change						
When OP1 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_SUB

Parallel execution of OP1 and subtraction

OP1_SUB Rm1, Rn1, Rm2, Rn2						
Operation	<p>[Rn2 - Rm2 -> Rn2] with op1</p> <p>This performs the parallel execution of the subtraction of the register (Rm2) from the register (Rn2) and the op1 operation between the registers (Rm1 and Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_sub Rm1, Rn1, Rm2, Rn2	Same as add_sub of the ADD_OP2 instruction	-	-	-	-	4
cmp_sub Rm1, Rn1, Rm2, Rn2	Same as cmp_sub of the CMP_OP2 instruction	-	-	-	-	4
sub_sub Rm1, Rn1, Rm2, Rn2	Same as sub_sub of the SUB_OP2 instruction	Δ	Δ	Δ	Δ	4
mov_sub Rm1, Rn1, Rm2, Rn2	Same as mov_sub of the MOV_OP2 instruction	-	-	-	-	4
and_sub Rm1, Rn1, Rm2, Rn2	Same as and_sub of the AND_OP2 instruction	-	-	-	-	4
or_sub Rm1, Rn1, Rm2, Rn2	Same as or_sub of the OR_OP2 instruction	-	-	-	-	4
xor_sub Rm1, Rn1, Rm2, Rn2	Same as xor_sub of the XOR_OP2 instruction	-	-	-	-	4
dmach_sub Rm1, Rn1, Rm2, Rn2	Same as dmach_sub of the DMACH_OP2 instruction	-	-	-	-	4
swhw_sub Rm1, Rn1, Rm2, Rn2	Same as swhw_sub of the SWHW_OP2 instruction	-	-	-	-	4
sat16_sub Rm1, Rn1, Rm2, Rn2	Same as sat16_sub of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_SUB imm, Rn1, Rm2, Rn2

Operation	<p>[Rn2 - Rm2 -> Rn2] with op1</p> <p>This performs the parallel execution of the subtraction of the register (Rm2) from the register (Rn2) and the op1 operation between the immediate value and the register (Rn1).</p>
-----------	--

Assembler mnemonic	Note	V	C	N	Z	Size
add_sub imm4, Rn1, Rm2, Rn2	Same as add_sub of the ADD_OP2 instruction	-	-	-	-	4
cmp_sub imm4, Rn1, Rm2, Rn2	Same as cmp_sub of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_sub imm4, Rn1, Rm2, Rn2	Same as sub_sub of the SUB_OP2 instruction	-	-	-	-	4
mov_sub imm4, Rn1, Rm2, Rn2	Same as mov_sub of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP2 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_SUB Rm1, Rn1, imm, Rn2

Operation	<p>[Rn2 - (sign_ext)imm4 -> Rn2] with op1</p> <p>This performs the parallel execution of the subtraction of the register (Rn2) from the sign-extended immediate value (imm4) and the op1 operation between the registers (Rm1 and Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_sub Rm1, Rn1, imm4, Rn2	Same as add_sub of the ADD_OP2 instruction	-	-	-	-	4
cmp_sub Rm1, Rn1, imm4, Rn2	Same as cmp_sub of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_sub Rm1, Rn1, imm4, Rn2	Same as sub_sub of the SUB_OP2 instruction	-	-	-	-	4
mov_sub Rm1, Rn1, imm4, Rn2	Same as mov_sub of the MOV_OP2 instruction	-	-	-	-	4
and_sub Rm1, Rn1, imm4, Rn2	Same as and_sub of the AND_OP2 instruction	-	-	-	-	4
or_sub Rm1, Rn1, imm4, Rn2	Same as or_sub of the OR_OP2 instruction	-	-	-	-	4
xor_sub Rm1, Rn1, imm4, Rn2	Same as xor_sub of the XOR_OP2 instruction	-	-	-	-	4
dmach_sub Rm1, Rn1, imm4, Rn2	Same as dmach_sub of the DMACH_OP2 instruction	-	-	-	-	4
swhw_sub Rm1, Rn1, imm4, Rn2	Same as swhw_sub of the SWHW_OP2 instruction	-	-	-	-	4
sat16_sub Rm1, Rn1, imm4, Rn2	Same as sat16_sub of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_SUB imm, Rn1, imm, Rn2

Operation	<p>[Rn2 - (sign_ext)imm4* -> Rn2] with op1 (* : 3rd operand)</p> <p>This performs the parallel execution of the subtraction of the register (Rn2) from the sign-extended first immediate value (imm4) and the op1 operation between the immediate value and the register (Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_sub imm4, Rn1, imm4, Rn2	Same as add_sub of the ADD_OP2 instruction	-	-	-	-	4
cmp_sub imm4, Rn1, imm4, Rn2	Same as cmp_sub of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_sub imm4, Rn1, imm4, Rn2	Same as sub_sub of the SUB_OP2 instruction	-	-	-	-	4
mov_sub imm4, Rn1, imm4, Rn2	Same as mov_sub of the MOV_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP2 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP2 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_CMP

Parallel execution of comparison and OP1

OP1_CMP Rm1, Rn1, Rm2, Rn2						
Operation	[Rn2 - Rm2 : EPSW] with op1 This performs the parallel execution of the comparison between the registers (Rm2 and Rn2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_cmp Rm1, Rn1, Rm2, Rn2	Same as add_cmp of the ADD_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_cmp Rm1, Rn1, Rm2, Rn2	Same as sub_cmp of the SUB_OP2 instruction	Δ	Δ	Δ	Δ	4
mov_cmp Rm1, Rn1, Rm2, Rn2	Same as mov_cmp of the MOV_OP2 instruction	Δ	Δ	Δ	Δ	4
and_cmp Rm1, Rn1, Rm2, Rn2	Same as and_cmp of the AND_OP2 instruction	Δ	Δ	Δ	Δ	4
or_cmp Rm1, Rn1, Rm2, Rn2	Same as or_cmp of the OR_OP2 instruction	Δ	Δ	Δ	Δ	4
xor_cmp Rm1, Rn1, Rm2, Rn2	Same as xor_cmp of the XOR_OP2 instruction	Δ	Δ	Δ	Δ	4
dmach_cmp Rm1, Rn1, Rm2, Rn2	Same as dmach_cmp of the DMACH_OP2 instruction	Δ	Δ	Δ	Δ	4
swhw_cmp Rm1, Rn1, Rm2, Rn2	Same as swhw_cmp of the SWHW_OP2 instruction	Δ	Δ	Δ	Δ	4
sat16_cmp Rm1, Rn1, Rm2, Rn2	Same as sat16_cmp of the SAT16_OP2 instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

OP1_CMP imm, Rn1, Rm2, Rn2						
Operation	[Rn2 - Rm2 : EPSW] with op1 This performs the parallel execution of the comparison between the registers (Rm2 and Rn2) and the op1 operation between the immediate value (imm) and the register (Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_cmp imm4, Rn1, Rm2, Rn2	Same as add_cmp of the ADD_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_cmp imm4, Rn1, Rm2, Rn2	Same as sub_cmp of the SUB_OP2 instruction	Δ	Δ	Δ	Δ	4
mov_cmp imm4, Rn1, Rm2, Rn2	Same as mov_cmp of the MOV_OP2 instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

OP1_CMP Rm1, Rn1, imm, Rn2						
Operation	[Rn2 - (sign_ext)imm4* : EPSW] with op1 (* : 3rd operand) This performs the parallel execution of the comparison between the sign-extended immediate value (imm4) and the register (Rn2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_cmp Rm1, Rn1, imm4, Rn2	Same as add_cmp of the ADD_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_cmp Rm1, Rn1, imm4, Rn2	Same as sub_cmp of the SUB_OP2 instruction	Δ	Δ	Δ	Δ	4
mov_cmp Rm1, Rn1, imm4, Rn2	Same as mov_cmp of the MOV_OP2 instruction	Δ	Δ	Δ	Δ	4
and_cmp Rm1, Rn1, imm4, Rn2	Same as and_cmp of the AND_OP2 instruction	Δ	Δ	Δ	Δ	4
or_cmp Rm1, Rn1, imm4, Rn2	Same as or_cmp of the OR_OP2 instruction	Δ	Δ	Δ	Δ	4
xor_cmp Rm1, Rn1, imm4, Rn2	Same as xor_cmp of the XOR_OP2 instruction	Δ	Δ	Δ	Δ	4
dmach_cmp Rm1, Rn1, imm4, Rn2	Same as dmach_cmp of the DMACH_OP2 instruction	Δ	Δ	Δ	Δ	4
swhw_cmp Rm1, Rn1, imm4, Rn2	Same as swhw_cmp of the SWHW_OP2 instruction	Δ	Δ	Δ	Δ	4
sat16_cmp Rm1, Rn1, imm4, Rn2	Same as sat16_cmp of the SAT16_OP2 instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

OP1_CMP imm, Rn1, imm, Rn2						
Operation	[Rn2 - (sign_ext)imm4* : EPSW] with op1 (* : 3rd operand) This performs the parallel execution of the comparison between the registers (Rm2 and Rn2) and the op1 operation between the immediate value (imm) and the register (Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_cmp imm4, Rn1, imm4, Rn2	Same as add_cmp of the ADD_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_cmp imm4, Rn1, imm4, Rn2	Same as sub_cmp of the SUB_OP2 instruction	Δ	Δ	Δ	Δ	4
mov_cmp imm4, Rn1, imm4, Rn2	Same as mov_cmp of the MOV_OP2 instruction	Δ	Δ	Δ	Δ	4
Flag change						
VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						

OP1_MOV Parallel execution of transfer and OP1

OP1_MOV Rm1, Rn1, Rm2, Rn2

Operation	<p>[Rm2 -> Rn2] with op1</p> <p>This performs the parallel execution of the transfer of the register (Rm2) to the register (Rn2) and the op1 operation between the registers (Rm1 and Rn1).</p>
-----------	--

Assembler mnemonic	Note	V	C	N	Z	Size
add_mov Rm1, Rn1, Rm2, Rn2	Same as add_mov of the ADD_OP2 instruction	-	-	-	-	4
cmp_mov Rm1, Rn1, Rm2, Rn2	Same as cmp_mov of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_mov Rm1, Rn1, Rm2, Rn2	Same as sub_mov of the SUB_OP2 instruction	-	-	-	-	4
mov_mov Rm1, Rn1, Rm2, Rn2	Same as mov_mov of the MOV_OP2 instruction	-	-	-	-	4
and_mov Rm1, Rn1, Rm2, Rn2	Same as and_mov of the AND_OP2 instruction	-	-	-	-	4
or_mov Rm1, Rn1, Rm2, Rn2	Same as or_mov of the OR_OP2 instruction	-	-	-	-	4
xor_mov Rm1, Rn1, Rm2, Rn2	Same as xor_mov of the XOR_OP2 instruction	-	-	-	-	4
dmach_mov Rm1, Rn1, Rm2, Rn2	Same as dmach_mov of the DMACH_OP2 instruction	-	-	-	-	4
swhw_mov Rm1, Rn1, Rm2, Rn2	Same as swhw_mov of the SWHW_OP2 instruction	-	-	-	-	4
sat16_mov Rm1, Rn1, Rm2, Rn2	Same as sat16_mov of the SAT16_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_MOV imm, Rn1, Rm2, Rn2

Operation	<p>[Rm2 -> Rn2] with op1</p> <p>This performs the parallel execution of the transfer of the register (Rm2) to the register (Rn2) and the op1 operation between the registers (Rm1 and Rn1).</p>
-----------	--

Assembler mnemonic	Note	V	C	N	Z	Size
add_mov imm4, Rn1, Rm2, Rn2	Same as add_mov of the ADD_OP2 instruction	-	-	-	-	4
cmp_mov imm4, Rn1, Rm2, Rn2	Same as cmp_mov of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_mov imm4, Rn1, Rm2, Rn2	Same as sub_mov of the SUB_OP2 instruction	-	-	-	-	4
mov_mov imm4, Rn1, Rm2, Rn2	Same as mov_mov of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_MOV Rm1, Rn1, imm, Rn2

Operation	[(sign_ext)imm4* -> Rn2] with op1 (* : 3rd operand) This performs the parallel execution of the transfer of the sign-extended immediate value (imm4) to the register (Rn2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_mov Rm1, Rn1, imm4, Rn2	Same as add_mov of the ADD_OP2 instruction	-	-	-	-	4
cmp_mov Rm1, Rn1, imm4, Rn2	Same as cmp_mov of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_mov Rm1, Rn1, imm4, Rn2	Same as sub_mov of the SUB_OP2 instruction	-	-	-	-	4
mov_mov Rm1, Rn1, imm4, Rn2	Same as mov_mov of the MOV_OP2 instruction	-	-	-	-	4
and_mov Rm1, Rn1, imm4, Rn2	Same as and_mov of the AND_OP2 instruction	-	-	-	-	4
or_mov Rm1, Rn1, imm4, Rn2	Same as or_mov of the OR_OP2 instruction	-	-	-	-	4
xor_mov Rm1, Rn1, imm4, Rn2	Same as xor_mov of the XOR_OP2 instruction	-	-	-	-	4
dmach_mov Rm1, Rn1, imm4, Rn2	Same as dmach_mov of the DMACH_OP2 instruction	-	-	-	-	4
swhw_mov Rm1, Rn1, imm4, Rn2	Same as swhw_mov of the SWHW_OP2 instruction	-	-	-	-	4
sat16_mov Rm1, Rn1, imm4, Rn2	Same as sat16_mov of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
When OP1 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_MOV imm, Rn1, imm, Rn2

Operation	[(sign_ext)imm4* -> Rn2] with op1 (* : 3rd operand) This performs the parallel execution of the transfer of the sign-extended third-operand immediate value (imm4) to the register (Rn2) and the op1 operation between the immediate value (imm) and the register (Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_mov imm4, Rn1, imm4, Rn2	Same as add_mov of the ADD_OP2 instruction	-	-	-	-	4
cmp_mov imm4, Rn1, imm4, Rn2	Same as cmp_mov of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_mov imm4, Rn1, imm4, Rn2	Same as sub_mov of the SUB_OP2 instruction	-	-	-	-	4
mov_mov imm4, Rn1, imm4, Rn2	Same as mov_mov of the MOV_OP2 instruction	-	-	-	-	4
Flag change						
When OP1 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASR

Parallel execution of arithmetic shift right and OP1

OP1_ASR Rm1, Rn1, Rm2, Rn2

Operation	<p>[(Rn2 >> Rm2) -> Rn2] with op1</p> <p>This performs the parallel execution of the arithmetic right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_asr Rm1, Rn1, Rm2, Rn2	Same as add_asr of the ADD_OP2 instruction	-	-	-	-	4
cmp_asr Rm1, Rn1, Rm2, Rn2	Same as cmp_asr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asr Rm1, Rn1, Rm2, Rn2	Same as sub_asr of the SUB_OP2 instruction	-	-	-	-	4
mov_asr Rm1, Rn1, Rm2, Rn2	Same as mov_asr of the MOV_OP2 instruction	-	-	-	-	4
and_asr Rm1, Rn1, Rm2, Rn2	Same as and_asr of the AND_OP2 instruction	-	-	-	-	4
or_asr Rm1, Rn1, Rm2, Rn2	Same as or_asr of the OR_OP2 instruction	-	-	-	-	4
xor_asr Rm1, Rn1, Rm2, Rn2	Same as xor_asr of the XOR_OP2 instruction	-	-	-	-	4
dmach_asr Rm1, Rn1, Rm2, Rn2	Same as dmach_asr of the DMACH_OP2 instruction	-	-	-	-	4
swhw_asr Rm1, Rn1, Rm2, Rn2	Same as swhw_asr of the SWHW_OP2 instruction	-	-	-	-	4
sat16_asr Rm1, Rn1, Rm2, Rn2	Same as sat16_asr of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASR imm, Rn1, Rm2, Rn2

Operation	<p>$[(Rn2 \gg Rm2) \rightarrow Rn2]$ with op1</p> <p>This performs the parallel execution of the arithmetic right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>
-----------	---

Assembler mnemonic	Note	V	C	N	Z	Size
add_asr imm4, Rn1, Rm2, Rn2	Same as add_asr of the ADD_OP2 instruction	-	-	-	-	4
cmp_asr imm4, Rn1, Rm2, Rn2	Same as cmp_asr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asr imm4, Rn1, Rm2, Rn2	Same as sub_asr of the SUB_OP2 instruction	-	-	-	-	4
mov_asr imm4, Rn1, Rm2, Rn2	Same as mov_asr of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASR Rm1, Rn1, imm, Rn2

Operation	[(sign_ext)imm4* -> Rn2] with op1 (* : 3rd operand) This performs the parallel execution of the arithmetic right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_asr Rm1, Rn1, imm4, Rn2	Same as add_asr of the ADD_OP2 instruction	-	-	-	-	4
cmp_asr Rm1, Rn1, imm4, Rn2	Same as cmp_asr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asr Rm1, Rn1, imm4, Rn2	Same as sub_asr of the SUB_OP2 instruction	-	-	-	-	4
mov_asr Rm1, Rn1, imm4, Rn2	Same as mov_asr of the MOV_OP2 instruction	-	-	-	-	4
and_asr Rm1, Rn1, imm4, Rn2	Same as and_asr of the AND_OP2 instruction	-	-	-	-	4
or_asr Rm1, Rn1, imm4, Rn2	Same as or_asr of the OR_OP2 instruction	-	-	-	-	4
xor_asr Rm1, Rn1, imm4, Rn2	Same as xor_asr of the XOR_OP2 instruction	-	-	-	-	4
dmach_asr Rm1, Rn1, imm4, Rn2	Same as dmach_asr of the DMACH_OP2 instruction	-	-	-	-	4
swhw_asr Rm1, Rn1, imm4, Rn2	Same as swhw_asr of the SWHW_OP2 instruction	-	-	-	-	4
sat16_asr Rm1, Rn1, imm4, Rn2	Same as sat16_asr of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
When OP1 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASR imm, Rn1, imm, Rn2

Operation	<p>$[(Rn2 \gg (\text{sign_ext})imm4* \rightarrow Rn2)]$ with op1 (* : 3rd operand)</p> <p>This performs the parallel execution of the arithmetic right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>
-----------	---

Assembler mnemonic	Note	V	C	N	Z	Size
add_asr imm4, Rn1, imm4, Rn2	Same as add_asr of the ADD_OP2 instruction	-	-	-	-	4
cmp_asr imm4, Rn1, imm4, Rn2	Same as cmp_asr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asr imm4, Rn1, imm4, Rn2	Same as sub_asr of the SUB_OP2 instruction	-	-	-	-	4
mov_asr imm4, Rn1, imm4, Rn2	Same as mov_asr of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_LSR

Parallel execution of logical
shift right and OP1

OP1_LSR Rm1, Rn1, Rm2, Rn2						
Operation	<p>[(Rn2 >> Rm2) -> Rn2] with op1</p> <p>This performs the parallel execution of the logical right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_lsr Rm1, Rn1, Rm2, Rn2	Same as add_lsr of the ADD_OP2 instruction	-	-	-	-	4
cmp_lsr Rm1, Rn1, Rm2, Rn2	Same as cmp_lsr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_lsr Rm1, Rn1, Rm2, Rn2	Same as sub_lsr of the SUB_OP2 instruction	-	-	-	-	4
mov_lsr Rm1, Rn1, Rm2, Rn2	Same as mov_lsr of the MOV_OP2 instruction	-	-	-	-	4
and_lsr Rm1, Rn1, Rm2, Rn2	Same as and_lsr of the AND_OP2 instruction	-	-	-	-	4
or_lsr Rm1, Rn1, Rm2, Rn2	Same as or_lsr of the OR_OP2 instruction	-	-	-	-	4
xor_lsr Rm1, Rn1, Rm2, Rn2	Same as xor_lsr of the XOR_OP2 instruction	-	-	-	-	4
dmach_lsr Rm1, Rn1, Rm2, Rn2	Same as dmach_lsr of the DMACH_OP2 instruction	-	-	-	-	4
swhw_lsr Rm1, Rn1, Rm2, Rn2	Same as swhw_lsr of the SWHW_OP2 instruction	-	-	-	-	4
sat16_lsr Rm1, Rn1, Rm2, Rn2	Same as sat16_lsr of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_LSR imm, Rn1, Rm2, Rn2

Operation	<p>$[(Rn2 \gg Rm2) \rightarrow Rn2]$ with op1</p> <p>This performs the parallel execution of the logical right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>
-----------	--

Assembler mnemonic	Note	V	C	N	Z	Size
add_lsr imm4, Rn1, Rm2, Rn2	Same as add_lsr of the ADD_OP2 instruction	-	-	-	-	4
cmp_lsr imm4, Rn1, Rm2, Rn2	Same as cmp_lsr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_lsr imm4, Rn1, Rm2, Rn2	Same as sub_lsr of the SUB_OP2 instruction	-	-	-	-	4
mov_lsr imm4, Rn1, Rm2, Rn2	Same as mov_lsr of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_LSR Rm1, Rn1, imm, Rn2

Operation	$[(Rn2 \gg (zero_ext)imm4*) \rightarrow Rn2]$ with op1 (*: 3rd operand) This performs the parallel execution of the logical right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).					
Assembler mnemonic	Note	V	C	N	Z	Size
add_lsr Rm1, Rn1, imm4, Rn2	Same as add_lsr of the ADD_OP2 instruction	-	-	-	-	4
cmp_lsr Rm1, Rn1, imm4, Rn2	Same as cmp_lsr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_lsr Rm1, Rn1, imm4, Rn2	Same as sub_lsr of the SUB_OP2 instruction	-	-	-	-	4
mov_lsr Rm1, Rn1, imm4, Rn2	Same as mov_lsr of the MOV_OP2 instruction	-	-	-	-	4
and_lsr Rm1, Rn1, imm4, Rn2	Same as and_lsr of the AND_OP2 instruction	-	-	-	-	4
or_lsr Rm1, Rn1, imm4, Rn2	Same as or_lsr of the OR_OP2 instruction	-	-	-	-	4
xor_lsr Rm1, Rn1, imm4, Rn2	Same as xor_lsr of the XOR_OP2 instruction	-	-	-	-	4
dmach_lsr Rm1, Rn1, imm4, Rn2	Same as dmach_lsr of the DMACH_OP2 instruction	-	-	-	-	4
swhw_lsr Rm1, Rn1, imm4, Rn2	Same as swhw_lsr of the SWHW_OP2 instruction	-	-	-	-	4
sat16_lsr Rm1, Rn1, imm4, Rn2	Same as sat16_lsr of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
When OP1 is other than the CMP instruction VF : No change CF : No change NF : No change ZF : No change When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.) VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases. CF : "1" when borrow to bit 31 occurs; "0" in all other cases. NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases. ZF : "1" when the operation result is "0"; "0" in all other cases.						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_LSR imm, Rn1, imm, Rn2

Operation	$[(Rn2 \gg (\text{zero_ext } imm4 * -> Rn2)]$ with op1 (* : 3rd operand) This performs the parallel execution of the logical right shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).
-----------	---

Assembler mnemonic	Note	V	C	N	Z	Size
add_lsr imm4, Rn1, imm4, Rn2	Same as add_lsr of the ADD_OP2 instruction	-	-	-	-	4
cmp_lsr imm4, Rn1, imm4, Rn2	Same as cmp_lsr of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_lsr imm4, Rn1, imm4, Rn2	Same as sub_lsr of the SUB_OP2 instruction	-	-	-	-	4
mov_lsr imm4, Rn1, imm4, Rn2	Same as mov_lsr of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_AS_L

Parallel execution of arithmetic shift left and OP1

OP1_AS_L Rm1, Rn1, Rm2, Rn2						
Operation	<p>[(Rn2 << Rm2) -> Rn2] with op1</p> <p>This performs the parallel execution of the arithmetic left shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_asl Rm1, Rn1, Rm2, Rn2	Same as add_asl of the ADD_OP2 instruction	-	-	-	-	4
cmp_asl Rm1, Rn1, Rm2, Rn2	Same as cmp_asl of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asl Rm1, Rn1, Rm2, Rn2	Same as sub_asl of the SUB_OP2 instruction	-	-	-	-	4
mov_asl Rm1, Rn1, Rm2, Rn2	Same as mov_asl of the MOV_OP2 instruction	-	-	-	-	4
and_asl Rm1, Rn1, Rm2, Rn2	Same as and_asl of the AND_OP2 instruction	-	-	-	-	4
or_asl Rm1, Rn1, Rm2, Rn2	Same as or_asl of the OR_OP2 instruction	-	-	-	-	4
xor_asl Rm1, Rn1, Rm2, Rn2	Same as xor_asl of the XOR_OP2 instruction	-	-	-	-	4
dmach_asl Rm1, Rn1, Rm2, Rn2	Same as dmach_asl of the DMACH_OP2 instruction	-	-	-	-	4
swhw_asl Rm1, Rn1, Rm2, Rn2	Same as swhw_asl of the SWHW_OP2 instruction	-	-	-	-	4
sat16_asl Rm1, Rn1, Rm2, Rn2	Same as sat16_asl of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASL imm, Rn1, Rm2, Rn2

Operation	<p>$[(Rn2 \ll Rm2) \rightarrow Rn2]$ with op1</p> <p>This performs the parallel execution of the arithmetic left shift of the register (Rn2) by the lower-5-bit values of the register (Rm2) and the op1 operation between the registers (Rm1 and Rn1).</p>
-----------	--

Assembler mnemonic	Note	V	C	N	Z	Size
add_asl imm4, Rn1, Rm2, Rn2	Same as add_asl of the ADD_OP2 instruction	-	-	-	-	4
cmp_asl imm4, Rn1, Rm2, Rn2	Same as cmp_asl of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asl imm4, Rn1, Rm2, Rn2	Same as sub_asl of the SUB_OP2 instruction	-	-	-	-	4
mov_asl imm4, Rn1, Rm2, Rn2	Same as mov_asl of the MOV_OP2 instruction	-	-	-	-	4

Flag change

When OP1 is other than the CMP instruction

VF : No change

CF : No change

NF : No change

ZF : No change

When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)

VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.

CF : "1" when borrow to bit 31 occurs; "0" in all other cases.

NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.

ZF : "1" when the operation result is "0"; "0" in all other cases.



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASL Rm1, Rn1, imm, Rn2

Operation	<p>$[(Rn2 \ll (\text{zero_ext})imm4 * -> Rn2)]$ with op1 (*:3rd operand)</p> <p>This performs the parallel execution of the arithmetic left shift of the register (Rn2) by the lower-5-bit values of the immediate value (imm4) and the op1 operation between the registers (Rm1 and Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_asl Rm1, Rn1, imm4, Rn2	Same as add_asl of the ADD_OP2 instruction	-	-	-	-	4
cmp_asl Rm1, Rn1, imm4, Rn2	Same as cmp_asl of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asl Rm1, Rn1, imm4, Rn2	Same as sub_asl of the SUB_OP2 instruction	-	-	-	-	4
mov_asl Rm1, Rn1, imm4, Rn2	Same as mov_asl of the MOV_OP2 instruction	-	-	-	-	4
and_asl Rm1, Rn1, imm4, Rn2	Same as and_asl of the AND_OP2 instruction	-	-	-	-	4
or_asl Rm1, Rn1, imm4, Rn2	Same as or_asl of the OR_OP2 instruction	-	-	-	-	4
xor_asl Rm1, Rn1, imm4, Rn2	Same as xor_asl of the XOR_OP2 instruction	-	-	-	-	4
dmach_asl Rm1, Rn1, imm4, Rn2	Same as dmach_asl of the DMACH_OP2 instruction	-	-	-	-	4
swhw_asl Rm1, Rn1, imm4, Rn2	Same as swhw_asl of the SWHW_OP2 instruction	-	-	-	-	4
sat16_asl Rm1, Rn1, imm4, Rn2	Same as sat16_asl of the SAT16_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						



When the OP1 is not a compare operation (CMP) and Rd1=Rd2 is specified, the operation result is undefined.

OP1_ASL imm, Rn1, imm, Rn2

Operation	<p>[(Rn2 << (zero_ext)imm4* -> Rn2] with op1 (*:3rd operand)</p> <p>This performs the parallel execution of the arithmetic left shift of the register (Rn2) by the lower-5-bit values of the third-operand immediate value (imm4) and the op1 operation between the immediate value (imm) and the register (Rn1).</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
add_asl imm4, Rn1, imm4, Rn2	Same as add_asl of the ADD_OP2 instruction	-	-	-	-	4
cmp_asl imm4, Rn1, imm4, Rn2	Same as cmp_asl of the CMP_OP2 instruction	Δ	Δ	Δ	Δ	4
sub_asl imm4, Rn1, imm4, Rn2	Same as sub_asl of the SUB_OP2 instruction	-	-	-	-	4
mov_asl imm4, Rn1, imm4, Rn2	Same as mov_asl of the MOV_OP2 instruction	-	-	-	-	4
Flag change						
<p>When OP1 is other than the CMP instruction</p> <p>VF : No change</p> <p>CF : No change</p> <p>NF : No change</p> <p>ZF : No change</p> <p>When OP1 is the CMP instruction (The flag change is according to the result of the CMP instruction.)</p> <p>VF : "1" when overflow occurs as a 32-bit signed number; "0" in all other cases.</p> <p>CF : "1" when borrow to bit 31 occurs; "0" in all other cases.</p> <p>NF : "1" when bit 31 of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0"; "0" in all other cases.</p>						

MOV_Lcc

Parallel execution of transfer and
loop-dedicated conditional branch

MOV_Lcc (Rm+, imm), Rn

Operation

This instruction is a combination instruction of MOV (Rm+), Rn operation and LCC operation.

MOV (Rm+), Rn instruction

mem32(Rm) -> Rn

Rm + 4 -> Rm

The data indicated by the register (Rm) is loaded from the memory (Mem), and is stored in Rn. Moreover, the register (Rm) and "4" are added to each other, and the result is stored in the register (Rm).

Lcc instruction

Lcc is used together with STLB in order to increase the loop execution speed, and performs conditional branch to the top of the loop set by SETLB.

The types of condition codes (CC) are listed below.

cc	Conditions	Meaning
eq	Z	Branch when Z = 1 or Z flag set
ne	~ Z	Branch when not Z = 1 or Z flag cleared
gt	~ (Z or (N ^ V))	Branch when < (signed)
ge	~ (N ^ V)	Branch when ≤(signed)
le	Z or (N ^ V)	Branch when ≥ (signed)
lt	N ^ V	Branch when > (signed)
hi	~ (C or Z)	Branch when < (signed)
cc	~ C	Branch when ≤ or C flag cleared (unsigned)
ls	C or Z	Branch when ≥ (unsigned)
cs	C	Branch when > or C flag set (unsigned)
ra	None	Always branch

When branch is taken

LAR -4 -> nPC (the next instruction PC)

The instruction loaded to the loop instruction register (LIR) is executed and instruction fetch starts for the address loaded to the loop address register (LAR).

At the same time, 4 is subtracted from the loop address register (LAR) and the result is written into the PC.

Even if the addition result overflows, this overflow is ignored and the result is written into the PC.

Lcc is not coordinated with SETLB, execution cannot be assured.

When branch is not taken

PC (Self-instruction address) + CodeSize -> nPC (Next instruction PC)

The next instruction is executed.

Assembler mnemonic	Note	V	C	N	Z	Size
mov_leq (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lne (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lgt (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lge (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lle (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_llt (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lhi (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lcc (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lls (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lcs (Rm+, imm4), Rn	imm4 is sign-extended.	-	-	-	-	4
mov_lra (Rm+, imm4), Rn	imm4 is sign-extended.					
Flag change						
VF : No change CF : No change NF : No change ZF : No change						



When Rd1=Rd2 is specified, the operation result is undefined.

UDF00

Signed multiplication instruction

UDF00 Dm, Dn (MULQ Dm, Dn)						
Operation	Dm * Dn -> {MDRQ , Dn} The instruction multiplies the contents of the signed 32-bit integer register (Dm: multiplicand) by the contents of the signed 32-bit integer register (Dn: multiplier), and then stores the upper 32 bits of the 64-bit result in the MDRQ and the lower 32 bits in the Dn.					
Assembler mnemonic	Note	V	C	N	Z	Size
udf00 Dm, Dn		?	?	Δ	Δ	2
Flag change						
VF : Undefined. CF : Undefined. NF : "1" when the MSB of the lower 32 bits of the operation result is "1"; "0" in all other cases. ZF : "1" when the lower 32 bits of the operation result is "0"; "0" in all other cases.						

UDF00 imm, Dn (MULQ imm, Dn)						
Operation	imm * Dn -> {MDRQ , Dn} The instruction multiplies the contents of the 32-bit sign-extended immediate value (imm: multiplicand) by the contents of the signed 32-bit integer register (Dn: multiplier), and then stores the upper 32 bits of the 64-bit result in the MDRQ and the lower 32 bits in the Dn. This instruction performs quick multiplication using the multiplier in the extension function unit.					
Assembler mnemonic	Note	V	C	N	Z	Size
udf00 imm8, Dn	imm8 is sign-extended.	?	?	Δ	Δ	3
udf00 imm16, Dn	imm16 is sign-extended.	?	?	Δ	Δ	4
udf00 imm32, Dn		?	?	Δ	Δ	6
Flag change						
VF : Undefined. CF : Undefined. NF : "1" when the MSB of the lower 32 bits of the operation result is "1"; "0" in all other cases. ZF : "1" when the lower 32 bits of the operation result is "0"; "0" in all other cases.						

UDF01

Unsigned multiplication instruction

UDF01 Dm, Dn (MULQU Dm, Dn)						
Operation	<p>$Dm * Dn \rightarrow \{MDRQ, Dn\}$</p> <p>The instruction multiplies the contents of the unsigned 32-bit integer register (Dm: multiplicand) by the contents of the unsigned 32-bit integer register (Dn: multiplier), and then stores the upper 32 bits of the 64-bit result in the MDRQ and the lower 32 bits in the Dn.</p> <p>This instruction performs quick multiplication using the multiplier in the extension function unit.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf01 Dm, Dn		?	?	Δ	Δ	2
Flag change						
<p>VF : Undefined.</p> <p>CF : Undefined.</p> <p>NF : "1" when the MSB of the lower 32 bits of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the lower 32 bits of the operation result is "0"; "0" in all other cases.</p>						

UDFU01 imm, Dn (MULQU imm, Dn)						
Operation	<p>$imm * Dn \rightarrow \{MDRQ, Dn\}$</p> <p>The instruction multiplies the contents of the 32-bit zero-extended immediate value (imm: multiplicand) by the contents of the unsigned 32-bit integer register (Dn: multiplier), and then stores the upper 32 bits of the 64-bit result in the MDRQ and the lower 32 bits in the Dn.</p> <p>This instruction performs quick multiplication using the multiplier in the extension function unit.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udfu01 imm8, Dn	imm8 is zero-extended.	?	?	Δ	Δ	3
udfu01 imm16, Dn	imm16 is zero-extended.	?	?	Δ	Δ	4
udfu01 imm32, Dn		?	?	Δ	Δ	6
Flag change						
<p>VF : Undefined.</p> <p>CF : Undefined.</p> <p>NF : "1" when the MSB of the lower 32 bits of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the lower 32 bits of the operation result is "0"; "0" in all other cases.</p>						

UDF02

Instruction of the saturation operation
for the multiply-and-accumulate result

UDF02 Dm, Dn (MCST32, MCST16, MCST8)

Operation	<p>This instruction sets the contents of the multiply-and-accumulate operation overflow detect register MCVF into the V flag. In addition, depending on the value of Dm, the following operations are performed.</p> <p>When the value of Dm is 32 (0x00000020)</p> <p>(1) When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 32-bit signed number (0x000000007FFFFFFF), the maximum positive value (0x7FFFFFFF) is stored in Dn. 0x7FFFFFFF -> Dn</p> <p>(2) If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 32-bit signed number (0xFFFFFFFF80000000), the maximum negative value (0x80000000) is stored in Dn. 0x80000000 -> Dn</p> <p>(3) In all other cases, the contents of MCRL are stored in Dn. MCRL -> Dn</p> <p>When the value of Dm is 16 (0x00000010)</p> <p>(1) When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or greater than the maximum positive value for a 16-bit signed number (0x0000000000007FFF), the maximum positive value (0x00007FFF) is stored in Dn. 0x00007FFF -> Dn</p> <p>(2) If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 16-bit signed number (0xFFFFFFFFFFFF8000), the maximum negative value (0xFFFF8000) is stored in Dn. 0x80000000 -> Dn</p> <p>(3) In all other cases, the contents of MCRL are stored in Dn. MCRL -> Dn</p> <p>When the value of Dm is 8 (0x00000008)</p> <p>(1) When the 32-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate register MCRL is equal to or greater than the maximum positive value for an 8-bit signed number (0x0000007F), the maximum positive value (0x7F) is stored in Dn. 0x7F -> Dn</p> <p>(2) If the value stored in the multiply-and-accumulate register MCRL is equal to or less than the maximum negative value for an 8-bit signed number (0xFFFFFFFF80), the maximum negative value (0x80) is stored in Dn.</p> <p>When the value of Dm is any other value The value in Dn is undefined.</p>
-----------	---

Assembler mnemonic	Note	V	C	N	Z	Size
udf02 Dm, Dn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	2
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	
Flag change						
<p>When multiply-and-accumulate operation overflow was not detected</p> <p>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When multiply-and-accumulate operation overflow was detected (MCVF = 1)</p> <p>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

UDF03

Instruction of the saturation operation
for the multiply-and-accumulate result

UDF03 Dm, Dn (MCST9 Dn)						
Operation	<p>This instruction sets the contents of the multiply-and-accumulate operation overflow detect register MCVF into the V flag. In addition, depending on the value of Dm, the following operations are performed.</p> <p>(1) When the 32-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate register MCRL is equal to or greater than the maximum positive value for a 9-bit signed number (0x000000FF), the maximum positive value (0xFF) is stored in Dn. 0xFF -> Dn</p> <p>(2) If the value stored in the multiply-and-accumulate register MCRL is equal to or less than the maximum negative value for a 8-bit signed number (0x00000000), the maximum negative value (0x00) is stored in Dn. 0x00000000 ->Dn</p> <p>(3) In all other cases, the contents of MCRL are stored in Dn. MCRL -> Dn</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf03 Dm, Dn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	2
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	
Flag change						
<p>When multiply-and-accumulate operation overflow was not detected</p> <p>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When multiply-and-accumulate operation overflow was detected (MCVF = 1)</p> <p>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

UDF04

Instruction of the 16-bit saturation operation

UDF04 Dm, Dn (SAT16 Dm, Dn)						
Operation	<p>(1) When the register Dm is equal to or greater than the maximum positive value for a 16-bit signed number (0x00007FFF) 0x00007FFF -> Dn</p> <p>(2) When the register Dm is equal to or less than the maximum negative value for a 16-bit signed number (0xFFFF8000) 0xFFFF8000 -> Dn</p> <p>In all other cases, Dm -> Dn</p> <p>(1) When the register Dm is equal to or greater than the maximum positive value for a 16-bit signed number (0x00007FFF), the maximum positive value (0x00007FFF) is stored in the register Dn.</p> <p>(2) When the register Dm is equal to or less than the maximum negative value for a 16-bit signed number (0xFFFF8000), the maximum negative value (0xFFFF8000) is stored in the register Dn.</p> <p>In all other cases, the contents of the register Dm are stored in the register Dn.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf04 Dm, Dn		?	?	Δ	Δ	2
	Flag change					
VF : Undefined.						
CF : Undefined.						
NF : "1" when the MSB of the operation result is "1"; "0" in all other cases.						
ZF : "1" when the operation result is "0", "0" in all other cases.						

UDF05

Instruction of the 24-bit saturation
operation

UDF05 Dm, Dn (SAT24 Dm, Dn)						
Operation	<p>(1) When the register Dm is equal to or greater than the maximum positive value for a 24-bit signed number (0x007FFFFFFF) 0x007FFFFFFF -> Dn</p> <p>(2) When the register Dm is equal to or less than the maximum negative value for a 16-bit signed number (0xFFFF8000) 0xFF800000 -> Dn</p> <p>In all other cases, Dm -> Dn</p> <p>(1) When the register Dm is equal to or greater than the maximum positive value for a 16-bit signed number (0x007FFFFFFF), the maximum positive value (0x007FFFFFFF) is stored in the register Dn.</p> <p>(2) When the register Dm is equal to or less than the maximum negative value for a 16-bit signed number (0xFF800000), the maximum negative value (0xFF800000) is stored in the register Dn.</p> <p>In all other cases, the contents of the register Dm are stored in the register Dn.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf05 Dm, Dn		?	?	Δ	Δ	2
Flag change						
<p>VF : Undefined.</p> <p>CF : Undefined.</p> <p>NF : "1" when the MSB of the operation result is "1"; "0" in all other cases.</p> <p>ZF : "1" when the operation result is "0", "0" in all other cases.</p>						

UDF06

Instruction of the saturation operation
for the multiply-and-accumulate result

UDF06 Dm, Dn (MCST48 Dn)						
Operation	<p>This instruction sets the contents of the multiply-and-accumulate operation overflow detect register MCVF into the V flag.</p> <p>(1) When the 64-bit result of the multiply-and-accumulate operation that is stored in the multiply-and-accumulate register MCRH and MCRL is equal to or greater than the maximum positive value for a 48-bit signed number (0x00007FFFFFFFFF), the maximum positive value (0x00007FFFFFFFFF) is output and bits 47 through bits 16 of that output are stored in Dn. 0x7FFFFFFFFF -> Dn</p> <p>(2) If the value stored in the multiply-and-accumulate registers MCRH and MCRL is equal to or less than the maximum negative value for a 48-bit signed number (0xFFFF800000000000), the maximum negative value (0xFFFF800000000000) is output and bits 47 through bits 16 of that output are stored in Dn. 0x80000000 -> Dn</p> <p>(3) In all other cases, the contents of MCRH[31:0] and MCRL[31:0] are output and bits 47 through bits 16 of that output are stored in Dn. { MCRH[15:0], MCRL[31:16] } -> Dn[31:0]</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf06 Dm, Dn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	2
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	
Flag change						
<p>When multiply-and-accumulate operation overflow was not detected</p> <p>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When multiply-and-accumulate operation overflow was detected</p> <p>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

UDF07

Bit search instruction

UDF07 Dm, Dn (BSCH Dm, Dn)						
Operation	<p>This instruction conducts a bit search within the 32-bit bit string stored in Dm, starting from the bit position of the bit number indicated by the contents of (DN - 1) and continuing in the direction of descending bit numbers. The bit number of the first bit position where a “1” is found is then stored in Dn.</p> <p>When the least significant five bits of Dn are zeroes, the bit search is conducted from bit 31 and continues in the direction of descending bit numbers.</p> <p>If the bit search reaches bit 0 without finding a “1”, the “C” flag is set, 0x00000000 is written in Dn and execution of this instruction ends.</p> <p>When execution of this instruction starts, the upper 27 bits of Dn are ignored.</p>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf07 Dm, Dn	When a bit search was succeeded ("1" was founded)	?	0	?	?	2
	When a bit search was not succeeded ("1" was not founded)	?	1	?	?	
Flag change						
<p>When a bit search was succeeded ("1" was found)</p> <p>VF: Undefined</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p> <p>When a bit search was not succeeded ("1" was not founded)</p> <p>VF: Undefined</p> <p>CF: Always “0”</p> <p>NF: Undefined</p> <p>ZF: Undefined</p>						

UDF08

Data swapping instruction that swaps bytes from high-order to low-order and vice versa in four byte data.

UDF08 Dm, Dn (SWAP Dm, Dn)										
Operation	Dm[31:24] -> Dn[7:0]									
	Dm[23:16] -> Dn[15:8]									
	Dm[15:8] -> Dn[23:16]									
	Dm[7:0] -> Dn[31:24]									
<p>This instruction swaps the positions of the high-order and low-order 8-bit bytes within the respective high-and low-order 16 bits in Dm, and then swaps the positions of the high-order and low-order 16 bits in Dm, and then stores the result in Dn.</p> <p>An example of execution is shown here.</p> <p>Before execution: Dm=0x12345678 -> After execution: Dn=0x78563412</p>										
Assembler mnemonic		Note				V	C	N	Z	Size
udf08 Dm, Dn						-	-	-	-	2
		Flag change								
VF : This is not changed.										
CF : This is not changed.										
NF : This is not changed.										
ZF : This is not changed.										

UDF09

Data swapping instruction that swaps bytes from high-order to low-order and vice versa in two byte data.

UDF09 Dm, Dn (SWAPH Dm, Dn)					
Operation	Dm[31:24] -> Dn[23:16] Dm[23:16] -> Dn[31:24] Dm[15:8] -> Dn[7:0] Dm[7:0] -> Dn[15:8] This instruction swaps the positions of the high-order and low-order 8-bit bytes within the respective high-and low-order 16 bits in Dm, and then stores the result in Dn. An example of execution is shown here. Before execution: Dm=0x12345678 -> After execution: Dn=0x34127856				
Assembler mnemonic	Note	V	C	N	Z
udf09 Dm, Dn		-	-	-	-
	Flag change				
VF : This is not changed. CF : This is not changed. NF : This is not changed. ZF : This is not changed.					

UDF12

Transfer instruction of the upper 32 bits
in the multiply-and-accumulate register

UDF12 Dm, Dn (GETCHX Dn)						
Operation	<div>.MCRH -> Dn MCVF -> EPSW.V</div> <div>This instruction stores the upper 32 bits of the multiply-and-accumulate operation register (MCRH) in the register, Dn. This sets the contents of the multiply-and-accumulate operation overflow,which is shown in MCVF, in the V flag.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf12 Dm, Dn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	2
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	
Flag change						
<div>When multiply-and-accumulate operation overflow was not detected</div> <div>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</div> <div>CF: Always “0”</div> <div>NF: Undefined</div> <div>ZF: Undefined</div> <div>When multiply-and-accumulate operation overflow was detected</div> <div>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</div> <div>CF: Always “0”</div> <div>NF: Undefined</div> <div>ZF: Undefined</div>						

UDF13

Transfer instruction of the upper 32 bits
in the multiply-and-accumulate register

UDF13 Dm, Dn (GETCLX Dn)						
Operation	<div>.MCRL -> Dn MCVF -> EPSW.V</div> <div>This instruction stores the lower 32 bits of the multiply-and-accumulate operation register (MCRH) in the register, Dn. This sets the contents of the multiply-and-accumulate operation overflow,which is shown in MCVF, in the V flag.</div>					
Assembler mnemonic	Note	V	C	N	Z	Size
udf13 Dm, Dn	When multiply-and-accumulate operation overflow was not detected (MCVF = 0)	0	0	?	?	2
	When multiply-and-accumulate operation overflow was detected (MCVF = 1)	1	0	?	?	
Flag change						
<div>When multiply-and-accumulate operation overflow was not detected</div> <div>VF: This is "0" and indicates that the multiply-and-accumulate operation is valid.</div> <div>CF: Always “0”</div> <div>NF: Undefined</div> <div>ZF: Undefined</div> <div>When multiply-and-accumulate operation overflow was detected</div> <div>VF: This is "1" and indicates that the multiply-and-accumulate operation is invalid.</div> <div>CF: Always “0”</div> <div>NF: Undefined</div> <div>ZF: Undefined</div>						



UDF15

Transfer instruction of the high-speed multiplication register

UDF15 Dm, Dn (GETX Dn)						
Operation	MDRQ -> Dn					
	The instruction stores the high-speed multiplication register (MDRQ) in the register, Dn.					
Assembler mnemonic	Note	V	C	N	Z	Size
udf15 Dm, Dn		0	0	Δ	Δ	2
	Flag change					
VF : Undefined.						
CF : Undefined.						
NF : "1" when the MSB of the operation result is "1"; "0" in all other cases.						
ZF : "1" when the operation result is "0"; "0" in all other cases.						

FMOV

Transfer (Floating-point unit)

FMOV Mem, FS _n											
 											
Operation	<p>FMOV in the case of (R_m), FS_n mem32(R_m) -> FS_n</p> <p>FMOV in the case of (SP), FS_n mem32(SP) -> FS_n</p> <p>FMOV in the case of (R_m, R_i), FS_n mem32(R_m+R_i) -> FS_n</p> <p>FMOV in the case of (disp, R_m), FS_n mem32(R_m+disp) -> FS_n</p> <p>FMOV in the case of (disp, SP), FS_n mem32(SP+disp) -> FS_n</p> <p>Memory contents are stored in the single-precision floating-point register (FS_n).</p>										
Assembler mnemonic	Note	EC flag					FCC flag				
		V	Z	O	U	I	L	G	E	U	Size
fmov (R _m), FS _n		-	-	-	-	-	-	-	-	-	3
fmov (SP), FS _n		-	-	-	-	-	-	-	-	-	3
fmov (R _m , R _i), FS _n		-	-	-	-	-	-	-	-	-	4
fmov (d8, R _m), FS _n	d8 is code-extended.	-	-	-	-	-	-	-	-	-	4
fmov (d24, R _m), FS _n	d24 is code-extended.	-	-	-	-	-	-	-	-	-	6
fmov (d32, R _m), FS _n		-	-	-	-	-	-	-	-	-	7
fmov (d8, SP), FS _n	d8 is zero-extended.	-	-	-	-	-	-	-	-	-	4
fmov (d24, SP), FS _n	d24 is zero-extended.	-	-	-	-	-	-	-	-	-	6
fmov (d32, SP), FS _n		-	-	-	-	-	-	-	-	-	7
Flag change											
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change											



When the memory address is not a multiple of 4, a system exception (address misalignment exception) occurs.



FMOV FSm, Mem



Operation	FMOV in the case of FSm, (Rn) FSm -> mem32 (Rn)												
	FMOV in the case of FSm, (SP) FSm -> mem32 (SP)												
	FMOV in the case of FSm, (Rn, Ri) FSm -> mem32 (Rn+Ri)												
	FMOV in the case of FSm, (disp, Rn) FSm -> mem32 (Rn+disp)												
	FMOV in the case of FSm, (disp, SP) FSm -> mem32 (SP+disp)												
	The contents of the single-precision floating-point register (FSn) are stored in the memory.												
Assembler mnemonic		Notes		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fmov FSm, (Rn)				-	-	-	-	-	-	-	-	-	3
fmov FSm, (SP)				-	-	-	-	-	-	-	-	-	3
fmov FSm, (Rn, Ri)				-	-	-	-	-	-	-	-	-	4
fmov FSm, (d8, Rn)		d8 is code-extended.		-	-	-	-	-	-	-	-	-	4
fmov FSm, (d24, Rn)		d24 is code-extended.		-	-	-	-	-	-	-	-	-	6
fmov FSm, (d32, Rn)				-	-	-	-	-	-	-	-	-	7
fmov FSm, (d8, SP)		d8 is zero-extended.		-	-	-	-	-	-	-	-	-	4
fmov FSm, (d24, SP)		d24 is zero-extended.		-	-	-	-	-	-	-	-	-	6
fmov FSm, (d32, SP)				-	-	-	-	-	-	-	-	-	7
Flag change													
VF : No change													
ZF : No change													
OF : No change													
UF : No change													
IF : No change													
LF : No change													
GF : No change													
EF : No change													
UF : No change													





When the memory address is not a multiple of 4, a system exception (address misalignment exception) occurs.

FMOV (Rm+), FSn												
Operation	mem32(Rm) -> FSn Rm + 0x00000004 -> Rm Data specified by the register (Rm) are loaded from the memory and stored in the single-precision floating-point register (FSn). Moreover, the register (Rm) and 4 are added and the result is stored in the register (Rm).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov (Rm+), FSn		-	-	-	-	-	-	-	-	-	3	
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												





When memory address is not a multiple of 4, system exception (address misalignment exception) occurs.

FMOV (Rm+, imm), FSn															
Operation	mem32(Rm) -> FSn														
	Rm + imm -> Rm														
	Data specified by the register (Rm) are loaded from the memory (Mem) and stored in the contents of the single-precision floating-point register (FSn).														
	Moreover, the register (Rm) and immediate value (imm) are added and the result is stored in the register (Rm).														
Assembler mnemonic		Note				EC flag					FCC flag				Size
						V	Z	O	U	I	L	G	E	U	
fmov (Rn+, imm8), FSn		imm8 is code-extended.				-	-	-	-	-	-	-	-	-	4
fmov (Rn+, imm24), FSn		imm24 is code-extended.				-	-	-	-	-	-	-	-	-	6
fmov (Rn+, imm32), FSn						-	-	-	-	-	-	-	-	-	7
Flag change															
VF : No change															
ZF : No change															
OF : No change															
UF : No change															
IF : No change															
LF : No change															
GF : No change															
EF : No change															
UF : No change															



When memory address is not a multiple of 4, system exception (address misalignment exception) occurs.

FMOV FSm, (Rn+)												
Operation	FSm -> mem32(Rn) Rn + 0x00000004 -> Rn The contents of the single-precision floating-point register (FSm) are stored in the memory specified by the register (Rn). Moreover, the register (Rn) and 4 are added and the result is stored in the register (Rn).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov FSn, (Rn+)		-	-	-	-	-	-	-	-	-	3	
Flag change												
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												



When the memory address is not a multiple of 4, system exception (address misalignment exception) occurs.



FMOV FSm, (Rn+, imm)







Operation	FSm -> mem32 (Rn) Rn + imm -> Rn The contents of the single-precision floating-point register (FSm) are stored in the memory (Mem) specified by the register (Rn). Moreover, the register (Rn) and immediate value (imm) are added and the result is stored in the register (Rn).										
Assembler mnemonic	Note	EC flag					FCC flag				Size
		V	Z	O	U	I	L	G	E	U	
fmov FSm, (Rn+, imm8)	imm8 is code-extended.	-	-	-	-	-	-	-	-	-	4
fmov FSm, (Rn+, imm24)	imm24 is code-extended.	-	-	-	-	-	-	-	-	-	6
fmov FSm, (Rn+, imm32)		-	-	-	-	-	-	-	-	-	7
Flag change											
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change											







When the memory address is not a multiple of 4, system exception (address misalignment exception) occurs.



FMOV FSm, FS <i>n</i>												
Operation	FS <i>m</i> -> FS <i>n</i> The contents of the single-precision-floatingpoint register (FS <i>m</i>) are stored in the single-precision floating-point register (FS <i>n</i>).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov FS <i>m</i> , FS <i>n</i>		-	-	-	-	-	-	-	-	-	3	
Flag change												
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												

FMOV FSm, Rn												
Operation	FSm -> Rn The contents of the single-precision floating-point register (FSm) are stored in the register (Rn).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov FSm, Rn		-	-	-	-	-	-	-	-	-	3	
Flag change												
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												

FMOV Rm,FSn													
Operation	Rm -> FSn The contents of the register (Rm) are stored in the single-precision floating-point register(FSn).												
Assembler mnemonic	Note	EC flag					FCC flag				Size		
		V	Z	O	U	I	L	G	E	U			
fmov Rm,FSn		-	-	-	-	-	-	-	-	-	3		
Flag change													
VF : No change													
ZF : No change													
OF : No change													
UF : No change													
IF : No change													
LF : No change													
GF : No change													
EF : No change													
UF : No change													

FMOV imm,FSn												
Operation	imm32 -> FS _n The immediate value (imm) is stored in the single-precision floating-point register (FS _n).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov imm32,FS _n		-	-	-	-	-	-	-	-	-	7	
Flag change												
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												



FMOV FPCR,Rn																
Operation	FPCR -> Rn															
	The contents of the floating-point control unit register (FPCR) are stored in the register (Rn).															
Assembler mnemonic	Note	EF flag					EC flag					FCC flag				Size
		V	Z	O	U	I	V	Z	O	U	I	L	G	E	U	
fmov FPCR,Rn		-	-	-	-	-	-	-	-	-	-	-	-	-	-	3
Flag change																
EF flag	VF : No change															
	ZF : No change															
	OF : No change															
	UF : No change															
	IF : No change															
EC flag	VF : No change															
	ZF : No change															
	OF : No change															
	UF : No change															
	IF : No change															
FCC flag	LF : No change															
	GF : No change															
	EF : No change															
	UF : No change															

FMOV Rm,FPCR																			
Operation		Rm -> FPCR																	
		The contents of the register (Rm) are stored in the floating-point control unit register (FCPR).																	
Assembler mnemonic		Note		EF flag					EC flag					FCC flag				Size	
				V	Z	O	U	I	V	Z	O	U	I	L	G	E	U		
fmov Rm,FPCR				Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	3		
Flag change																			
EF flag		VF : This is "0" when Bit 4 of the source data is "1". This not changed when Bit 4 of the source data is"0". ZF : This is "0" when Bit 3 of the source data is "1". This is not changed when Bit 3 of the source data is"0". OF : This is "0" when Bit 2 of the source data is "1". This is not changed when Bit 2 of the source data is"0". UF : This is "0" when Bit 1 of the source data is "1". This is not changed when Bit 1 of the source data is "0". IF : This is "0" when Bit 0 of the source data is "1". This is not changed when Bit 0 of the source data is "0".																	
EC flag		VF : Bit 14 of the source data is set. ZF : Bit 13 of the source data is set. OF : Bit 12 of the source data is set. UF : Bit 11 of the source data is set. IF : Bit 10 of the source data is set.																	
FCC flag		LF : Bit 21 of the source data is set. GF : Bit 20 of the source data is set. EF : Bit 19 of the source data is set. UF : Bit 18 of the source data is set.																	

FMOV imm,FPCR



Operation	imm32 -> FPCR															
	The contents of the immediate values (imm32) are stored in the floating-point control unit register (FCPR).															
Assembler mnemonic	Note	EF flag					EC flag					FCC flag				Size
		V	Z	O	U	I	V	Z	O	U	I	L	G	E	U	
fmov imm32,FPCR		Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	6
Flag change																
EF flag	VF : This is "0" when Bit 4 of the source data is "1". This not changed when Bit 4 of the source data is"0". ZF : This is "0" when Bit 3 of the source data is "1". This is not changed when Bit 3 of the source data is"0". OF : This is "0" when Bit 2 of the source data is "1". This is not changed when Bit 2 of the source data is"0". UF : This is "0" when Bit 1 of the source data is "1". This is not changed when Bit 1 of the source data is "0". IF : This is "0" when Bit 0 of the source data is "1". This is not changed when Bit 0 of the source data is "0".															
EC flag	VF : Bit 14 of the source data is set. ZF : Bit 13 of the source data is set. OF : Bit 12 of the source data is set. UF : Bit 11 of the source data is set. IF : Bit 10 of the source data is set.															
FCC flag	LF : Bit 21 of the source data is set. GF : Bit 20 of the source data is set. EF : Bit 19 of the source data is set. UF : Bit 18 of the source data is set.															

FMOV Mem,FDn												
Operation	<div>FMOV (Rm),FDn mem64(Rm) -> FDn</div> <div>FMOV (SP),FDn mem64(SP) -> FDn</div> <div>FMOV (Rm,Ri),FSn mem64(Rm + Ri)-> FDn</div> <div>FMOV (disp,Rm),FDn mem64(disp+ Rm) -> FDn</div> <div>FMOV (disp,SP),FDn mem64(disp+ SP) -> FDn</div> <div>FDn -> {FSn+1,FSn} (n : even)</div> <div>The contents of the 64-bit memory are stored in the double-precision floating point register (FDn).</div>											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov (Rm),FDn		-	-	-	-	-	-	-	-	-	3	
fmov (SP),FDn		-	-	-	-	-	-	-	-	-	3	
fmov (Rm,Ri),FDn		-	-	-	-	-	-	-	-	-	4	
fmov (d8,Rm),FDn	d8 is sign-extended.	-	-	-	-	-	-	-	-	-	4	
fmov (d24,Rm),FDn	d24 is sign-extended.	-	-	-	-	-	-	-	-	-	6	
fmov (d32,Rm),FDn		-	-	-	-	-	-	-	-	-	7	
fmov (d8,SP),FDn	d8 is zero-extended.	-	-	-	-	-	-	-	-	-	4	
fmov (d24,SP),FDn	d24 is zero-extended.	-	-	-	-	-	-	-	-	-	6	
fmov (d32,SP),FDn		-	-	-	-	-	-	-	-	-	7	
Flag change												
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												



When the Mem address is not a multiple of 8, system exception (Address misalignment exception) occurs.

FMOV Fm,Mem



Operation	<p>FMOV Fm,(Rn) Fm -> mem64(Rn)</p> <p>FMOV Fm,(SP) Fm -> mem64(SP)</p> <p>FMOV Fm,(Rn,Ri) Fm -> mem64(Rn + Ri)</p> <p>FMOV Fm,(disp,Rn) Fm -> mem64(Rn + disp)</p> <p>FMOV Fm,(disp,SP) Fm -> mem64(SP + disp)</p> <p>Fm -> {FSm+1,FSm} (m : even)</p> <p>The contents of the double-precision floating point register (Fm) are stored in the 64-bit memory.</p>
-----------	---



Assembler mnemonic	Note	EC flag					FCC flag				Size
		V	Z	O	U	I	L	G	E	U	
fmov Fm,(Rn)		-	-	-	-	-	-	-	-	-	3
fmov Fm,(SP)		-	-	-	-	-	-	-	-	-	3
fmov Fm,(Rn,Ri)		-	-	-	-	-	-	-	-	-	4
fmov Fm,(d8,Rn)	d8 is sign-extended.	-	-	-	-	-	-	-	-	-	4
fmov Fm,(d24,Rn)	d24 is sign-extended.	-	-	-	-	-	-	-	-	-	6
fmov Fm,(d32,Rn)		-	-	-	-	-	-	-	-	-	7
fmov Fm,(d8,SP)	d8 is zero-extended.	-	-	-	-	-	-	-	-	-	4
fmov Fm,(d24,SP)	d24 is zero-extended.	-	-	-	-	-	-	-	-	-	6
fmov Fm,(d32,SP)		-	-	-	-	-	-	-	-	-	7

Flag change

VF : No change
 ZF : No change
 OF : No change
 UF : No change
 IF : No change
 LF : No change
 GF : No change
 EF : No change
 UF : No change





When the Mem address is not a multiple of 8, system exception (Address misalignment exception) occurs.

FMOV (Rm+),FDn													
Operation	mem64(Rm)-> FDn												
	Rm + 0x00000008 -> Rm												
	FDn = {FSn+1,FSn} (n:even)												
The contents of the 64-bit memory which is specified by the register (Rm) are stored in the double-precision floating-point register (FDn). Then, 0x00000008 is added to the register (Rm).													
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fmov (Rm+),FDn				-	-	-	-	-	-	-	-	-	3
Flag change													
VF : No change													
ZF : No change													
OF : No change													
UF : No change													
IF : No change													
LF : No change													
GF : No change													
EF : No change													
UF : No change													





When the Mem address is not a multiple of 8, system exception (Address misalignment exception) occurs.

FMOV (Rm+,imm),FDn													
Operation	mem64(Rm) -> FDn												
	Rm + imm -> Rm												
	FDn = {FSn+1,FSn} (n:even)												
	The contents of the 64-bit memory which is specified by the register (Rm) are stored in the double-precision floating-point register (FDn). Then, an immediate value is added to the register (Rm).												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fmov (Rm+,imm8),FDn		imm8 is sign-extended.		-	-	-	-	-	-	-	-	-	4
fmov (Rm+,imm24),FDn		imm24 is sign-extended.		-	-	-	-	-	-	-	-	-	6
fmov (Rm+,imm32),FDn				-	-	-	-	-	-	-	-	-	7
Flag change													
VF : No change													
ZF : No change													
OF : No change													
UF : No change													
IF : No change													
LF : No change													
GF : No change													
EF : No change													
UF : No change													



When the Mem address is not a multiple of 8, system exception (Address misalignment exception) occurs.

FMOV FDm,(Rn+)												
Operation	FDm -> mem64(Rn) Rn + 0x00000008 -> Rn FDm={FSm+1,FSm}(m:even) The contents of the double-precision floating-point register (FDm) which is specified by the register (Rn) are stored in the 64-bit memory. Then 0x00000008 is added to the register (Rn).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fmov FDm,(Rn+)		-	-	-	-	-	-	-	-	-	3	
Flag change												
VF : No change ZF : No change OF : No change UF : No change IF : No change LF : No change GF : No change EF : No change UF : No change												



When the Mem address is not a multiple of 8, system exception (Address misalignment exception) occurs.

FMOV F_{Dm}, (R_n+, imm)





Operation	FDm -> mem64(Rn) Rn + imm -> Rn FDm={FSm+1,FSm}(m:even) The contents of the double-precision floating-point register (FDm) are stored in the 64-bit memory specified by the register (Rn). Then an immediate value is added to the register (Rn).											
	Assembler mnemonic	Note	EC flag					FCC flag				Size
			V	Z	O	U	I	L	G	E	U	
fmov FDm,(Rn+,imm8)	imm8 is sign-extended	-	-	-	-	-	-	-	-	-	4	
fmov FDm,(Rn+,imm24)	imm24 is sign-extended	-	-	-	-	-	-	-	-	-	6	
fmov FDm,(Rn+,imm32)		-	-	-	-	-	-	-	-	-	7	
Flag change												
VF : No change												
ZF : No change												
OF : No change												
UF : No change												
IF : No change												
LF : No change												
GF : No change												
EF : No change												
UF : No change												





When the Mem address is not a multiple of 8, system exception (Address misalignment exception) occurs.

FABS



Floating-point absolute value



FABS FS _n													
Operation	FS _n -> FS _n												
	This takes an absolute value of the register (FS _n), and stores the result in the register (FS _n).												
Assembler mnemonic	Note	EC flag					FCC flag				Size		
		V	Z	O	U	I	L	G	E	U			
fabs FS _n		-	-	-	-	-	-	-	-	-	3		
Flag change													
VF: This is not changed.													
ZF: This is not changed.													
OF: This is not changed.													
UF: This is not changed.													
IF: This is not changed.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

FABS FSm, FS <i>n</i>													
Operation	F <i>Sm</i> -> F <i>Sn</i>												
	This takes an absolute value of the register (F <i>Sm</i>), and stores the result in the register (F <i>Sn</i>).												
Assembler mnemonic	Note	EC flag					FCC flag				Size		
		V	Z	O	U	I	L	G	E	U			
fabs F <i>Sm</i> , F <i>Sn</i>		-	-	-	-	-	-	-	-	-	4		
Flag change													
VF: This is not changed.													
ZF: This is not changed.													
OF: This is not changed.													
UF: This is not changed.													
IF: This is not changed.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

FNEG

Floating-point negative numbers

FNEG FS _n													
Operation		FS _n * (-1) -> FS _n This multiplies the register (FS _n) by -1, and stores the result in the register (FS _n).											
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fneg FS _n				-	-	-	-	-	-	-	-	-	3
Flag change													
VF: This is not changed. ZF: This is not changed. OF: This is not changed. UF: This is not changed. IF: This is not changed. LF: This is not changed. GF: This is not changed. EF: This is not changed. UF: This is not changed.													

FNEG FSm, FS <i>n</i>													
Operation	FS <i>n</i> * (-1) -> FS <i>n</i>												
	This multiplies the register (FS <i>m</i>) by -1, and stores the result in the register (FS <i>n</i>).												
Assembler mnemonic	Note	EC flag					FCC flag				Size		
		V	Z	O	U	I	L	G	E	U			
fneg F <i>Sm</i> , F <i>Sn</i>		-	-	-	-	-	-	-	-	-	4		
Flag change													
VF: This is not changed.													
ZF: This is not changed.													
OF: This is not changed.													
UF: This is not changed.													
IF: This is not changed.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

1/square root of the floating point

FRSQRT 255

(1) When the FPU operation exception enable flag is "1"

FSm	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
Data Output	1/Sqrt	-	-	-	+0	-	qNaN	-

FSm	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
EC Flag	1/Sqrt	V	Z	Z	0	V	0	V

(1-1) 1/Sqrt

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-	O
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	* / -	0 / I
$0\text{x7f7fffff} < v$	-	O

(2) When the FPU operation exception enable flag is "0"

FSm	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
Data Output	1/Sqrt	qNaN	+INF	-INF	+0	qNaN	qNaN	qNaN



FSm	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
EC Flag	1/Sqrt	-	-	-	-	-	-	-



(2-1) 1/Sqrt

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-INF	-
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 / +0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	*	-
$0\text{x7f7fffff} < v$	+INF	-

FCMP

Comparison of the floating-point data

FCMP FSm1, FSm2												 
Operation	FSm2 - FSm1 : FPCR (EC) This subtracts the register (FSm1) from the register (FSm2), and reflects the result in the floating-point unit control register (FPCR).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fcmp FSm1, FSm2		Δ	0	0	0	0	Δ	Δ	Δ	Δ	3	
Flag change												
VF: This is "1" when the source data is sNaN. This is "0" in all other cases. ZF: This is always "0". OF: This is always "0". UF: This is always "0". IF: This is always "0". LF: This is not changed. GF: This is not changed. EF: This is not changed. UF: This is not changed.												

FCMP imm, FSm												 
Operation	FSm - imm32 : FPCR (EC) This subtracts the register (FSm1) from the register (FSm2), and reflects the result in the floating-point unit control register (FPCR).											
Assembler mnemonic	Note	EC flag					FCC flag				Size	
		V	Z	O	U	I	L	G	E	U		
fcmp imm32, FSm, FSm		Δ	0	0	0	0	Δ	Δ	Δ	Δ	3	
Flag change												
VF: This is "1" when the source data is sNaN. This is "0" in all other cases. ZF: This is always "0". OF: This is always "0". UF: This is always "0". IF: This is always "0". LF: This is "1" when FSm1>FSm2 or fimm32>FSm. GF: This is "1" when FSm1>FSm2 or fimm32>FSm. EF: This is "1" when FSm1=FSm2 or fimm32=FSm. UF: This is "1" when comparison is impossible.												

(1) When the FPU operation exception enable flag is "1"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-

EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	0	0	0	0	0	0	V
	+0	0	0	0	0	0	0	V
	-0	0	0	0	0	0	0	V
	+INF	0	0	0	0	0	0	V
	-INF	0	0	0	0	0	0	V
	qNaN	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V

FCC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	CMP	CMP	CMP	G	L	U	0
	+0	CNP	E	E	G	L	U	0
	-0	CNP	E	E	G	L	U	0
	+INF	L	L	L	E	L	U	0
	-INF	G	G	G	G	E	U	0
	qNaN	U	U	U	U	U	U	0
	sNaN	0	0	0	0	0	0	0

(1-1) CMP instruction

Calculation result	Data Output	EC Flag	FCC Flag
$v < 0\text{xff7fffff}$	-	-	L
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	-	-	L
$0\text{x80800000} < v < 0$	-	-	L
0	-	-	E
$0 < v < 0\text{x00800000}$	-	-	G
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	-	-	G
$0\text{x7f7fffff} < v$	-	-	G

(1) When the FPU operation exception enable flag is "0"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-

EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-



FCC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	CMP	CMP	CMP	G	L	U	U
	+0	CNP	E	E	G	L	U	U
	-0	CNP	E	E	G	L	U	U
	+INF	L	L	L	E	L	U	U
	-INF	G	G	G	G	E	U	U
	qNaN	U	U	U	U	U	U	U
	sNaN	U	U	U	U	U	U	U



(2-1) CMP instruction



Calculation result	Data Output	EC Flag	FCC Flag
$v < 0\text{xff7ffff}$	-	-	L
$0\text{xff7ffff} \leq v \leq 0\text{x80800000}$	-	-	L
$0\text{x80800000} < v < 0$	-	-	L
0	-	-	E
$0 < v < 0\text{x00800000}$	-	-	G
$0\text{x00800000} \leq v \leq 0\text{x7f7ffff}$	-	-	G
$0\text{x7f7ffff} < v$	-	-	G

FADD

Addition of the floating-point data

FADD FSm, FS <i>n</i>													
Operation		FS <i>n</i> + FSm -> FS <i>n</i> This adds the contents of the register (FS <i>n</i>) and the register (FS <i>m</i>) to each other, and stores the result in the register (FS <i>n</i>).											
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fadd FSm, FS <i>n</i>				Δ	0	Δ	Δ	Δ	-	-	-	-	3
Flag change													
VF: This is "1" when the source data is sNaN or (+INF)+(-INF). This is "0" in all other cases. ZF: This is always "0". OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases. UF: This is "1" when the operation result is not "0" and is between ±2E _{min} . This is "0" in all other cases. IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases. LF: This is not changed. GF: This is not changed. EF: This is not changed. UF: This is not changed.													

FADD FSm1, FSm2, FS <i>n</i>													
Operation	FS <i>m</i> 1 + FSm2 -> FS <i>n</i>												
	This adds the contents of the register (FS <i>m</i> 2) and the register (FS <i>m</i> 1) to each other, and stores the result in the register (FS <i>n</i>).												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fadd FSm1, FSm2, FS <i>n</i>				Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change													
VF: This is "1" when the source data is sNaN or (+INF)+(-INF). This is "0" in all other cases.													
ZF: This is always "0".													
OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases.													
UF: This is "1" when the operation result is not "0" and is between ±2Emin. This is "0" in all other cases.													
IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

FADD imm, FSm, FS _n													
Operation	FS _m + imm32 -> FS _n												
	This adds the register (FS _m) to the immediate value (imm32), and stores the result in the register (FS _n).												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fadd imm32, FSm, FS _n				Δ	-	Δ	Δ	Δ	-	-	-	-	7
Flag change													
VF: This is "1" when the source data is sNaN or (+INF)+(-INF). This is "0" in all other cases.													
ZF: This is always "0".													
OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases.													
UF: This is "1" when the operation result is not "0" and is between ±2E _{min} . This is "0" in all other cases.													
IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

(1) When the FPU operation exception enable flag

Data Output		FS _{m2} / FS _m						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FS _{m1} / imm32	NORM	ADD	ADD	ADD	+INF	-INF	qNaN	-
	+0	ADD	+0	+0	+INF	-INF	qNaN	-
	-0	ADD	+0	-0	+INF	-INF	qNaN	-
	+INF	+INF	+INF	+INF	+INF	-	qNaN	-
	-INF	-INF	-INF	-INF	-	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-

EC Flag		FS _{m2} / FS _m						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FS _{m1} / imm32	NORM	ADD	0	0	0	0	0	V
	+0	0	0	0	0	0	0	V
	-0	0	0	0	0	0	0	V
	+INF	0	0	0	0	V	0	V
	-INF	0	0	0	V	0	0	V
	qNaN	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V

(1-1) ADD instruction

Calculation result	Data Output	EC Flag
$v < 0\text{xff7ffff}$	-	O
$0\text{xff7ffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7ffff}$	* / -	0 / I
$0\text{x7f7ffff} < v$	-	O

(2) When the FPU operation exception enable flag is "0"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	ADD	ADD	ADD	+INF	-INF	qNaN	qNaN
	+0	ADD	+0	+0	+INF	-INF	qNaN	qNaN
	-0	ADD	+0	-0	+INF	-INF	qNaN	qNaN
	+INF	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN
	-INF	-INF	-INF	-INF	qNaN	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN



EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-



(2-1) ADD instruction



Calculation result	Data Output	EC Flag
$v < 0\text{xff7ffff}$	-INF	-
$0\text{xff7ffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 / +0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7ffff}$	*	-
$0\text{x7f7ffff} < v$	+INF	-

FSUB

Subtraction of the floating-point data

FSUB FSm, FS _n												
Operation	FS _n - FSm -> FS _n											
	This subtracts the contents of the register (FS _m) from the contents of the register (FS _n), and stores the result in the register (FS _n).											
Assembler mnemonic		Note	EC flag					FCC flag				Size
			V	Z	O	U	I	L	G	E	U	
fsud FSm, FS _n			Δ	0	Δ	Δ	Δ	-	-	-	-	3
Flag change												
VF: This is "1" when the source data is sNaN or (±INF)+(±INF). This is "0" in all other cases.												
ZF: This is always "0".												
OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases.												
UF: This is "1" when the operation result is not "0" and is between ±2E _{min} . This is "0" in all other cases.												
IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases.												
LF: This is not changed.												
GF: This is not changed.												
EF: This is not changed.												
UF: This is not changed.												

FSUB FSm1, FSm2, FS _n													
Operation	FSm2 - FSm1 -> FS _n												
	This subtracts the contents of the register (FSm1) from the contents of the register (FSm2), and stores the result in the register (FS _n).												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fsub FSm1, FSm2, FS _n				Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change													
VF: This is "1" when the source data is sNaN or (±INF)+(±INF). This is "0" in all other cases.													
ZF: This is always "0".													
OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases.													
UF: This is "1" when the operation result is not "0" and is between ±2Emin. This is "0" in all other cases.													
IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

FSUB imm, FSm, FS _n													
Operation		FS _m - imm32 -> FS _n This subtracts the immediate value (imm32) from the contents of the register (FS _m), and stores the result in the register (FS _n).											
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fsub imm32, FSm, FS _n				Δ	0	Δ	Δ	Δ	-	-	-	-	7
Flag change													
VF: This is "1" when the source data is sNaN or (±INF)+(±INF). This is "0" in all other cases. ZF: This is always "0". OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases. UF: This is "1" when the operation result is not "0" and is between ±2E _{min} . This is "0" in all other cases. IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases. LF: This is not changed. GF: This is not changed. EF: This is not changed. UF: This is not changed.													

(1) When the FPU operation exception enable flag is "1"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	SUB	SUB	SUB	+INF	-INF	qNaN	-
	+0	SUB	+0	-0	+INF	-INF	qNaN	-
	-0	SUB	+0	+0	+INF	-INF	qNaN	-
	+INF	-INF	-INF	-INF	•	-INF	qNaN	-
	-INF	+INF	+INF	+INF	+INF	•	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-

EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	SUB	0	0	0	0	0	V
	+0	0	0	0	0	0	0	V
	-0	0	0	0	0	0	0	V
	+INF	0	0	0	V	0	0	V
	-INF	0	0	0	0	V	0	V
	qNaN	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V

(1-1) SUB instruction

Calculation result	Data Output	EC Flag
$v < 0\text{xff7ffff}$	-	O
$0\text{xff7ffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7ffff}$	* / -	0 / I
$0\text{x7f7ffff} < v$	-	O

(2) When the FPU operation exception enable flag is "0"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	SUB	SUB	SUB	+INF	-INF	qNaN	qNaN
	+0	SUB	+0	-0	+INF	-INF	qNaN	qNaN
	-0	SUB	+0	+0	+INF	-INF	qNaN	qNaN
	+INF	-INF	-INF	-INF	qNaN	-INF	qNaN	qNaN
	-INF	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN



EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-



(2-1) SUB instruction

Calculation result	Data Output	EC Flag
$v < 0\text{xff7ffff}$	-INF	-
$0\text{xff7ffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 / +0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7ffff}$	*	-
$0\text{x7f7ffff} < v$	+INF	-

FMUL

Multiplication of floating-point data

FMUL FSm, FS <i>n</i>													
Operation	FS <i>n</i> * FSm -> FS <i>n</i>												
	This multiplies the contents of the register (FS <i>m</i>) by the contents of the register (FS <i>n</i>), and stores the result in the register (FS <i>n</i>).												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fmul FSm, FS <i>n</i>				Δ	0	Δ	Δ	Δ	-	-	-	-	3
Flag change													
VF: This is "1" when the source data is sNaN or (±INF) * (0). This is "0" in all other cases.													
ZF: This is always "0".													
OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases.													
UF: This is "1" when the operation result is not "0" and is between ±2Emin. This is "0" in all other cases.													
IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

FMUL FSm1, FSm2, FS <i>n</i>													
Operation	FS <i>n</i> * FSm -> FS <i>n</i>												
	This multiplies the contents of the register (FS <i>m</i> 1) by the contents of the register (FS <i>m</i> 2), and stores the result in the register (FS <i>n</i>).												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fmul FSm1, FSm2, FS <i>n</i>				Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change													
VF: This is "1" when the source data is sNaN or (±INF) * (0). This is "0" in all other cases.													
ZF: This is always "0".													
OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases.													
UF: This is "1" when the operation result is not "0" and is between ±2Emin. This is "0" in all other cases.													
IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

FMUL imm32, FSm, FS_n



Operation	FS _n * imm32 -> FS _n This multiplies the contents of the register (FS _m) by the immediate value (imm32), and stores the result in the register (FS _n).										
Assembler mnemonic	Note	EC flag					FCC flag				Size
		V	Z	O	U	I	L	G	E	U	
fmul imm32, FSm, FS _n		Δ	0	Δ	Δ	Δ	-	-	-	-	7
Flag change											
VF: This is "1" when the source data is sNaN or (±INF) * (0). This is "0" in all other cases. ZF: This is always "0". OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases. UF: This is "1" when the operation result is not "0" and is between ±2Emin. This is "0" in all other cases. IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases. LF: This is not changed. GF: This is not changed. EF: This is not changed. UF: This is not changed.											

(1) When the FPU operation exception enable flag is "1"

Data Output		FS _{m2} / FS _m						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FS _{m1} / imm32	NORM	MUL	0	0	INF	INF	qNaN	-
	+0	0	+0	-0	-	-	qNaN	-
	-0	0	-0	+0	-	-	qNaN	-
	+INF	INF	-	-	+INF	-INF	qNaN	-
	-INF	INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-

EC Flag		FS _{m2} / FS _m						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FS _{m1} / imm32	NORM	MUL	0	0	0	0	0	V
	+0	0	0	0	V	V	0	V
	-0	0	0	0	V	V	0	V
	+INF	0	V	V	0	0	0	V
	-INF	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V

(1-1) MUL instruction

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-	O
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	* / -	0 / I
$0\text{x7f7fffff} < v$	-	O

(2) When the FPU operation exception enable flag is "0"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	MUL	0	0	INF	INF	qNaN	qNaN
	+0	0	+0	-0	qNaN	qNaN	qNaN	qNaN
	-0	MUL	-0	+0	qNaN	qNaN	qNaN	qNaN
	+INF	INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	-INF	INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-

(2-1) MUL instruction

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-INF	-
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 / +0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	*	-
$0\text{x7f7fffff} < v$	+INF	-



FDIV imm, FSm, FS_n

Operation	FS _m /imm32 -> FS _n									
	This divides the contents of the register (FS _m) by the immediate value (imm32), and stores the result in the register (FS _n).									
Assembler mnemonic	Note	EC flag					FCC flag			
		V	Z	O	U	I	L	G	E	U
fddiv imm32, FS _m , FS _n		Δ	Δ	Δ	Δ	Δ	-	-	-	-
Flag change										
VF: This is "1" when the source data is sNaN, (±INF)/(±INF), or (±0)/(±0). This is "0" in all other cases. ZF: This is "1" when the dividend is ±0. This is "0" in all other cases. OF: This is "1" when the operation result is greater than the positive maximum value or smaller than the negative maximum value. This is "0" in all other cases. UF: This is "1" when the operation result is not "0" and is between ±2E _{min} . This is "0" in all other cases. IF: This is "1" when the operation result is inaccurate. This is "0" in all other cases. LF: This is not changed. GF: This is not changed. EF: This is not changed. UF: This is not changed.										

(1) When the FPU operation exception enable flag is "1"

Data Output		FS _m 2 / FS _m						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FS _m 1 / imm32	NORM	DIV	0	0	INF	INF	qNaN	-
	+0	-	-	-	+INF	-INF	qNaN	-
	-0	-	-	-	-INF	+INF	qNaN	-
	+INF	0	+0	-0	-	-	qNaN	-
	-INF	0	-0	+0	-	-	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-

EC Flag		FS _m 2 / FS _m						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FS _m 1 / imm32	NORM	DIV	0	0	0	0	0	V
	+0	Z	V	V	0	0	0	V
	-0	Z	V	V	0	0	0	V
	+INF	0	0	0	V	V	0	V
	-INF	0	0	0	V	V	0	V
	qNaN	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V

(1-1) DIV instruction

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-	O
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	* / -	0 / I
$0\text{x7f7fffff} < v$	-	O

(2) When the FPU operation exception enable flag is "0"

Data Output		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	DIV	0	0	INF	INF	qNaN	qNaN
	+0	INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	-0	INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	+INF	0	+0	-0	qNaN	qNaN	qNaN	qNaN
	-INF	0	-0	+0	qNaN	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

EC Flag		FSm2 / FSm						
		NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSm1 / imm32	NORM	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-INF	-
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 / +0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	*	-
$0\text{x7f7fffff} < v$	+INF	-

FMADD

Compound instructions of multiplication and addition of the floating-point data

FMADD FSm1, FSm2, FSm3, FS_n



Operation	((FSm1*FSm2) + FSm3) -> FSn										
	The result of multiplying the contents of the register (FSm1) by the contents of the register (FSm2) is added to the contents of the register (FSm3), and the result of addition is stored in the register (FSn). The purpose register is any of FS0-FS7.										
Assembler mnemonic	Note	EC flag					FCC flag				Size
		V	Z	O	U	I	L	G	E	U	
fmadd FSm1, FSm2, FSm3, FSn		Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change											
VF: This is "1" when the source data is any of the followings: sNaN, (±INF)*(±0), (+INF)+(-INF), (-INF)+(+INF), or (± INF)- (±INF). This is "0" in all other cases.											
ZF: This is always "0".											
OF: This is "1" when the operation result is greater than the positive maximum value and smaller than the negative maximum value. This is "0" in all other cases.											
UF: This is "1" when the operation result is not "0" and is between ±2Emin, and this is "0" in other cases.											
IF: This is "1" when the operation result is inaccurate, and this is "0" in all other cases.											
LF: This is not changed.											
GF: This is not changed.											
EF: This is not changed.											
UF: This is not changed.											



The purpose register is one of the FS0-FS7.

(1) The followings are shown when the FPU operational exception enable flag is "1".

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	-
	-NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	-
	+0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	-0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	+INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	-INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+0	+NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	-
	-NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	-
	+0	+0	+0	+0	+0	-	-	qNaN	-
	-0	+0	+0	+0	+0	-	-	qNaN	-
	+INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	-INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-0	+NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	-
	-NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	-
	+0	+0	-0	+0	-0	-	-	qNaN	-
	-0	-0	+0	-0	+0	-	-	qNaN	-

	+INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	-INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+INF	+NORM	+INF	+INF	+INF	+INF	+INF	-	qNaN	-
	-NORM	+INF	+INF	+INF	+INF	-	+INF	qNaN	-
	+0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	-0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	+INF	+INF	-	-	-	+INF	-	qNaN	-
	-INF	-	+INF	-	-	-	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-INF	+NORM	-INF	-INF	-INF	-INF	-	-INF	qNaN	-
	-NORM	-INF	-INF	-INF	-INF	-INF	-	qNaN	-
	+0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	-0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	+INF	-	-INF	-	-	-	-INF	qNaN	-
	-INF	-INF	-	-	-	-INF	-	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	0	qNaN	qNaN	qNaN	qNaN	-	-	qNaN	-
	INF	qNaN	qNaN	-	-	qNaN	qNaN	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
sNaN	all types	-	-	-	-	-	-	-	-

EC Flag		FSm2							
F3m3	F3m1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	0	0	0	V
	-NORM	MAC	MAC	MAC	MAC	0	0	0	V
	+0	MAC	MAC	MAC	MAC	V	V	0	V
	-0	MAC	MAC	MAC	MAC	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+0	+NORM	MAC	MAC	0	0	0	0	0	V
	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
-0	+NORM	MAC	MAC	0	0	0	0	0	V

	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+INF	+NORM	0	0	0	0	V	0	0	V
	-NORM	0	0	0	0	0	V	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	V	0	V	V	V	0	0	V
	-INF	0	V	V	V	0	V	0	V
	qNaN	0	0	0	0	0	0	0	V
-INF	sNaN	V	V	V	V	V	V	V	V
	+NORM	0	0	0	0	0	V	0	V
	-NORM	0	0	0	0	V	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	V	V	V	0	V	0	V
	-INF	V	0	V	V	V	0	0	V
qNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
	NORM	0	0	0	0	0	0	0	V
	0	0	0	0	0	V	V	0	V
	INF	0	0	V	V	0	0	0	V
sNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
sNaN	all Types	V	V	V	V	V	V	V	V

(1-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-	O
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	* / -	0 / I
$0\text{x7f7fffff} < v$	-	O

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	qNaN
	+0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN

	-0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+0	+NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	qNaN
	+0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	-0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-0	+NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	qNaN
	+0	-0	+0	-0	+0	qNaN	qNaN	qNaN	qNaN
	-0	+0	-0	+0	-0	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+INF	+NORM	+INF	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN
	-NORM	+INF	+INF	+INF	+INF	qNaN	+INF	qNaN	qNaN
	+0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN	qNaN
	-INF	+INF	qNaN	qNaN	qNaN	qNaN	+INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	+NORM	-INF	-INF	-INF	-INF	qNaN	-INF	qNaN	qNaN
	-NORM	-INF	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN
	+0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	-0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN	qNaN
	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	0	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	all types	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN



EC Flag		FSm2							
F3m3	F3m1	+NORM	-NORM		-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
+0	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
-0	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
+INF	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
-INF	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
qNaN	NORM	-	-		-	-	-	-	-
	0	-	-		-	-	-	-	-
	INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
sNaN	all types	-	-		-	-	-	-	-

(2-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-INF	-
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 /+0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	*	-
$0\text{x7f7fffff} < v$	+INF	-

FMSUB

Compound instructions of multiplication and subtraction of the floating-point data

FMSUB FSm1, FSm2, FSm3, FSn																
Operation		((FSm1*FSm2) - FSm3) -> FSn														
		The contents of the register (FSm3) is subtracted from the result of multiplying the contents of the register (FSm1) by the contents of the register (FSm2) , and the result of subtraction is stored in the register (FSn).														
The purpose register is		one of FS0-FS7.					EC flag					FCC flag				Size
Assembler mnemonic		Note					V	Z	O	U	I	L	G	E	U	
fmsub FSm1, FSm2, FSm3, FSn							Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change																
VF: This is "1" when the source data is any of the followings: sNaN, (±INF)*(±0), (+INF)+(-INF), (-INF)+(+INF), or (± INF)-(±INF). This is "0" in the other cases.																
ZF: This is always "0".																
OF: This is "1" when the operation result is greater than the positive maximum value and smaller than the negative maximum value. This is "0" in other cases.																
UF: This is "1" when the operation result is not "0" and is between ±2Emin, and this is "0" in other cases.																
IF: This is "1" when the operation result is inaccurate, and this is "0" in other cases.																
LF: This is not changed.																
GF: This is not changed.																
EF: This is not changed.																
UF: This is not changed.																



The purpose register is one of FS0-FS7.

(1) When the FPU operational exception enable flag is "1"

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	-
	-NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	-
	+0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	-0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	+INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	-INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+0	+NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	-
	-NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	-
	+0	+0	-0	+0	-0	-	-	qNaN	-
	-0	-0	+0	-0	+0	-	-	qNaN	-
	+INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	-INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-0	+NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	-
	-NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	-
	+0	+0	+0	+0	+0	-	-	qNaN	-
	-0	+0	+0	+0	+0	-	-	qNaN	-

	+INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	-INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+INF	+NORM	-INF	-INF	-INF	-INF	-	-INF	qNaN	-
	-NORM	-INF	-INF	-INF	-INF	-INF	-	qNaN	-
	+0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	-0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	+INF	-	-INF	-	-	-	-INF	qNaN	-
	-INF	-INF	-	-	-	-INF	-	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-INF	+NORM	+INF	+INF	+INF	+INF	+INF	-	qNaN	-
	-NORM	+INF	+INF	+INF	+INF	-	+INF	qNaN	-
	+0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	-0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	+INF	+INF	-	-	-	+INF	-	qNaN	-
	-INF	-	+INF	-	-	-	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	0	qNaN	qNaN	qNaN	qNaN	-	-	qNaN	-
	INF	qNaN	qNaN	-	-	qNaN	qNaN	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
sNaN	all types	-	-	-	-	-	-	-	-

EC Flag		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	0	0	0	V
	-NORM	MAC	MAC	MAC	MAC	0	0	0	V
	+0	MAC	MAC	MAC	MAC	V	V	0	V
	-0	MAC	MAC	MAC	MAC	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+0	+NORM	MAC	MAC	0	0	0	0	0	V
	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
-0	+NORM	MAC	MAC	0	0	0	0	0	V

	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+INF	+NORM	0	0	0	0	V	0	0	V
	-NORM	0	0	0	0	0	V	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	V	0	V	V	V	0	0	V
	-INF	0	V	V	V	0	V	0	V
	qNaN	0	0	0	0	0	0	0	V
-INF	sNaN	V	V	V	V	V	V	V	V
	+NORM	0	0	0	0	0	V	0	V
	-NORM	0	0	0	0	V	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	V	V	V	0	V	0	V
	-INF	V	0	V	V	V	0	0	V
qNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
	NORM	0	0	0	0	0	0	0	V
	0	0	0	0	0	V	V	0	V
	INF	0	0	V	V	0	0	0	V
sNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
sNaN	all Types	V	V	V	V	V	V	V	V

(1-1)MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-	O
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	* / -	0 / I
$0\text{x7f7fffff} < v$	-	O

(2) When the FPU operational exception enable flag is "0"

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	qNaN
	-NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	qNaN
	+0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN



	-0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	-INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+0	+NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	qNaN
	-NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	qNaN
	+0	+0	-0	+0	-0	qNaN	qNaN	qNaN	qNaN
	-0	-0	+0	-0	+0	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	-INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-0	+NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	qNaN
	+0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	+0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	-INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+INF	+NORM	-INF	-INF	-INF	-INF	qNaN	-INF	qNaN	qNaN
	-NORM	-INF	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN
	+0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	-0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	+INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN
	-INF	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	+NORM	+INF	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN
	-NORM	+INF	+INF	+INF	+INF	qNaN	+INF	qNaN	qNaN
	+0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN	qNaN
	-INF	qNaN	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	0	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	all types	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

EC Flag		FSm2							
FSm3	FSm1		-NORM		-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	+NORM	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
+0	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
-0	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
+INF	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
-INF	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
qNaN	NORM	-	-		-	-	-	-	-
	0	-	-		-	-	-	-	-
	INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
sNaN	all types	-	-		-	-	-	-	-

(2-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-INF	-
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 /+0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	*	-
$0\text{x7f7fffff} < v$	+INF	-

FNMADD Compound instructions of multiplication and addition of the floating-point data

FNMADD FSm1, FSm2, FSm3,FSn													
Operation	(-FSm1*FSm2) + FSm3) -> FSn												
	The contents of the register (FSm3) is added to the result of multiplying the contents of the register (FSm1), the contents of the register (FSm2) and "-1" by each other, and the result of addition is stored in the register (FSn). The purpose register is one of FS0-FS7.												
Assembler mnemonic		Note		EC flag					FCC flag				Size
				V	Z	O	U	I	L	G	E	U	
fnmadd FSm1, FSm2, FSm3, FSn				Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change													
VF: This is "1" when the source data is any of the followings: sNaN, (±INF)*(±0), (+INF)+(-INF), (-INF)+(+INF), or (± INF)- (±INF). This is "0" in allother cases.													
ZF: This is always "0".													
OF: This is "1" when the operation result is greater than the positive maximum value and smaller than the negative maximum value. This is "0" in all other cases.													
UF: This is "1" when the operation result is not "0" and is between ±2Emin, and this is "0" in all other cases.													
IF: This is "1" when the operation result is inaccurate, and this is "0" in all other cases.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													



The purpose register is one of the FS0-FS7.

(1) When the FPU operational exception enable flag is "1"

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	-
	-NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	-
	+0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	-0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	+INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	-INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+0	+NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	-
	-NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	-
	+0	+0	+0	+0	+0	-	-	qNaN	-
	-0	+0	+0	+0	+0	-	-	qNaN	-
	+INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	-INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-0	+NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	-
	-NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	-
	+0	-0	+0	-0	+0	-	-	qNaN	-
	-0	+0	-0	+0	-0	-	-	qNaN	-

	+INF	-INF	+INF	-	-	+INF	-INF	qNaN	-
	-INF	+INF	-INF	-	-	-INF	+INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+INF	+NORM	+INF	+INF	+INF	+INF	-	+INF	qNaN	-
	-NORM	+INF	+INF	+INF	+INF	+INF	-	qNaN	-
	+0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	-0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	+INF	-	+INF	-	-	-	+INF	qNaN	-
	-INF	+INF	-	-	-	+INF	-	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-INF	+NORM	-INF	-INF	-INF	-INF	-INF	-	qNaN	-
	-NORM	-INF	-INF	-INF	-INF	-	-INF	qNaN	-
	+0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	-0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	+INF	-INF	-	-	-	-INF	-	qNaN	-
	-INF	-	-INF	-	-	-	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	0	qNaN	qNaN	qNaN	qNaN	-	-	qNaN	-
	INF	qNaN	qNaN	-	-	qNaN	qNaN	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
sNaN	all types	-	-	-	-	-	-	-	-

EC Flag		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	0	0	0	V
	-NORM	MAC	MAC	MAC	MAC	0	0	0	V
	+0	MAC	MAC	MAC	MAC	V	V	0	V
	-0	MAC	MAC	MAC	MAC	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+0	+NORM	MAC	MAC	0	0	0	0	0	V
	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
-0	+NORM	MAC	MAC	0	0	0	0	0	V

	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+INF	+NORM	0	0	0	0	V	0	0	V
	-NORM	0	0	0	0	0	V	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	V	0	V	V	V	0	0	V
	-INF	0	V	V	V	0	V	0	V
	qNaN	0	0	0	0	0	0	0	V
-INF	sNaN	V	V	V	V	V	V	V	V
	+NORM	0	0	0	0	0	V	0	V
	-NORM	0	0	0	0	V	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	V	V	V	0	V	0	V
	-INF	V	0	V	V	V	0	0	V
qNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
	NORM	0	0	0	0	0	0	0	V
	0	0	0	0	0	V	V	0	V
	INF	0	0	V	V	0	0	0	V
sNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
sNaN	all Types	V	V	V	V	V	V	V	V

(1-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-	O
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	* / -	0 / I
$0\text{x7f7fffff} < v$	-	O

(2) When the FPU operational exception enable flag is "0"

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	qNaN
	+0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN



	-0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+0	+NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	qNaN
	+0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	-0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-0	+NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	qNaN
	+0	-0	+0	-0	+0	qNaN	qNaN	qNaN	qNaN
	-0	+0	-0	+0	-0	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+INF	+NORM	+INF	+INF	+INF	+INF	qNaN	+INF	qNaN	qNaN
	-NORM	+INF	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN
	+0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	qNaN	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN
	-INF	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	+NORM	-INF	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN
	-NORM	-INF	-INF	-INF	-INF	qNaN	-INF	qNaN	qNaN
	+0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	-0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN	qNaN
	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	0	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	all types	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

EC Flag		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	-	-	-	-	-	-	-	-
	-NORM	-	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-	-
+0	+NORM	-	-	-	-	-	-	-	-
	-NORM	-	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-	-
-0	+NORM	-	-	-	-	-	-	-	-
	-NORM	-	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-	-
+INF	+NORM	-	-	-	-	-	-	-	-
	-NORM	-	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-	-
-INF	+NORM	-	-	-	-	-	-	-	-
	-NORM	-	-	-	-	-	-	-	-
	+0	-	-	-	-	-	-	-	-
	-0	-	-	-	-	-	-	-	-
	+INF	-	-	-	-	-	-	-	-
	-INF	-	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-	-
qNaN	NORM	-	-	-	-	-	-	-	-
	0	-	-	-	-	-	-	-	-
	INF	-	-	-	-	-	-	-	-
	qNaN	-	-	-	-	-	-	-	-
	sNaN	-	-	-	-	-	-	-	-
sNaN	all types	-	-	-	-	-	-	-	-

(2-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7fffff}$	-INF	-
$0\text{xff7fffff} \leq v \leq 0\text{x80800000}$	*	-
$0\text{x80800000} < v < 0$	-0	-
0	-0 /+0	-
$0 < v < 0\text{x00800000}$	+0	-
$0\text{x00800000} \leq v \leq 0\text{x7f7fffff}$	*	-
$0\text{x7f7fffff} < v$	+INF	-

FNMSUB Compound instructions of multiplication and addition of the floating-point data

FNMSUB FSm1, FSm2, FSm3,FSn															
Operation		(-FSm1*FSm2) - FSm3) -> FSn													
		The contents of the register (FSm3) is subtracted from the result of multiplying the contents of the register (FSm1), the contents of the register (FSm2) and "-1" by each other, and the result of subtraction is stored in the register (FSn).													
The purpose register is one of FS0-FS7.		Assembler mnemonic		Note		EC flag					FCC flag				Size
						V	Z	O	U	I	L	G	E	U	
fnmsub FSm1, FSm2, FSm3, FSn						Δ	0	Δ	Δ	Δ	-	-	-	-	4
Flag change															
VF: This is "1" when the source data is any of the followings: sNaN, (±INF)*(±0), (+INF)+(-INF), (-INF)+(+INF), or (± INF)- (±INF). This is "0" in all other cases.															
ZF: This is always "0".															
OF: This is "1" when the operation result is greater than the positive maximum value and smaller than the negative maximum value. This is "0" in all other cases.															
UF: This is "1" when the operation result is not "0" and is between ±2Emin, and this is "0" in all other cases.															
IF: This is "1" when the operation result is inaccurate, and this is "0" in all other cases.															
LF: This is not changed.															
GF: This is not changed.															
EF: This is not changed.															
UF: This is not changed.															



The purpose register is one of FS0-FS7.

(1) When the FPU operational exception enable flag is "1"

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	-
	-NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	-
	+0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	-0	MAC	MAC	MAC	MAC	-	-	qNaN	-
	+INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	-INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+0	+NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	-
	-NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	-
	+0	-0	+0	+0	+0	-	-	qNaN	-
	-0	+0	-0	+0	+0	-	-	qNaN	-
	+INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	-INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-0	+NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	-
	-NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	-
	+0	+0	+0	-0	+0	-	-	qNaN	-
	-0	+0	+0	+0	-0	-	-	qNaN	-

	+INF	-INF	+INF	-	-	-INF	+INF	qNaN	-
	-INF	+INF	-INF	-	-	+INF	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
+INF	+NORM	-INF	-INF	-INF	-INF	-INF	-	qNaN	-
	-NORM	-INF	-INF	-INF	-INF	-	-INF	qNaN	-
	+0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	-0	-INF	-INF	-INF	-INF	-	-	qNaN	-
	+INF	-INF	-	-	-	-INF	-	qNaN	-
	-INF	-	-INF	-	-	-	-INF	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
-INF	+NORM	+INF	+INF	+INF	+INF	-	+INF	qNaN	-
	-NORM	+INF	+INF	+INF	+INF	+INF	-	qNaN	-
	+0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	-0	+INF	+INF	+INF	+INF	-	-	qNaN	-
	+INF	-	+INF	-	-	-	+INF	qNaN	-
	-INF	+INF	-	-	-	+INF	-	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	0	qNaN	qNaN	qNaN	qNaN	-	-	qNaN	-
	INF	qNaN	qNaN	-	-	qNaN	qNaN	qNaN	-
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	-
	sNaN	-	-	-	-	-	-	-	-
sNaN	all types	-	-	-	-	-	-	-	-

EC Flag		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	0	0	0	V
	-NORM	MAC	MAC	MAC	MAC	0	0	0	V
	+0	MAC	MAC	MAC	MAC	V	V	0	V
	-0	MAC	MAC0	MAC	MAC	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+0	+NORM	MAC	MAC	0	0	0	0	0	V
	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
-0	+NORM	MAC	MAC	0	0	0	0	0	V

	-NORM	MAC	MAC	0	0	0	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	0	V	V	0	0	0	V
	-INF	0	0	V	V	0	0	0	V
	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
+INF	+NORM	0	0	0	0	0	V	0	V
	-NORM	0	0	0	0	V	0	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	0	V	V	V	0	V	0	V
	-INF	V	0	V	V	V	0	0	V
	qNaN	0	0	0	0	0	0	0	V
-INF	sNaN	V	V	V	V	V	V	V	V
	+NORM	0	0	0	0	V	0	0	V
	-NORM	0	0	0	0	0	V	0	V
	+0	0	0	0	0	V	V	0	V
	-0	0	0	0	0	V	V	0	V
	+INF	V	0	V	V	V	0	0	V
	-INF	0	V	V	V	0	V	0	V
qNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
	NORM	0	0	0	0	0	0	0	V
	0	0	0	0	0	V	V	0	V
	INF	0	0	V	V	0	0	0	V
sNaN	qNaN	0	0	0	0	0	0	0	V
	sNaN	V	V	V	V	V	V	V	V
	all Types	V	V	V	V	V	V	V	V

(1-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff7ffff}$	-	O
$0\text{xff7ffff} \leq v \leq 0\text{x80800000}$	* / -	0 / I
$0\text{x80800000} < v < 0$	-	U
0	+0 / -0	0
$0 < v < 0\text{x00800000}$	-	U
$0\text{x00800000} \leq v \leq 0\text{x7f7ffff}$	* / -	0 / I
$0\text{x7f7ffff} < v$	-	O

(2) When the FPU operational exception enable flag is "1"

Data Output		FSm2							
FSm3	FSm1	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	MAC	MAC	MAC	MAC	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	MAC	MAC	+INF	-INF	qNaN	qNaN
	+0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN

	-0	MAC	MAC	MAC	MAC	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+0	+NORM	MAC	MAC	-0	+0	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	-0	+INF	-INF	qNaN	qNaN
	+0	-0	+0	-0	+0	qNaN	qNaN	qNaN	qNaN
	-0	+0	-0	+0	-0	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-0	+NORM	MAC	MAC	+0	+0	-INF	+INF	qNaN	qNaN
	-NORM	MAC	MAC	+0	+0	+INF	-INF	qNaN	qNaN
	+0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	-0	+0	+0	+0	+0	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	+INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
	-INF	+INF	-INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
+INF	+NORM	-INF	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN
	-NORM	-INF	-INF	-INF	-INF	qNaN	-INF	qNaN	qNaN
	+0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	-0	-INF	-INF	-INF	-INF	qNaN	qNaN	qNaN	qNaN
	+INF	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN	qNaN
	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	+NORM	+INF	+INF	+INF	+INF	qNaN	+INF	qNaN	qNaN
	-NORM	+INF	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN
	+0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-0	+INF	+INF	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	qNaN	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN
	-INF	+INF	qNaN	qNaN	qNaN	+INF	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
qNaN	NORM	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	0	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	all types	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

EC Flag		FSm2							
FSm3	FSm1	+NORM	-NORM		-0	+INF	-INF	qNaN	sNaN
NORM	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
+0	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
-0	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
+INF	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
-INF	+NORM	-	-		-	-	-	-	-
	-NORM	-	-		-	-	-	-	-
	+0	-	-		-	-	-	-	-
	-0	-	-		-	-	-	-	-
	+INF	-	-		-	-	-	-	-
	-INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
qNaN	NORM	-	-		-	-	-	-	-
	0	-	-		-	-	-	-	-
	INF	-	-		-	-	-	-	-
	qNaN	-	-		-	-	-	-	-
	sNaN	-	-		-	-	-	-	-
sNaN	all types	-	-		-	-	-	-	-

(2-1) MAC

Calculation result	Data Output	EC Flag
$v < 0\text{xff}7\text{ffff}$	-INF	-
$0\text{xff}7\text{ffff} \leq v \leq 0\text{x}80800000$	*	-
$0\text{x}80800000 < v < 0$	-0	-
0	-0 /+0	-
$0 < v < 0\text{x}00800000$	+0	-
$0\text{x}00800000 \leq v \leq 0\text{x}7\text{f}7\text{ffff}$	*	-
$0\text{x}7\text{f}7\text{ffff} < v$	+INF	-

FBCC

Conditional branch under the floating-point condition

FBCC (d8, PC)



Operation	<p>When branch is taken, PC (current instruction address) + (sign_ext) d8 -> nPC (next instruction PC) 8-bit displacement (d8) is sign-extended and added to the PC, and the result is written into the program counter of the next instruction. (nPC). Even if the addition result overflows, this overflow is ignored and the result is written into the PC.</p> <p>When branch is not taken, The next instruction is executed. PC (current instruction address) + CodeSize -> nPC (next instruction PC) The next instruction is executed.</p>
-----------	---



Assembler mnemonic	Note	EC flag					FCC flag				Size
		V	Z	O	U	I	L	G	E	U	
fbeq label	E=1	-	-	-	-	-	-	-	-	-	3
fbne label	U=1 or L=1 or G=1	-	-	-	-	-	-	-	-	-	3
fbgt label	G=1	-	-	-	-	-	-	-	-	-	3
fbge label	G=1 or E=1	-	-	-	-	-	-	-	-	-	3
fblt label	L=1	-	-	-	-	-	-	-	-	-	3
fble label	L=1 or E=1	-	-	-	-	-	-	-	-	-	3
fbuo label	U=1	-	-	-	-	-	-	-	-	-	3
fblg label	L=1 or G=1	-	-	-	-	-	-	-	-	-	3
fbleg label	L=1 or E=1 or G=1	-	-	-	-	-	-	-	-	-	3
fbug label	U=1 or G=1	-	-	-	-	-	-	-	-	-	3
fbuge label	U=1 or G=1 or E=1	-	-	-	-	-	-	-	-	-	3
fbul label	U=1 or L=1	-	-	-	-	-	-	-	-	-	3
fbule label	U=1 or L=1 or E=1	-	-	-	-	-	-	-	-	-	3
fbue label	U=1 or E=1	-	-	-	-	-	-	-	-	-	3

Flag change

VF: This is not changed.
ZF: This is not changed.
OF: This is not changed.
UF: This is not changed.
IF: This is not changed.
LF: This is not changed.
GF: This is not changed.
EF: This is not changed.
UF: This is not changed.

FLCC

Conditional branch only for the loop
under the floating-point condition

FLCC													
Operation	When conditional branch is taken												
	LAR - 4 -> nPC (next instruction PC)												
	The instruction loaded from the loop instruction register (LIR) is executed and the instruction fetch starts for the address loaded to the loop address register (LAR).												
	At the same time, 4 is subtracted from the instruction fetch address register (LAR) and the result is written into the program counter (PC).												
	Even if the subtract result overflows, this overflow is ignored and the result is written into the program counter (PC). Without coordination with SETLB, the operation cannot be guaranteed. This is used with the SETLB instruction in order to speed up the loop control, and conditionally branches to the start of the loop registered in the SETLB instruction.												
	When branch is not taken												
	PC (current instruction address) + 1 -> nPC (next instruction PC)												
	The next instruction is executed.												
Assembler mnemonic		Note	EC flag					FCC flag				Size	
			V	Z	O	U	I	L	G	E	U		
fleq		E=1	-	-	-	-	-	-	-	-	-	2	
flne		U=1 or L=1 or G=1	-	-	-	-	-	-	-	-	-	2	
flgt		G=1	-	-	-	-	-	-	-	-	-	2	
flge		G=1 or E=1	-	-	-	-	-	-	-	-	-	2	
fllt		L=1	-	-	-	-	-	-	-	-	-	2	
file		L=1 or E=1	-	-	-	-	-	-	-	-	-	2	
fluo		U=1	-	-	-	-	-	-	-	-	-	2	
flg		L=1 or G=1	-	-	-	-	-	-	-	-	-	2	
flleg		L=1 or E=1 or G=1	-	-	-	-	-	-	-	-	-	2	
flug		U=1 or G=1	-	-	-	-	-	-	-	-	-	2	
fluge		U=1 or G=1 or E=1	-	-	-	-	-	-	-	-	-	2	
flul		U=1 or L=1	-	-	-	-	-	-	-	-	-	2	
flule		U=1 or L=1 or E=1	-	-	-	-	-	-	-	-	-	2	
flue		U=1 or E=1	-	-	-	-	-	-	-	-	-	2	
Flag change													
VF: This is not changed.													
ZF: This is not changed.													
OF: This is not changed.													
UF: This is not changed.													
IF: This is not changed.													
LF: This is not changed.													
GF: This is not changed.													
EF: This is not changed.													
UF: This is not changed.													

Chapter 3

Directions for using instructions

3

Cautions for programming

The 32-bit microcontroller MN103E series has accomplished speedup through the following devices concerning implementations.

- Speedup of the instruction execution cycle

This has accomplished high-speed execution for Lcc/SETLB/RETF through performing the 5-step pipeline processing as well as performing the normal pipeline processing and a different processing with specialized software.

- Speedup of operational frequencies

This has accomplished a speedup of operation frequencies through performing the optimized allocation of the pipeline stage for processing preventing the improvement of operation frequencies such as flag generation and alignment/extension of load data.

The description of the pipeline architecture, the cautions for programming about a part of the instruction description, and the recommendations are shown here for the effective use of the above-mentioned speed-up methods.

1. Pipeline architecture

This describes the pipeline configuration adopted in the 32-bit microcontroller MN103E series and the operations.

2. Cautions on the instruction description

The cautions are for programming about the instruction description, instruction design and combination. Please take note that malfunctions may be caused if these cautions are not followed.

3. Recommendations on the instruction description

The recommendations are for the instruction description, instruction design and combination. Malfunctions are not caused even if the recommended requirements are not fulfilled. However, please take note that the increase in the number of execution cycles may be caused if the requirements are not fulfilled.

4. General on the instruction description

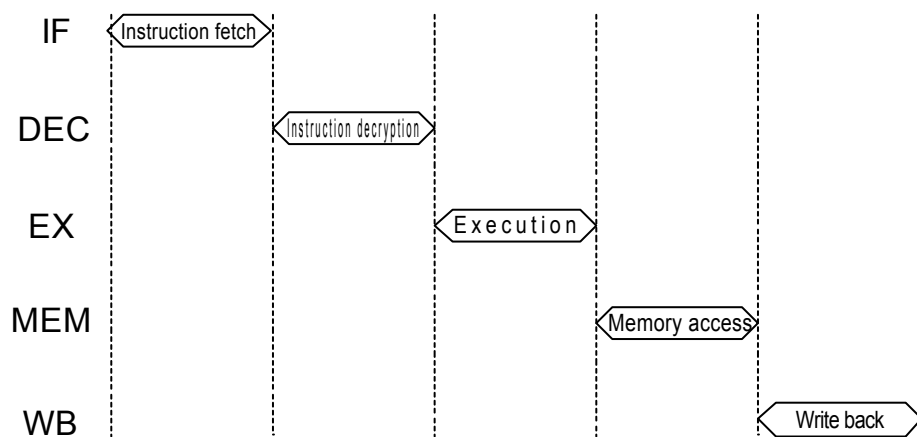
The general shows the examples of generally-used programming.

1 Pipeline architecture

The 32-bit microcontroller MN103E series performs pipeline processing for executing instructions in order to accomplish high-speed operations. Multiple instructions can be overlapped and parallel-executed through the pipeline processing.

1-1 Pipeline operation

The 32-bit microcontroller MN103E series executes instructions at the following five-step pipeline stage.



IF (Instruction fetch)	: This takes in instructions from the memory.
DEC (Instruction decryption)	: This decrypts the instructions taken in. This performs the address calculation of the branch target in a part of the branch instructions.
EX (Execution)	: This performs the operations or address calculation based on the decryption results.
MEM (Memory access)	: This accesses memory in the instructions with memory access, and generates and stores flags in the instructions with flag change.
WB (Write back)	: This stores operation results in registers. This aligns, extends and stores the memory access results in registers.

This sets an instruction queue between the IF and DEC stages, precedently stores instructions from memory to the take-in instruction queue. If there are the data necessary to decryption and execution in the instruction queue, the instruction decryption stage is started. When the instruction queue is empty as immediately after branching or it does not have necessary instruction data, The instruction queue wait of at least one cycle occurs.

The instruction queue is controlled by hardware, and its function is different depending on core types even in the MN103E series. Therefore, this chapter describes the cases that there are necessary data in the instruction queue. Although any special care is unnecessary at making programs, it is necessary to understand instruction queue operations of each core at calculating the instruction execution cycle.

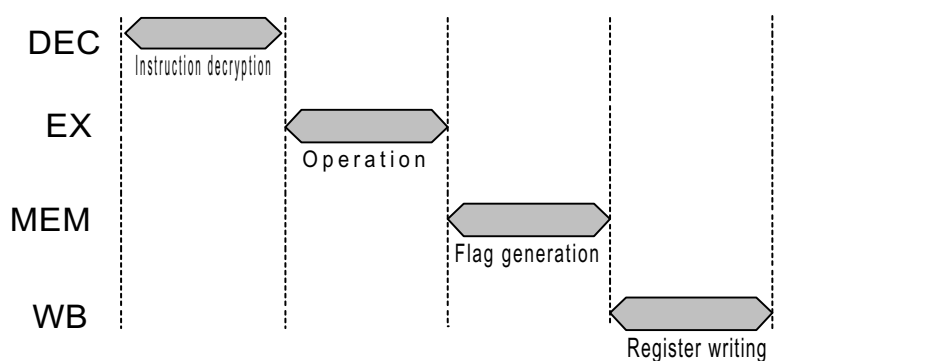
1-2 Pipeline operations of operations between registers

Each pipeline stage of the operation between registers performs the following operations.

DEC	: This decrypts instructions.
EX	: This executes operations based on the decrypted results.
MEM	: This updates flags based on the operation result in the instruction with flag change.
WB	: This stores the operation result in registers.

The instructions performing this pipeline operation are as follows:

addition and subtraction/ logical operation/ shift instruction/ data transfer instruction between CPU registers/ data transfer instruction between CPU and FPU registers/ data transfer instruction between FPU registers.



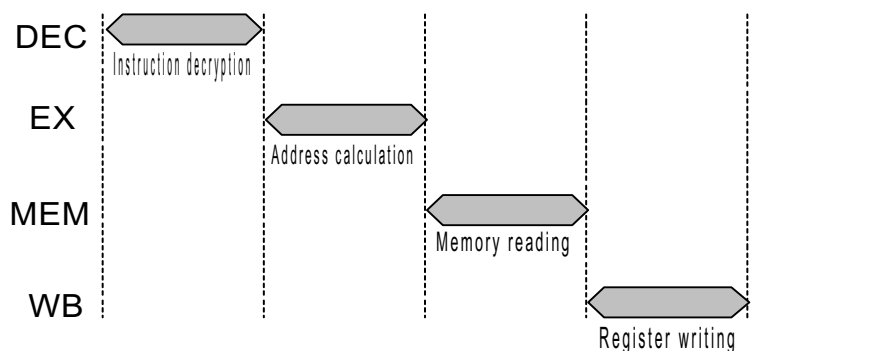
1-3 Pipeline operation of data load

Each pipeline stage of the data load performs the following operations.

DEC	: This decrypts instructions.
EX	: This executes address calculation based on the decrypted result and at the same time, determines the memory space for memory access.
MEM	: This reads TLB if necessary, and reads data from memory through the changed address.
WB	: This stores the read data in registers.

The instructions performing this pipeline operation are as follows:

data transfer instruction between memory and CPU registers/ data transfer instruction between memory and FPU registers.



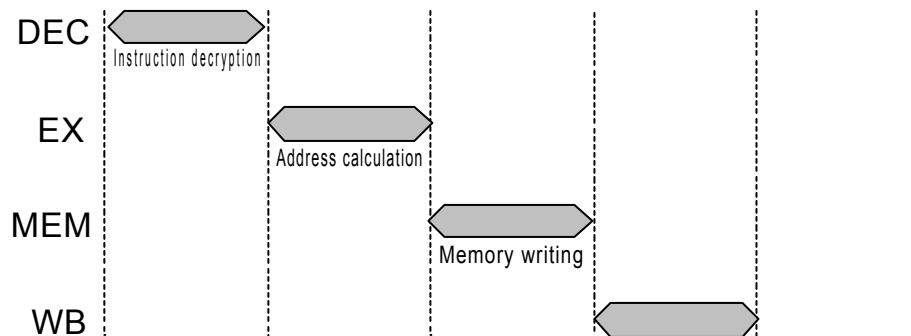
1-4 Pipeline operations of data store

Each pipeline stage of data store performs the following operations.

DEC	: This decrypts instructions.
EX	: This executes address calculation based on the decrypted result.
MEM	: This writes data into memory between the memory stages.
WB	: This does not perform this processing.

The instructions performing this pipeline operation are as follows:

data transfer instruction between memory and CPU registers/ data transfer instruction between memory and FPU registers.

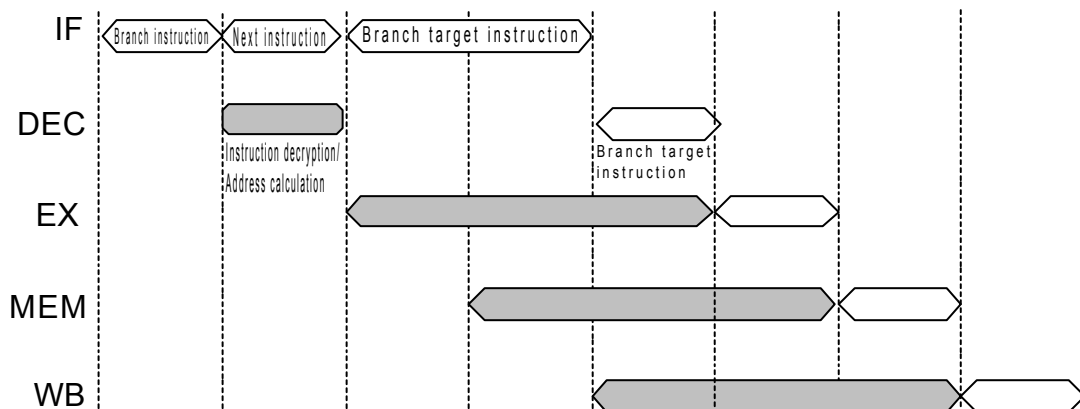


1-5 Branch pipeline operations

Each pipeline stage of the branch instruction performs the following operations.

DEC	: This decrypts instructions, and at the same time performs address calculation for the branch target. This starts reading the branch instruction based on the calculation result after the next cycle.
EX	: This does not perform this processing.
MEM	: This does not perform this processing.
WB	: This does not perform this processing.

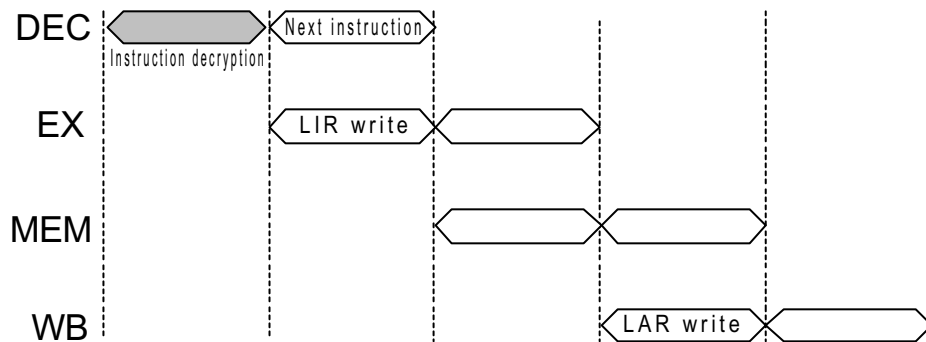
The instructions performing this pipeline operation are conditional branch instructions and others. The following figure shows the pipeline operations when branching occurs.



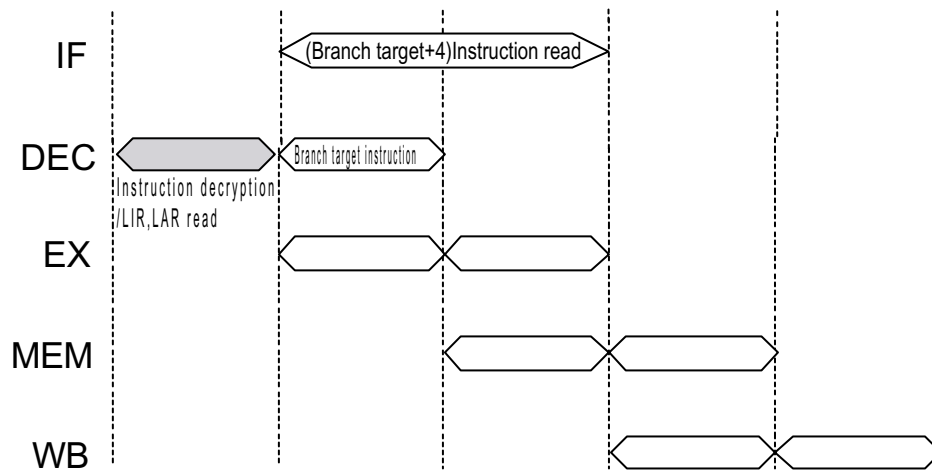
1-6 Pipeline operations of SETLB and LCC

Since the instructions are executed at high speed, the processing is not carried out in the order of IF stage/ DEC stage/ EX stage/ MEM stage/ WB stage as seen in the following pipeline operations of the instructions, and the instructions are executed by using specialized hardware.

SETLB pipeline operations



LCC pipeline operations



1-7 Number of instruction executing cycles

This chapter describes the number of the instruction executing cycle in the MN103E series.

This instruction manual has the tables of throughput and latency in the appendix. Throughput shows the minimum number of the cycles necessary to execute each instruction. Latency shows the number of the cycles that the subsequent instruction execution has to wait after the cycle for executing each instruction when each instruction and its subsequent instruction are dependent on each other.

There are the following three cases concerning the dependence between instructions.

(1) Register dependence between instructions

When the destination register of the preceding instruction (the storing-target register of the operation result) matches the source register of the subsequent instruction.

(2) Flag dependence between instructions

When the subsequent instruction refers to the flag with a possibility of being changed by the preceding instruction.

(3) When the FPU instruction is subsequent to the CPU load/store instruction.

1-7-1 No dependence between instructions

The number of executing cycles for each instruction is shown as THROUGHPUT in another table (MN103E series Throughput & Latency). Some instructions need multiple cycles for the instruction decode, operation, and memory address.

[Examples] The case of no dependence between instructions

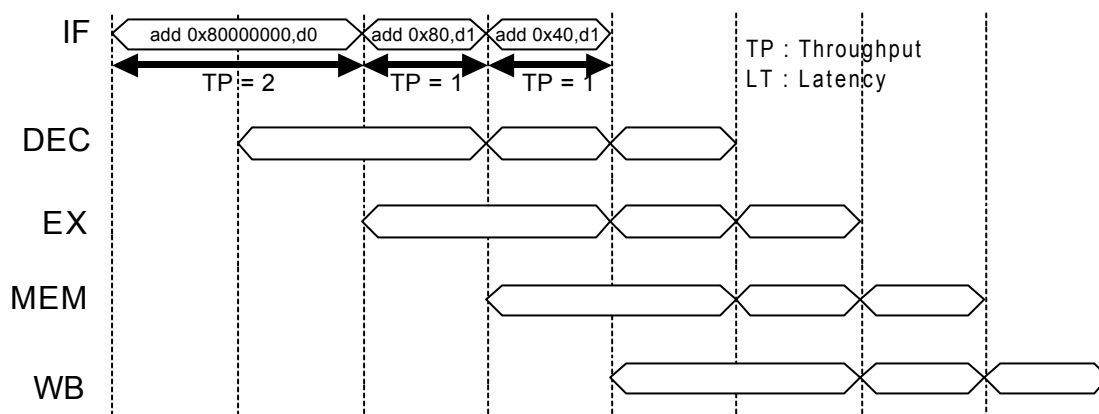
[Instruction strings to be executed]

```

add    0x80000000,d0 ..... ( A )
add    0x80,d0 .....      ( B )
add    0x40,d1 .....      ( C )

```

In the above examples, THROUGHPUT of instructions (A), (B), and (C) are 2/1/1 respectively, and LATENCY of them are 2/1/1 respectively. The pipeline operation in this example is shown as the following figure. The number of instruction executing cycles is determined depending on the number of the THROUGHPUT cycles because there is no dependence between the instructions in this example.



1-7-2 Register dependence between instructions

When the destination register (the storing target register of the operation result) of the preceding instruction is the same as the source register of the subsequent instruction, the subsequent instruction cannot be executed until the preceding instruction stores the result. The number of executing cycles for each instruction is shown as THROUGHPUT in another table (MN103E series Throughput & Latency).

[Examples] The case that there is register dependence between instructions

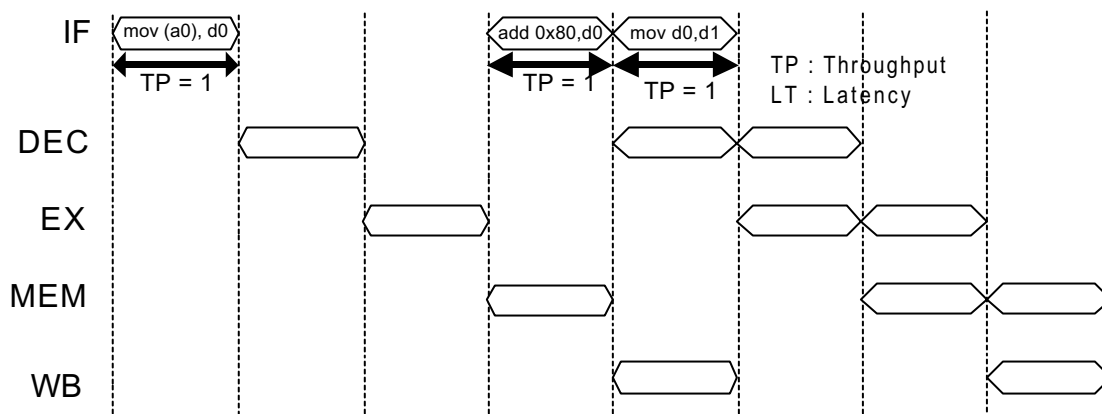
[Instruction strings to be executed]

```

mov    (a0), d0 ..... ( A )
add    0x80, d0 ..... ( B )
mov    d0,d1 ..... ( C )

```

In the above examples, THROUGHPUT of instructions (A), (B), and (C) are 1/1/1 respectively, and LATENCY of them are 3/1/1 respectively. The pipeline operation in this example is shown as the following figure. The number of instruction executing cycles of (B) is determined depending on the number of (A) latency cycles and the number of the (B) and (C) THROUGHPUT cycles because there is register dependence between (A) and (B).

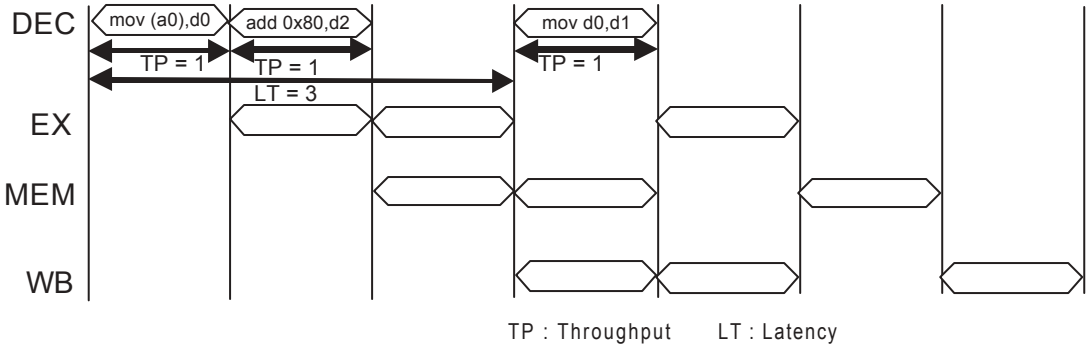


The following example shows that there is register dependence in the preceding instruction after two instructions.

[Examples] The case that there is register dependence between instructions

[Instruction strings to be executed]	
mov	(a0), d0 (A)
add	0x80, d2 (B)
mov	d0,d1 (C)

In the above examples, THROUGHPUT of instructions (A), (B), and (C) are 1/1/1 respectively, and LATENCY of them are 3/1/1 respectively. The pipeline operation in this example is shown as the following figure. Instruction (B) of (A) THROUGHPUT number can be executed because there is no register dependence between (A) and (B). On the other hand, (C) cannot be executed until the numbers of (B) THROUGHPUT cycles and (A) LATENCY cycles because there is register dependence between (A) and (C).



1-7-3 Flag dependence between instructions

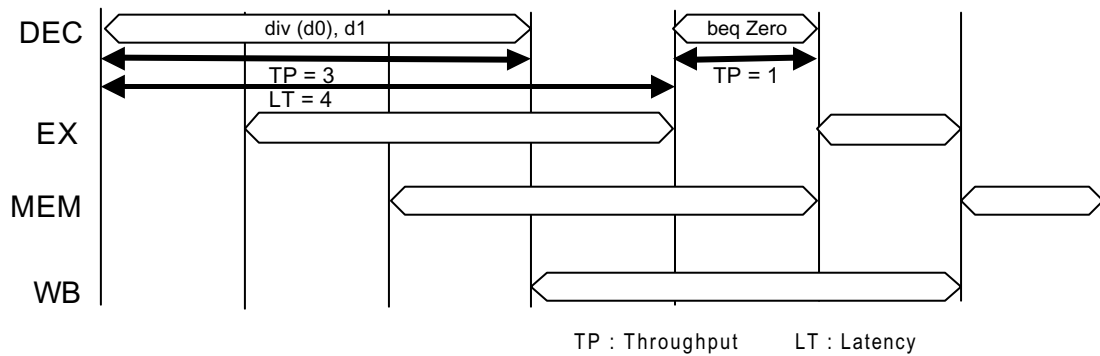
The subsequent instruction cannot be executed until flag updating by the preceding instruction when the subsequent instruction refers to the flag state of TPSW and FPCR updated by the preceding instruction.

[Example] Example of flag independence between instructions

[Instruction strings to be executed]

```
div    d0, d1 ..... ( A )
beq    zero .....   ( B )
```

In the above example, (A) THROUGHPUT is 2, (A) LATENCY is 4, and (B) THROUGHPUT is 3. (B) refers to the flag updated by the operation of (A). Accordingly (B) is executed until the number of the (A) LATENCY cycles.



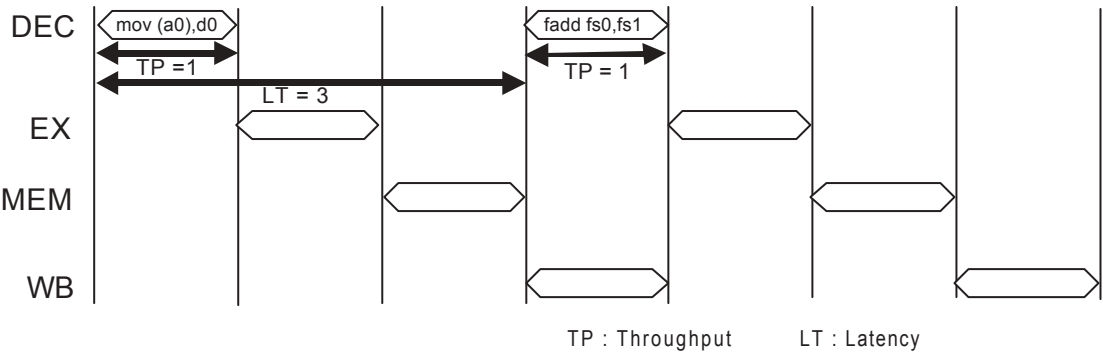
1-7-4 When the FPU instruction is subsequent to CPU load/store instruction

When the preceding instruction is the CPU load or store instruction, the following instruction cannot be executed until CPU completed the load/store instruction, even though there is no register and flag dependence between instructions.

[Example] Example of FPU following CPU load storing instruction

[Instruction strings to be executed]			
div	d0, d1	(A)
beq	zero	(B)

In the above example, (A) THROUGHPUT is 2, (A) LATENCY is 4, and (B) THROUGHPUT is 3. (B) refers to the flag updated by the operation of (A). Accordingly (B) is executed until the number of the (A) LATENCY cycles.



2

Cautions on the instruction description

The cautions are not on the programming for instruction description, assignment, and combination.
The next section shows "Recommendations on the instruction description".

3 Recommendations on Instruction description

The recommendations are for the instruction description, combination and instruction assignment. For the instruction description of this recommendations, meeting the requirement is not essential, however, when not meeting the requirement, please take note that the increase of the executing cycle number may be caused. The recommendations on instruction description consists of the following parts.

[Contents]

It describes the recommended contents.

For some recommended parts, it shows pipeline operation figures.

[Classification]



High-speed processing

High-speed processing

This is recommended in order to execute instructions at high speed through not causing fluctuation of pipeline operations.

[Examples]

It describes the programming examples in the assembly language and the explanation about that.

[General example of description]

This is an example of programming without considering pipeline fluctuation by the following instruction.

[Recommended example of description]

This is an example of executing instructions at high speed with considering in order not to occur pipeline fluctuation.

[Applicable instructions]

This describes the applicable instructions for recommendations.



In the programming examples, when inserting multiple cycles between instructions, it is not necessary to be INC instruction for easily understanding. An instruction with no dependence with the instructions around inserting can be inserted instead of INC.

Preceding instruction	Following instruction	Assignment	Recommendations	Position
All instructions	-	-	The instructions have two 8-byte instruction buffers, and assigns instructions not as to causing a lack of instruction supply. Multiple cycles is necessary for the decoding of over-4-byte instructions in order to decode instructions per 4-byte unit by the instruction decoder. An instruction with a large instruction length should not be used as much as possible.	
LCC instruction		Branch target	Branch target instruction assigns an instruction that m of $8n-4+m$ has a small address, short length, and small instruction consumption per cycle.	3-1 (1)
Branch instructions except for LCC instruction			Branch target instruction assigns an instruction that m of $8n+m$ has a small address, short length, and small instruction consumption per cycle.	3-1 (2)
BCC/LCC instruction		Subsequent	Non-branch side assigns an instruction that has short length and small instruction consumption per cycle.	3-1 (3)
Instructions except for branch instructions			This averages amount of instruction consumption.	3-1 (4)
MOVM (SP), regs	SETLB Instruction	Subsequent	1 cycle insertion	3-2
RET/RETF		Return target		
MOV Dm, MD EXT instruction MUL/MULU	RETF Instruction	Subsequent	3 cycle insertion	3-3 (1)
DIV/DIVU		Subsequent		3-3
CALL/CALLS		Branch target	2 cycle insertion 1 cycle insertion	(2)
MOVM (SP), regs		Subsequent	1 cycle insertion	3-3
RET/RETF		Return target		(3)
CALL/CALLS	MOV MDR, Dn DIV/DIVU	Branch target	2 cycle insertion	3-4

3-1 Instruction assignment subsequent to branch instruction

(1) Assignment of branch target instruction in Lcc instruction

[Contents]



High-speed processing

The instruction fetch is carried out by a unit of the 8-byte boundary, and 4-byte stores the branch instruction of Lcc instruction in LIR. Therefore, in the case of Lcc, m of $[8n-4+m]$ (m,n are integers from 0 to 7) locates an instruction length with short instruction length and the long number of execution cycle at a small address.

[Example]

align	8	
mov	0x00,a0	
clr	d0	
setlb		
mov	d0,(a0)	<- assigned at 8n-4
mov	d0,(0x10,a0)	
mov	d0, (0x20,a0)	<- assigned at 8n
mov	d0, (0x30,a0)	
inc	a0	
cmp	0x10,a0	
llt		

The above program is an example when the branch target of LLT is assigned at 8n-4.

As the result of this assignment, mov d0, (a0) and mov d0, (0x10, a0) are stored in LIR, and the address 8n is stored in LAR. Therefore, during executing the instructions stored in LIR, CPU core starts a fetch from the beginning of the 8-byte boundary, and can fetch the maximum 8-byte instruction.

[Applicable instruction]

<Preceding instruction>	Lcc instruction
<Branch target instruction>	All instructions



(2) Assignment of branch target instruction in the branch instructions other than LCC

[Contents]

The instruction fetch is carried out by a unit of the 8-byte boundary. Therefore, in the case of the branch instructions except for Lcc, m of $[8n-4+m]$ (m,n are integers from 0 to 7) locates an instruction length with long instruction length and the long number of execution cycle at a small address.

[Example]

	align	8	
	nop		
	nop		
	nop		
	nop		
	nop		
	mov	0x00,a0	
	clr	d0	
LABEL	mov	d0, (a0)	<- assigned at 8n
	mov	d0,(0x10,a0)	
	mov	d0, (0x20,a0)	
	mov	d0, (0x30,a0)	
	inc	a0	
	cmp	0x10,a0	
	blt	LABEL	

The above program is an example when the branch target of LLT is assigned at $8n-4$.

As the result of this assignment, mov d0, (a0) and mov d0, (0x10, a0) are stored in LIR, and the address 8n is stored in LAR. Therefore, during executing the instructions stored in LIR, CPU core starts a fetch from the beginning of the 8-byte boundary, and can fetch the maximum 8-byte instruction. This can decrease a possibility that lack of instruction supply occurs.

[Applicable instruction]

<Preceding instruction>	Bcc, JMP, CALL, CALLS , RET, RETF, RETS, RTI, TRAP
<Branch target instruction>	All instructions



High-speed processing

(3) Assignment of the instructions subsequent to Bcc and Lcc

[Contents]

The subsequent instruction, i.e. non-branch side, has a short instruction length and large number of execution cycles.

[Example]

blt	LABEL	
MUL	D0, D1	<- 2-byte instruction and multiple-cycle execution

The above program is an example when the branch target of MUL is assigned at non-branch side. As the result of this assignment, when BLT is non-branch, CPU core can fetch subsequent instructions during executing MUL instructions. Therefore, it can get a higher performance without the occurrence of the pipeline stall due to the lack of instruction supply.

[Applicable instructions]

<Preceding instruction>	Bcc and Lcc instructions
<Subsequent instruction>	All instructions



High-speed processing

(4) Assignment of the instructions subsequent to all instructions other than the branch instruction

[Contents]

If the instructions that consumes multiple-byte instructions once (instructions with long instructions word length) are continuously assigned, instruction consumption is more than instruction supply and the lack of instruction supply occurs. Therefore, if the instructions are assigned so as to consume instructions on the average, a higher performance is obtained.

[Example]

mov	0x12345678, d0	<- 6-byte instruction
mov	(a0), d1	<- 1-byte instruction
inc	d0	<- 1-byte instruction
add	d0, d1	<- 1-byte instruction
mov	0x9abcdef0, d2	<- 6-byte instruction

The above program is an example so as not to assign continuously the instructions with long instruction length. As the result of this assignment, instruction consumption can be average, and pipeline stall becomes difficult to occur. Accordingly, a higher performance is obtained.

[Applicable instruction]

<Preceding instruction>	All instructions other than the following instructions: Bcc, Lcc, JMP, CALL, CALLS, RET, RETF, RETS, RTI, TRAP
<Subsequent instruction>	All instructions

3-2 Instruction assignment subsequent to SETLB

SETLB performs write into LIR and LAR at high speed through different processing than normal pipeline processing with special hardware. Therefore, the instructions changing the preceding LIR and LAR are not executed until the change of LIR and LAR has completed.

[Contents]

It is recommended that more than 1 cycle should be inserted between the instructions changing LIR and LAR preceding to SETLB in order to run the pipeline smoothly.

[Examples]

[Example]			
inc	d2		
movm	(sp),[other]	<- Instruction changing LIR and LAR	
setlb		<- SETLB instruction	



A pipeline stall occurs and the operations of SETLB is delayed until the write of LIR and LAR in MOVm has completed since the operations of SETLB start before the write of LIR and LAR has completed.

[Recommended description examples]			
movm	(sp),[other]	<- Instruction changing LIR and LAR	
inc	d2		
setlb		<- SETLB instruction	

SETLB can be executed without the occurrence of pipeline stall since the write of LIR and LAR has already completed when executing SETLB.

[Applicable instructions]

<Preceding or return target instructions>	MOVm (SP), reg, RET, RETF
<Subsequent instruction>	SETLB

3-3 Assignment of the instructions preceding RETF



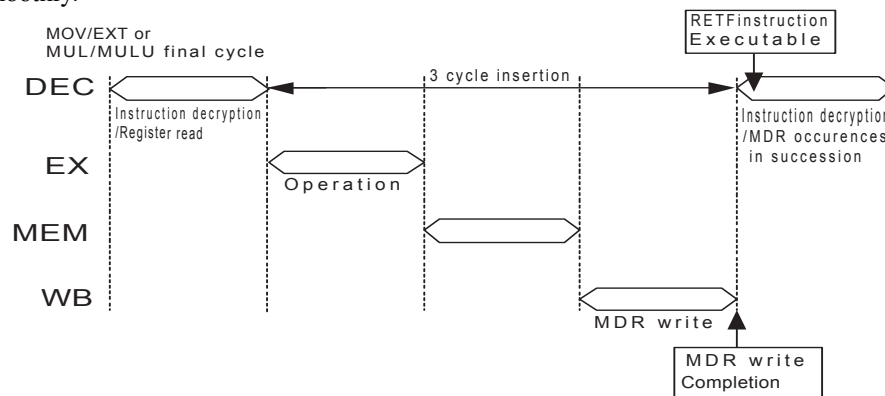
High-speed processing

RETF performs a branch from the subroutine to the return target at high speed by using the contents of MDR as return address. Therefore, the instructions changing the preceding MDR are not executed until the change of MDR has completed.

(1) Assignment of the instructions changing MDR in the final cycle of executing instructions and subsequent RETF.

[Contents]

It is recommended that more than 2 cycles should be inserted between the instructions changing MDR in the final cycle of executing instructions and subsequent RETF in order to run the pipeline smoothly.



[Examples]

[General description examples]

```

_func
_0func      FUNCINFO  _func,8,[ ]
            inc        a0
            mov        d0,mdr    <- Instructions changing MDR
                                   in the final cycle of executing instructions

            inc        a1
            inc        a2
            retf          <- RETF

OR
_func
_0func      FUNCINFO  _func,8,[ ]
            inc        a0
            inc        a1
            mov        a0,mdr    <- Instructions changing MDR
                                   in the final cycle of executing instructions

            inc        a2
            retf          <- RETF

OR
_func
_0func      FUNCINFO  _func,8,[ ]
            inc        a0
            inc        a1
            inc        a2
            mov        d0,mdr    <- Instructions changing MDR
                                   in the final cycle of executing instructions

            retf          <- RETF

```



A pipeline stall occurs and the operations of RETF is delayed until the write of MDR in MOV has completed since the operations of RETF start before the write of MDR has completed.

[General description examples]

_func			
_funco	FUNCINFO	_func,8,[]	
	mov	d0,mdr	<- Instructions changing MDR in the final cycle of executing instructions
	inc	a0	
	inc	a1	
	inc	a2	
	retf		<- RET

RETF can be executed without the occurrence of pipeline stall since the write of MDR in MOV has already completed when executing RETF.

[Applicable instructions]

<Preceding instructions> MOV Dm, MDR, EXT, MUL, MULU

<Subsequent instructions> RETF



For the details of the description of FUNCINFO pseudo-instruction, refer to MN103E series cross-assembler user's manual.

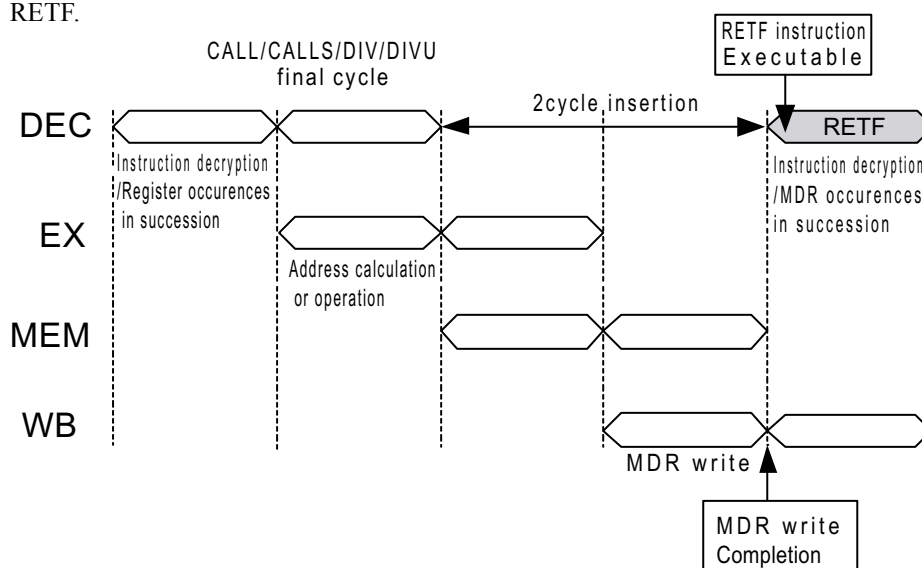
(2) Assignment of the instructions changing MDR in the final cycle of executing instructions and subsequent RETF



High-speed processing

[Contents]

It is recommended that more than 2 cycles should be inserted between the instructions changing MDR at one cycle earlier than the final cycle of the instruction execution and the subsequent RETF.



[Examples]**[General description examples]**

_lab			
LABEL	FUNCINFO	_func,8,[]	
	inc	a0	
	div	d1,d0	<- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
	inc	a1	
	retf		<- RET
OR			
_lab			
LABEL	FUNCINFO	_lab,8,[]	
	inc	a0	
	inc	a1	
	div	d1,d0	<- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
	retf		<- RETF



A pipeline stall occurs and the operations of RETF is delayed until the write of MDR in DIV has completed since the operations of RETF start before the write of MDR has completed.

[Recommended description examples]

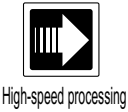
_lab			
LABEL	FUNCINFO	_func,8,[]	
	inc	a0	
	div	d1,d0	<- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
	inc	a0	
	inc	a1	
	retf		<- RET

[Applicable instructions]

<Preceding instructions, or Branch instructions>	DIV, DIVU, CALL, CALLS
<Subsequent instructions>	RETF



For the details of the description of FUNCINFO pseudo-instruction, refer to MN103E series cross-assembler user's manual.



(3) Assignment of the instructions changing MDR at 2 cycles earlier than the final cycle of executing instructions and subsequent RETF

[Contents]

It is recommended that more than 1 cycle should be inserted between the instructions changing MDR at 2 cycles earlier than the final cycle of the instruction execution and the subsequent RETF.

[Examples]

[General description examples]

_lab			
LABEL	FUNCINFO	_func,8,[]	
	inc	a0	
	movm	(sp), [other]	<- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
	retf		<- RET



A pipeline stall occurs and the operations of RETF is delayed until the write of MDR in MOVm has completed since the operations of RETF start before the write of MDR has completed.

[Recommended description examples]

_lab			
LABEL	FUNCINFO	_lab,8,[]	
	movm	(sp), [other]	<- Instructions changing MDR in one cycle earlier than the final cycle of instruction execution
	inc	d0	
	retf		<- RET

RETF can be executed without the occurrence of pipeline stall since the write of MDR in MOVm has already completed when executing RETF.

[Applicable instructions]

<Preceding instructions, or return target instruction>	MOV (SP), reg, RET, RETF
<Subsequent instructions>	RETF



For the details of the description of FUNCINFO pseudo-instruction, refer to MN103E0 series cross-assembler user's manual.

3-4 Assignment of the instructions of CALL/CALLS branch targets



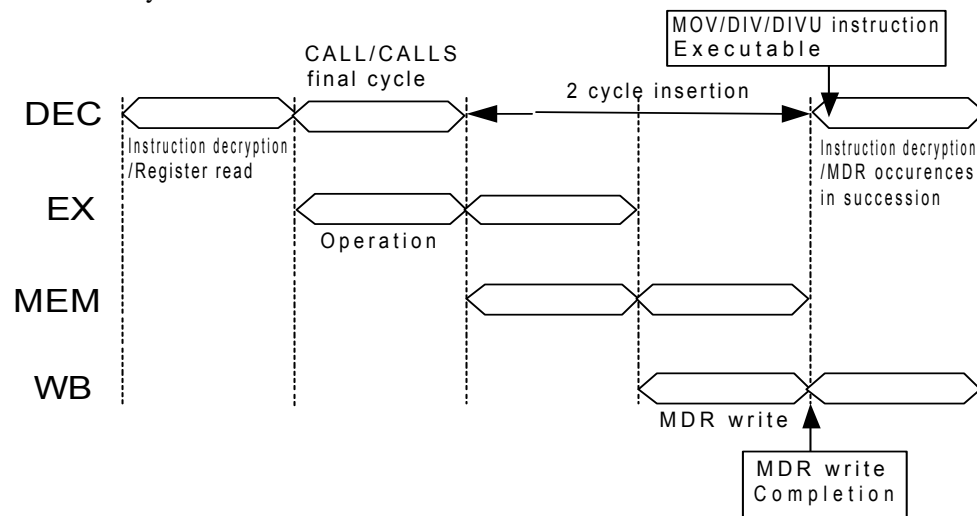
High-speed processing

[Contents]

CALL/CALLS store the return target address in MDR in order to speed up branching to the return address of RETF. Therefore, the instructions referring to MDR of the branch target are not executed until storing the return target address in MDR has completed.

It is recommended that more than 2 cycles should be inserted between CALL/CALLS and the instructions referring to MDR at the branch target in order to run the pipeline smoothly.

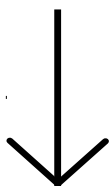
It is recommended that more than 2 cycles should be inserted between the instructions changing MDR in the final cycle of executing instructions and subsequent RETF in order to run the pipeline smoothly.



[Examples]

[General description examples]

	call	LABEL	
	:		
	:		
LABEL	inc	a0	
	mov	mdr, d0	<- Instructions referring to MDR
	inc	a1	
	retf		<- RETF
OR			
	call	LABEL	
	:		
	:		
LABEL	mov	mdr, d0	<- Instructions referring to MDR
	inc	a0	<- Instructions referring to MDR
	inc	a0	
	retf		<- RETF



A pipeline stall occurs and the operations of MOV is delayed until the write to MDR has completed since the operations of MOV start before the write to MDR in CALL has completed.

[Recommended description examples]			
	call	LABEL	<- CALL/CALLS
	:		
	:		
LABEL	inc	a0	
	inc	a1	
	mov	mdr, d0	<- Instructions referring to MDR

When executing MOV, MDR can be referred without the occurrence of pipeline stall since the write to MDR in CALL has completed.

[Applicable instructions]

<Preceding instructions>	CALL, CALLS
<Subsequent instructions>	MOV, MDR, Dn, DIV, DIVU

Chapter 4

Appendix

4

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic		AM33-1		AM33-2/AM33-2A		NOTE
		Throughput	Latency	Throughput	Latency	
MOV	Am,An	1	1	1	1	
MOV	Dm,Dn	1	1	1	1	
MOV	Am,Dn	1	1	1	1	
MOV	Dm,An	1	1	1	1	
MOV	Am,Rn	1	1	1	1	
MOV	Rm,An	1	1	1	1	
MOV	Dm,Rn	1	1	1	1	
MOV	Rm,Dn	1	1	1	1	
MOV	Rm,Rn	1	1	1	1	
MOV	imm8,An	1	1	1	1	
MOV	imm16,An	1	1	1	1	
MOV	imm32,An	2	2	2	2	
MOV	imm8,Dn	1	1	1	1	
MOV	imm16,Dn	1	1	1	1	
MOV	imm32,Dn	2	2	2	2	
MOV	imm8,Rn	1	1	1	1	
MOV	imm24,Rn	2	2	2	2	
MOV	imm32,Rn	2	2	2	2	
MOV	MDR,Dn	1	1	1	1	
MOV	Dm,MDR	1	1	1	1	
MOV	SP,An	1	1	1	1	
MOV	Am,SP	1	1	1	1	
MOV	PC,An	1	1	1	1	
MOV	(Am),Dn	1	3	1	3	
MOV	(Am),An	1	3	1	3	
MOV	(Rm),Rn	1	3	1	3	
MOV	(d8,Am),Dn	1	3	1	3	
MOV	(d16,Am),Dn	1	3	1	3	
MOV	(d32,Am),Dn	2	4	2	4	
MOV	(d8,Am),An	1	3	1	3	
MOV	(d16,Am),An	1	3	1	3	
MOV	(d32,Am),An	2	4	2	4	
MOV	(d8,Rm),Rn	1	3	1	3	
MOV	(d24,Rm),Rn	2	4	2	4	
MOV	(d32,Rm),Rn	2	4	2	4	
MOV	(Di,Am),Dn	1	3	1	3	
MOV	(Di,Am),An	1	3	1	3	
MOV	(Ri,Rm),Rn	1	3	1	3	
MOV	(abs16),Dn	1	3	1	3	
MOV	(abs32),Dn	2	4	2	4	
MOV	(abs16),An	1	3	1	3	
MOV	(abs32),An	2	4	2	4	
MOV	(abs8),Rn	1	3	1	3	
MOV	(abs24),Rn	2	4	2	4	
MOV	(abs32),Rn	2	4	2	4	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
MOV (d8,SP),Dn	1	3	1	3	
MOV (d16,SP),Dn	1	3	1	3	
MOV (d32,SP),Dn	2	4	2	4	
MOV (d8,SP),An	1	3	1	3	
MOV (d16,SP),An	1	3	1	3	
MOV (d32,SP),An	2	4	2	4	
MOV (SP),Rn	1	3	1	3	
MOV (d8,SP),Rn	1	3	1	3	
MOV (d24,SP),Rn	2	4	2	4	
MOV (d32,SP),Rn	2	4	2	4	
MOV (d8,Am),SP	1	3	1	3	
MOV Dm,(An)	1	-	1	-	
MOV Am,(An)	1	-	1	-	
MOV Rm,(Rn)	1	-	1	-	
MOV Dm,(d8,An)	1	-	1	-	
MOV Dm,(d16,An)	1	-	1	-	
MOV Dm,(d32,An)	2	-	2	-	
MOV Am,(d8,An)	1	-	1	-	
MOV Am,(d16,An)	1	-	1	-	
MOV Am,(d32,An)	2	-	2	-	
MOV Rm,(d8,Rn)	1	-	1	-	
MOV Rm,(d24,Rn)	2	-	2	-	
MOV Rm,(d32,Rn)	2	-	2	-	
MOV Dm,(Di,An)	1	-	1	-	
MOV Am,(Di,An)	1	-	1	-	
MOV Rm,(Ri,Rn)	1	-	1	-	
MOV Dm,(abs16)	1	-	1	-	
MOV Dm,(abs32)	2	-	2	-	
MOV Am,(abs16)	1	-	1	-	
MOV Am,(abs32)	2	-	2	-	
MOV Rm,(abs8)	1	-	1	-	
MOV Rm,(abs24)	2	-	2	-	
MOV Rm,(abs32)	2	-	2	-	
MOV Dm,(d8,SP)	1	-	1	-	
MOV Dm,(d16,SP)	1	-	1	-	
MOV Dm,(d32,SP)	2	-	2	-	
MOV Am,(d8,SP)	1	-	1	-	
MOV Am,(d16,SP)	1	-	1	-	
MOV Am,(d32,SP)	2	-	2	-	
MOV Rm,(SP)	1	-	1	-	
MOV Rm,(d8,SP)	1	-	1	-	
MOV Rm,(d24,SP)	2	-	2	-	
MOV Rm,(d32,SP)	2	-	2	-	
MOV SP,(d8,An)	1	-	1	-	
MOV (Rm+,imm8),Rn	1	3	1	3	

Rm is valid at Latency of 1.

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic		AM33-1		AM33-2/AM33-2A		NOTE
		Throughput	Latency	Throughput	Latency	
MOV	(Rm+,imm24),Rn	2	4	2	4	Rm is valid at Latency of 2. Rm is valid at Latency of 2. Rm is valid at Latency of 1.
MOV	(Rm+,imm32),Rn	2	4	2	4	
MOV	(Rm+),Rn	1	3	1	3	
MOV	Rm,(Rn+,imm8)	1	1	1	1	
MOV	Rm,(Rn+,imm24)	2	2	2	2	
MOV	Rm,(Rn+,imm32)	2	2	2	2	
MOV	Rm,(Rn+)	1	1	1	1	
MOV	MSP,An	1	1	1	1	
MOV	Am,MSP	1	1	1	1	
MOV	SSP,An	1	1	1	1	
MOV	Am,SSP	1	1	1	1	
MOV	USP,An	1	1	1	1	
MOV	Dm,EPsw	1	1	1	1	
MOV	EPsw,Dn	1	1	1	1	
MOV	Dm,PSW	1	1	1	1	
MOV	PSW,Dn	1	1	1	1	
MOVU	imm8,Rn	1	1	1	1	
MOVU	imm24,Rn	2	2	2	2	
MOVU	imm32,Rn	2	2	2	2	
MOVHU	(Am),Dn	1	3	1	3	
MOVHU	(Rm),Rn	1	3	1	3	
MOVHU	(d8,Am),Dn	1	3	1	3	
MOVHU	(d16,Am),Dn	1	3	1	3	
MOVHU	(d32,Am),Dn	2	4	2	4	
MOVHU	(d8,Rm),Rn	1	3	1	3	
MOVHU	(d24,Rm),Rn	2	4	2	4	
MOVHU	(d32,Rm),Rn	2	4	2	4	
MOVHU	(Di,Am),Dn	1	3	1	3	
MOVHU	(Ri,Rm),Rn	1	3	1	3	
MOVHU	(abs16),Dn	1	3	1	3	
MOVHU	(abs32),Dn	2	4	2	4	
MOVHU	(abs8),Rn	1	3	1	3	
MOVHU	(abs24),Rn	2	4	2	4	
MOVHU	(abs32),Rn	2	4	2	4	
MOVHU	(d8,SP),Dn	1	3	1	3	
MOVHU	(d16,SP),Dn	1	3	1	3	
MOVHU	(d32,SP),Dn	2	4	2	4	
MOVHU	(SP),Rn	1	3	1	3	
MOVHU	(d8,SP),Rn	1	3	1	3	
MOVHU	(d24,SP),Rn	2	4	2	4	
MOVHU	(d32,SP),Rn	2	4	2	4	
MOVHU	Dm,(An)	1	-	1	-	
MOVHU	Rm,(Rn)	1	-	1	-	
MOVHU	Dm,(d8,An)	1	-	1	-	
MOVHU	Dm,(d16,An)	1	-	1	-	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
MOVHU Dm,(d32,An)	2	-	2	-	
MOVHU Rm,(d8,Rn)	1	-	1	-	
MOVHU Rm,(d24,Rn)	2	-	2	-	
MOVHU Rm,(d32,Rn)	2	-	2	-	
MOVHU Dm,(Di,An)	1	-	1	-	
MOVHU Rm,(Ri,Rn)	1	-	1	-	
MOVHU Dm,(abs16)	1	-	1	-	
MOVHU Dm,(abs32)	2	-	2	-	
MOVHU Rm,(abs8)	1	-	1	-	
MOVHU Rm,(abs24)	2	-	2	-	
MOVHU Rm,(abs32)	2	-	2	-	
MOVHU Dm,(d8,SP)	1	-	1	-	
MOVHU Dm,(d16,SP)	1	-	1	-	
MOVHU Dm,(d32,SP)	2	-	2	-	
MOVHU Rm,(SP)	1	-	1	-	
MOVHU Rm,(d8,SP)	1	-	1	-	
MOVHU Rm,(d24,SP)	2	-	2	-	
MOVHU Rm,(d32,SP)	2	-	2	-	
MOVHU (Rm+,imm8),Rn	1	3	1	3	Rm is valid at Latency of 1.
MOVHU (Rm+,imm24),Rn	2	4	2	4	Rm is valid at Latency of 2.
MOVHU (Rm+,imm32),Rn	2	4	2	4	Rm is valid at Latency of 2.
MOVHU (Rm+),Rn	1	3	1	3	Rm is valid at Latency of 1.
MOVHU Rm,(Rn+,imm8)	1	1	1	1	
MOVHU Rm,(Rn+,imm24)	2	2	2	2	
MOVHU Rm,(Rn+,imm32)	2	2	2	2	
MOVHU Rm,(Rn+)	1	1	1	1	
MOVBU (Am),Dn	1	3	1	3	
MOVBU (Rm),Rn	1	3	1	3	
MOVBU (d8,Am),Dn	1	3	1	3	
MOVBU (d16,Am),Dn	1	3	1	3	
MOVBU (d32,Am),Dn	2	4	2	4	
MOVBU (d8,Rm),Rn	1	3	1	3	
MOVBU (d24,Rm),Rn	2	4	2	4	
MOVBU (d32,Rm),Rn	2	4	2	4	
MOVBU (Di,Am),Dn	1	3	1	3	
MOVBU (Ri,Rm),Rn	1	3	1	3	
MOVBU (abs16),Dn	1	3	1	3	
MOVBU (abs32),Dn	2	4	2	4	
MOVBU (abs8),Rn	1	3	1	3	
MOVBU (abs24),Rn	2	4	2	4	
MOVBU (abs32),Rn	2	4	2	4	
MOVBU (d8,SP),Dn	1	3	1	3	
MOVBU (d16,SP),Dn	1	3	1	3	
MOVBU (d32,SP),Dn	2	4	2	4	
MOVBU (SP),Rn	1	3	1	3	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
MOVB (d8,SP),Rn	1	3	1	3	
MOVB (d24,SP),Rn	2	4	2	4	
MOVB (d32,SP),Rn	2	4	2	4	
MOVB Dm,(An)	1	-	1	-	
MOVB Rm,(Rn)	1	-	1	-	
MOVB Dm,(d8,An)	1	-	1	-	
MOVB Dm,(d16,An)	1	-	1	-	
MOVB Dm,(d32,An)	2	-	2	-	
MOVB Rm,(d8,Rn)	1	-	1	-	
MOVB Rm,(d24,Rn)	2	-	2	-	
MOVB Rm,(d32,Rn)	2	-	2	-	
MOVB Dm,(Di,An)	1	-	1	-	
MOVB Rm,(Ri,Rn)	1	-	1	-	
MOVB Dm,(abs16)	1	-	1	-	
MOVB Dm,(abs32)	2	-	2	-	
MOVB Rm,(abs8)	1	-	1	-	
MOVB Rm,(abs24)	2	-	2	-	
MOVB Rm,(abs32)	2	-	2	-	
MOVB Dm,(d8,SP)	1	-	1	-	
MOVB Dm,(d16,SP)	1	-	1	-	
MOVB Dm,(d32,SP)	2	-	2	-	
MOVB Rm,(SP)	1	-	1	-	
MOVB Rm,(d8,SP)	1	-	1	-	
MOVB Rm,(d24,SP)	2	-	2	-	
MOVB Rm,(d32,SP)	2	-	2	-	
MOVM (SP),regs	2+REGs	2+REGs	2+REGs	2+REGs	REGs=Numberofregister
MOVM (USP),regs	2+REGs	2+REGs	2+REGs	2+REGs	
MOVM regs,(SP)	2+REGs	2+REGs	2+REGs	2+REGs	
MOVM regs,(USP)	2+REGs	2+REGs	2+REGs	2+REGs	
EXT Dn	1	1	1	1	
EXT Rn	1	1	1	1	
EXTH Dn	1	1	1	1	
EXTH Rn	1	1	1	1	
EXTH Rm,Rn	1	1	1	1	
EXTHU Dn	1	1	1	1	
EXTHU Rn	1	1	1	1	
EXTHU Rm,Rn	1	1	1	1	
EXTB Dn	1	1	1	1	
EXTB Rn	1	1	1	1	
EXTB Rm,Rn	1	1	1	1	
EXTBU Dn	1	1	1	1	
EXTBU Rn	1	1	1	1	
EXTBU Rm,Rn	1	1	1	1	
CLR Dn	1	1	1	1	
CLR Rn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
DCPF (Rm)	-	-	1	-	
DCPF (d8,Rm)	-	-	1	-	
DCPF (d24,Rm)	-	-	2	-	
DCPF (d32,Rm)	-	-	2	-	
DCPF (Ri,Rm)	-	-	1	-	
DCPF (SP)	-	-	1	-	
ADD Dm,Dn	1	1	1	1	
ADD Dm,An	1	1	1	1	
ADD Am,Dn	1	1	1	1	
ADD Am,An	1	1	1	1	
ADD Rm,Rn,Rd	1	1	1	1	
ADD Rm,Rn	1	1	1	1	
ADD imm8,Dn	1	1	1	1	
ADD imm16,Dn	1	1	1	1	
ADD imm32,Dn	2	2	2	2	
ADD imm8,An	1	1	1	1	
ADD imm16,An	1	1	1	1	
ADD imm32,An	2	2	2	2	
ADD imm8,Rn	1	1	1	1	
ADD imm24,Rn	2	2	2	2	
ADD imm32,Rn	2	2	2	2	
ADD imm8,SP	1	1	1	1	
ADD imm16,SP	1	1	1	1	
ADD imm32,SP	2	2	2	2	
ADDC Dm,Dn	1	1	1	1	
ADDC Rm,Rn	1	1	1	1	
ADDC Rm,Rn,Rd	1	1	1	1	
ADDC imm8,Rn	1	1	1	1	
ADDC imm24,Rn	2	2	2	2	
ADDC imm32,Rn	2	2	2	2	
SUB Dm,Dn	1	1	1	1	
SUB Dm,An	1	1	1	1	
SUB Am,Dn	1	1	1	1	
SUB Am,An	1	1	1	1	
SUB Rm,Rn	1	1	1	1	
SUB Rm,Rn,Rd	1	1	1	1	
SUB imm8,Rn	1	1	1	1	
SUB imm24,Rn	2	2	2	2	
SUB imm32,Rn	2	2	2	2	
SUB imm32,Dn	2	2	2	2	
SUB imm32,An	2	2	2	2	
SUBC Dm,Dn	1	1	1	1	
SUBC Rm,Rn	1	1	1	1	
SUBC Rm,Rn,Rd	1	1	1	1	
SUBC imm8,Rn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic		AM33-1		AM33-2/AM33-2A		NOTE
		Throughput	Latency	Throughput	Latency	
SUBC	imm24,Rn	2	2	2	2	
SUBC	imm32,Rn	2	2	2	2	
MUL	Dm,Dn	2	3	2	3	
MUL	Rm,Rn	2	3	2	3	
MUL	Rm,Rn,Rd1,Rd2	2	3	2	3	
MUL	imm8,Rn	2	3	2	3	
MUL	imm24,Rn	3	4	3	4	
MUL	imm32,Rn	3	4	3	4	
MULU	Dm,Dn	2	3	2	3	
MULU	Rm,Rn	2	3	2	3	
MULU	Rm,Rn,Rd1,Rd2	2	3	2	3	
MULU	imm8,Rn	2	3	2	3	
MULU	imm24,Rn	3	4	3	4	
MULU	imm32,Rn	3	4	3	4	
DIV	Dm,Dn					
{MDR,Dn}=0		4	5	4	5	
Value{MDR,Dn} can specify by 1-byte.		14	14	14	14	
Value{MDR,Dn} can specify by 2-byte.		22	23	22	23	
Value{MDR,Dn} can specify by 3-byte.		30	31	30	31	
Value{MDR,Dn} can specify by 4-byte or more.		38	39	38	39	
DIV	Rm,Rn					
{MDR,Dn}=0		4	5	4	5	
Value{MDR,Dn} can specify by 1-byte.		14	14	14	14	
Value{MDR,Dn} can specify by 2-byte.		22	23	22	23	
Value{MDR,Dn} can specify by 3-byte.		30	31	30	31	
Value{MDR,Dn} can specify by 4-byte or more.		38	39	38	39	
DIVU	Dm,Dn					
{MDR,Dn}=0		4	5	4	5	
Value{MDR,Dn} can specify by 1-byte.		14	14	14	14	
Value{MDR,Dn} can specify by 2-byte.		22	23	22	23	
Value{MDR,Dn} can specify by 3-byte.		30	31	30	31	
Value{MDR,Dn} can specify by 4-byte or more.		38	39	38	39	
DIVU	Rm,Rn					
{MDR,Dn}=0		4	5	4	5	
Value{MDR,Dn} can specify by 1-byte.		14	14	14	14	
Value{MDR,Dn} can specify by 2-byte.		22	23	22	23	
Value{MDR,Dn} can specify by 3-byte.		30	31	30	31	
Value {MDR,Dn} can specify by 4-byte or more.		38	39	38	39	
INC	An	1	1	1	1	
INC	Dn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic		AM33-1		AM33-2/AM33-2A		NOTE
		Throughput	Latency	Throughput	Latency	
INC	Rn	1	1	1	1	
INC4	An	1	1	1	1	
INC4	Rn	1	1	1	1	
CMP	Dm,Dn	1	1	1	1	
CMP	Dm,An	1	1	1	1	
CMP	Am,Dn	1	1	1	1	
CMP	Am,An	1	1	1	1	
CMP	Rm,Rn	1	1	1	1	
CMP	imm8,Dn	1	1	1	1	
CMP	imm16,Dn	1	1	1	1	
CMP	imm32,Dn	2	2	2	2	
CMP	imm8,An	1	1	1	1	
CMP	imm16,An	1	1	1	1	
CMP	imm32,An	2	2	2	2	
CMP	imm8,Rn	1	1	1	1	
CMP	imm24,Rn	2	2	2	2	
CMP	imm32,Rn	2	2	2	2	
AND	Dm,Dn	1	1	1	1	
AND	Rm,Rn	1	1	1	1	
AND	Rm,Rn,Rd	1	1	1	1	
AND	imm8,Dn	1	1	1	1	
AND	imm16,Dn	1	1	1	1	
AND	imm32,Dn	2	2	2	2	
AND	imm8,Rn	1	1	1	1	
AND	imm24,Rn	2	2	2	2	
AND	imm32,Rn	2	2	2	2	
AND	imm16,PSW	1	1	1	1	
AND	imm32,EPSW	2	2	2	2	
OR	Dm,Dn	1	1	1	1	
OR	Rm,Rn	1	1	1	1	
OR	Rm,Rn,Rd	1	1	1	1	
OR	imm8,Dn	1	1	1	1	
OR	imm16,Dn	1	1	1	1	
OR	imm32,Dn	2	2	2	2	
OR	imm8,Rn	1	1	1	1	
OR	imm24,Rn	2	2	2	2	
OR	imm32,Rn	2	2	2	2	
OR	imm16,PSW	1	1	1	1	
OR	imm32,EPSW	2	2	2	2	
XOR	Dm,Dn	1	1	1	1	
XOR	Rm,Rn	1	1	1	1	
XOR	Rm,Rn,Rd	1	1	1	1	
XOR	imm16,Dn	1	1	1	1	
XOR	imm32,Dn	2	2	2	2	
XOR	imm8,Rn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic		AM33-1		AM33-2/AM33-2A		NOTE
		Throughput	Latency	Throughput	Latency	
XOR	imm24,Rn	2	2	2	2	
XOR	imm32,Rn	2	2	2	2	
NOT	Dn	1	1	1	1	
NOT	Rn	1	1	1	1	
BTST	imm8,Dn	1	1	1	1	
BTST	imm16,Dn	1	1	1	1	
BTST	imm32,Dn	2	2	2	2	
BTST	imm8,Rn	1	1	1	1	
BTST	imm24,Rn	2	2	2	2	
BTST	imm32,Rn	2	2	2	2	
BTST	imm8,(d8,An)	4	4	4	4	
BTST	imm8,(abs16)	-	-	4	4	
BTST	imm8,(abs32)	5	5	5	5	
BSET	Dm,(An)	6	6	6	6	
BSET	imm8,(d8,An)	6	6	6	6	
BSET	imm8,(abs16)	-	-	6	6	
BSET	imm8,(abs32)	7	7	7	7	
BCLR	Dm,(An)	6	6	6	6	
BCLR	imm8,(d8,An)	6	6	6	6	
BCLR	imm8,(abs16)	-	-	6	6	
BCLR	imm8,(abs32)	7	7	7	7	
ASR	Dm,Dn	1	1	1	1	
ASR	Rm,Rn	1	1	1	1	
ASR	Rm,Rn,Rd	1	1	1	1	
ASR	imm8,Dn	1	1	1	1	
ASR	imm8,Rn	1	1	1	1	
ASR	imm24,Rn	2	2	2	2	
ASR	imm32,Rn	2	2	2	2	
LSR	Dm,Dn	1	1	1	1	
LSR	Rm,Rn	1	1	1	1	
LSR	Rm,Rn,Rd	1	1	1	1	
LSR	imm8,Dn	1	1	1	1	
LSR	imm8,Rn	1	1	1	1	
LSR	imm24,Rn	2	2	2	2	
LSR	imm32,Rn	2	2	2	2	
ASL	Dm,Dn	1	1	1	1	
ASL	Rm,Rn	1	1	1	1	
ASL	Rm,Rn,Rd	1	1	1	1	
ASL	imm8,Dn	1	1	1	1	
ASL	imm8,Rn	1	1	1	1	
ASL	imm24,Rn	2	2	2	2	
ASL	imm32,Rn	2	2	2	2	
ASL2	Dn	1	1	1	1	
ASL2	Rn	1	1	1	1	
ROR	Dn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic		AM33-1		AM33-2/AM33-2A		NOTE
		Throughput	Latency	Throughput	Latency	
ROR	Rn	1	1	1	1	Throughput is 1 if branch is not taken.
ROL	Dn	1	1	1	1	
ROL	Rn	1	1	1	1	
BEQ	(d8,PC)	3	-	3	-	
BNE	(d8,PC)	3	-	3	-	
BGT	(d8,PC)	3	-	3	-	
BGE	(d8,PC)	3	-	3	-	
BLE	(d8,PC)	3	-	3	-	
BLT	(d8,PC)	3	-	3	-	
BHI	(d8,PC)	3	-	3	-	
BCC	(d8,PC)	3	-	3	-	
BLS	(d8,PC)	3	-	3	-	
BCS	(d8,PC)	3	-	3	-	
BVC	(d8,PC)	4	-	4	-	Throughput is 2 if branch is not taken.
BVS	(d8,PC)	4	-	4	-	
BNC	(d8,PC)	4	-	4	-	
BNS	(d8,PC)	4	-	4	-	
BRA	(d8,PC)	3	-	3	-	Throughput is 2 if branch is not taken.
LEQ		1	-	1	-	
LNE		1	-	1	-	
LGT		1	-	1	-	
LGE		1	-	1	-	
LLE		1	-	1	-	
LLT		1	-	1	-	
LHI		1	-	1	-	
LCC		1	-	1	-	
LLS		1	-	1	-	
LCS		1	-	1	-	
LRA		1	-	1	-	
SETLB		1	-	1	-	
JMP	(An)	3	-	3	-	REGs=Numberofregister
JMP	(d16,PC)	3	-	3	-	
JMP	(d32,PC)	4	-	4	-	
CALL	(d16,PC),regs,imm8	2+REGs	-	2+REGs	-	
CALL	(d32,PC),regs,imm8	3+REGs	-	3+REGs	-	REGs=Numberofregister
CALLS	(An)	3	-	3	-	
CALLS	(d16,PC)	4	-	4	-	
CALLS	(d32,PC)	4	-	4	-	
RET	regs,imm8	5+REGs	-	5+REGs	-	
RETF	regs,imm8	2+REGs	-	2+REGs	-	REGs=Numberofregister
RETS		6	-	6	-	
RTI		7	-	7	-	
TRAP		3	-	3	-	
NOP		1	-	1	-	
SYSCALL imm4		*	*	*	*	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
PI	*	*	*	*	
DMULH Rm,Rn	1	2	1	2	
DMULH Rm,Rn,Rd1,Rd2	1	2	1	2	
DMULH imm32,Rn	2	3	2	3	
DMULHU Rm,Rn	1	2	1	2	
DMULHU Rm,Rn,Rd1,Rd2	1	2	1	2	
DMULHU imm32,Rn	2	3	2	3	
DMACH Rm,Rn	1	2	1	2	
DMACH Rm,Rn,Rd	1	2	1	2	
DMACH imm32,Rn	2	3	2	3	
DMACHU Rm,Rn	1	2	1	2	
DMACHU Rm,Rn,Rd	1	2	1	2	
DMACHU imm32,Rn	2	3	2	3	
MAC Rm,Rn	2	3	2	3	
MAC Rm,Rn,Rd1,Rd2	2	3	2	3	
MAC imm8,Rn	2	3	2	3	
MAC imm24,Rn	3	4	3	4	
MAC imm32,Rn	3	4	3	4	
MACU Rm,Rn	2	3	2	3	
MACU Rm,Rn,Rd1,Rd2	2	3	2	3	
MACU imm8,Rn	2	3	2	3	
MACU imm24,Rn	3	4	3	4	
MACU imm32,Rn	3	4	3	4	
MACH Rm,Rn	2	3	2	3	
MACH Rm,Rn,Rd1,Rd2	2	3	2	3	
MACH imm8,Rn	2	3	2	3	
MACH imm24,Rn	3	4	3	4	
MACH imm32,Rn	3	4	3	4	
MACHU Rm,Rn	2	3	2	3	
MACHU Rm,Rn,Rd1,Rd2	2	3	2	3	
MACHU imm8,Rn	2	3	2	3	
MACHU imm24,Rn	3	4	3	4	
MACHU imm32,Rn	3	4	3	4	
MACB Rm,Rn	2	3	2	3	
MACB Rm,Rn,Rd	2	3	2	3	
MACB imm8,Rn	2	3	2	3	
MACB imm24,Rn	3	4	3	4	
MACB imm32,Rn	3	4	3	4	
MACBU Rm,Rn	2	3	2	3	
MACBU Rm,Rn,Rd	2	3	2	3	
MACBU imm8,Rn	2	3	2	3	
MACBU imm24,Rn	3	4	3	4	
MACBU imm32,Rn	3	4	3	4	
SWHW Rm,Rn	1	1	1	1	
SWAP Rm,Rn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
SWAPH Rm,Rn	1	1	1	1	
SAT16 Rm,Rn	1	1	1	1	
SAT24 Rm,Rn	1	1	1	1	
MCSTE Rm,Rn	1	1	1	1	
MCSTE imm8,Rn	1	1	1	1	
BSCH Rm,Rn	1	1	1	1	
BSCH Rm,Rn,Rd	1	1	1	1	
ADD_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
ADD_ASL Rm1,Rn1,imm4,Rn2	1	1	1	1	
ADD_ADD imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_SUB imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_CMP imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_MOV imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_ASR imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_LSR imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_ASL imm4,Rn1,Rm2,Rn2	1	1	1	1	
ADD_ADD imm4,Rn1,imm4,Rn2	1	1	1	1	
ADD_SUB imm4,Rn1,imm4,Rn2	1	1	1	1	
ADD_CMP imm4,Rn1,imm4,Rn2	1	1	1	1	
ADD_MOV imm4,Rn1,imm4,Rn2	1	1	1	1	
ADD_ASR imm4,Rn1,imm4,Rn2	1	1	1	1	
ADD_LSR imm4,Rn1,imm4,Rn2	1	1	1	1	
ADD_ASL imm4,Rn1,imm4,Rn2	1	1	1	1	
CMP_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
CMP_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
CMP_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
CMP_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
CMP_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
CMP_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
CMP_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
CMP_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
CMP_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
CMP_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
CMP_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
CMP_ASLRm1,Rn1,imm4,Rn2	1	1	1	1	
CMP_ADD imm4,Rn1,Rm2,Rn2	1	1	1	1	
CMP_SUB imm4,Rn1,Rm2,Rn2	1	1	1	1	
CMP_MOV imm4,Rn1,Rm2,Rn2	1	1	1	1	
CMP_ASR imm4,Rn1,Rm2,Rn2	1	1	1	1	
CMP_LSR imm4,Rn1,Rm2,Rn2	1	1	1	1	
CMP_ASLim4,Rn1,Rm2,Rn2	1	1	1	1	
CMP_ADD imm4,Rn1,imm4,Rn2	1	1	1	1	
CMP_SUB imm4,Rn1,imm4,Rn2	1	1	1	1	
CMP_MOV imm4,Rn1,imm4,Rn2	1	1	1	1	
CMP_ASR imm4,Rn1,imm4,Rn2	1	1	1	1	
CMP_LSR imm4,Rn1,imm4,Rn2	1	1	1	1	
CMP_ASLim4,Rn1,imm4,Rn2	1	1	1	1	
SUB_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_SUBRm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_ASRRm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SUB_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_SUBRm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_ASRRm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_ASL Rm1,Rn1,imm4,Rn2	1	1	1	1	
SUB_ADD imm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_SUBimm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_CMP imm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_MOV imm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_ASRimm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_LSR imm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_ASL imm4,Rn1,Rm2,Rn2	1	1	1	1	
SUB_ADD imm4,Rn1,imm4,Rn2	1	1	1	1	
SUB_SUBimm4,Rn1,imm4,Rn2	1	1	1	1	
SUB_CMP imm4,Rn1,imm4,Rn2	1	1	1	1	
SUB_MOV imm4,Rn1,imm4,Rn2	1	1	1	1	
SUB_ASRimm4,Rn1,imm4,Rn2	1	1	1	1	
SUB_LSR imm4,Rn1,imm4,Rn2	1	1	1	1	
SUB_ASL imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
MOV_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
MOV_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
MOV_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
MOV_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
MOV_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
MOV_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
MOV_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_ASL Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_ADD imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_SUB imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_CMP imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_MOV imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_ASR imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_LSR imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_ASL imm4,Rn1,Rm2,Rn2	1	1	1	1	
MOV_ADD imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_SUB imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_CMP imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_MOV imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_ASR imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_LSR imm4,Rn1,imm4,Rn2	1	1	1	1	
MOV_ASL imm4,Rn1,imm4,Rn2	1	1	1	1	
AND_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
AND_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
AND_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
AND_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
AND_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
AND_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
AND_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
AND_ASL Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
OR_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
OR_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
OR_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
OR_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
OR_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
OR_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	Rm is valid at Latency of 2.
OR_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
OR_ASL Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
XOR_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
XOR_ASL Rm1,Rn1,imm4,Rn2	1	1	1	1	
DMACH_ADDRm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_SUBRm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_CMPRm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_MOVRm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_ASR Rm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_LSR Rm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_ASL Rm1,Rn1,Rm2,Rn2	1	2	1	2	
DMACH_ADDRm1,Rn1,imm4,Rn2	1	2	1	2	
DMACH_SUBRm1,Rn1,imm4,Rn2	1	2	1	2	
DMACH_CMPRm1,Rn1,imm4,Rn2	1	2	1	2	
DMACH_MOVRm1,Rn1,imm4,Rn2	1	2	1	2	
DMACH_ASR Rm1,Rn1,imm4,Rn2	1	2	1	2	
DMACH_LSR Rm1,Rn1,imm4,Rn2	1	2	1	2	
DMACH_ASL Rm1,Rn1,imm4,Rn2	1	2	1	2	
SWHW_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_ASL Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SWHW_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
SWHW_SUBRm1,Rn1,imm4,Rn2	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
SWHW_CMPRm1,Rn1,imm4,Rn2	1	1	1	1	<div> Rm is valid at Latency of 1. Throughput is 2 if branch is not taken. </div>
SWHW_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
SWHW_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
SWHW_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
SWHW_ASX Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_ADD Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_SUB Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_CMP Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_MOV Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_ASR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_LSR Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_ASX Rm1,Rn1,Rm2,Rn2	1	1	1	1	
SAT16_ADD Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_SUB Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_CMP Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_MOV Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_ASR Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_LSR Rm1,Rn1,imm4,Rn2	1	1	1	1	
SAT16_ASX Rm1,Rn1,imm4,Rn2	1	1	1	1	
MOV_LEQ (Rm+,imm4),Rn	1	3	1	3	
MOV_LNE (Rm+,imm4),Rn	1	3	1	3	
MOV_LGT (Rm+,imm4),Rn	1	3	1	3	
MOV_LGE (Rm+,imm4),Rn	1	3	1	3	
MOV_LLE(Rm+,imm4),Rn	1	3	1	3	
MOV_LLT(Rm+,imm4),Rn	1	3	1	3	
MOV_LHI (Rm+,imm4),Rn	1	3	1	3	
MOV_LCC (Rm+,imm4),Rn	1	3	1	3	
MOV_LLS(Rm+,imm4),Rn	1	3	1	3	
MOV_LCS (Rm+,imm4),Rn	1	3	1	3	
MOV_LRA (Rm+,imm4),Rn	1	3	1	3	
UDF00 Dm,Dn	2	3	2	3	
UDF00 imm8,Dn	2	3	2	3	
UDF00 imm16,Dn	2	3	2	3	
UDF00 imm32,Dn	3	4	3	4	
UDF01 Dm,Dn	2	3	2	3	
UDFU01 imm8,Dn	2	3	2	3	
UDFU01 imm16,Dn	2	3	2	3	
UDFU01 imm32,Dn	3	4	3	4	
UDF02 Dm,Dn	1	1	1	1	
UDF03 Dm,Dn	1	1	1	1	
UDF04 Dm,Dn	1	1	1	1	
UDF05 Dm,Dn	1	1	1	1	
UDF06 Dm,Dn	1	1	1	1	
UDF07 Dm,Dn	1	1	1	1	
UDF08 Dm,Dn	1	1	1	1	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
UDF09 Dm,Dn	1	4	1	1	
UDF12 Dm,Dn	1	4	1	1	
UDF13 Dm,Dn	1	1	1	1	
UDF15 Dm,Dn	1	1	1	1	
FMOV (Rm),FSn	-	-	1	4	
FMOV (Ri,Rm),FSn	-	-	1	4	
FMOV (d8,Rm),FSn	-	-	1	4	
FMOV (d24,Rm),FSn	-	-	2	5	
FMOV (d32,Rm),FSn	-	-	2	5	
FMOV (d8,SP),FSn	-	-	1	4	
FMOV (d24,SP),FSn	-	-	2	5	
FMOV (d32,SP),FSn	-	-	2	5	
FMOV FSm,(Rn)	-	-	1	-	
FMOV FSm,(Ri,Rn)	-	-	1	-	
FMOV FSm,(d8,Rn)	-	-	1	-	
FMOV FSm,(d24,Rn)	-	-	2	-	
FMOV FSm,(d32,Rn)	-	-	2	-	
FMOV FSm,(d8,SP)	-	-	1	-	
FMOV FSm,(d24,SP)	-	-	2	-	
FMOV FSm,(d32,SP)	-	-	2	-	
FMOV (Rm+),FSn	-	-	1	4	
FMOV (Rm+,imm8),FSn	-	-	1	4	
FMOV (Rm+,imm24),FSn	-	-	2	5	
FMOV (Rm+,imm32),FSn	-	-	2	5	
FMOV FSm,(Rn+)	-	-	1	-	
FMOV FSm,(Rn+,imm8)	-	-	1	-	
FMOV FSm,(Rn+,imm24)	-	-	2	-	
FMOV FSm,(Rn+,imm32)	-	-	2	5	
FMOV FSm,FSn	-	-	1	4	
FMOV FSm,Rn	-	-	1	2	
FMOV Rm,FSn	-	-	1	4	
FMOV imm32,FSn	-	-	2	5	
FMOV FPCR,Rn	-	-	1	2	
FMOV Rm,FPCR	-	-	1	4	
FMOV imm32,FPCR	-	-	2	5	
FMOV (SP),FSn	-	-	1	4	
FMOV FSm,(SP)	-	-	1	-	
FABS FSn	-	-	1	4	
FABS FSm,FSn	-	-	1	4	
FNEG FSn	-	-	1	4	
FNEG FSm,FSn	-	-	1	4	
FRSQRT FSn	-	-	23	27	
When input operand is 0,¥,NaN or negative.	-	-	17	21	
FRSQRT FSm,FSn	-	-	23	27	
When input operand is 0,¥,NaN or negative.	-	-	17	21	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
FCMP FSm1,FSm2	-	-	1	4	
FCMP imm32,FSm	-	-	2	5	
FADD FSm,FSn	-	-	1	4	
FADD FSm1,FSm2,FSn	-	-	1	4	
FADD imm32,FSm,FSn	-	-	2	5	
FSUB FSm,FSn	-	-	1	4	
FSUB FSm1,FSm2,FSn	-	-	1	4	
FSUB imm32,FSm,FSn	-	-	2	5	
FMUL FSm,FSn	-	-	1	4	
FMUL FSm1,FSm2,FSn	-	-	1	4	
FMUL imm32,FSm,FSn	-	-	2	5	
FDIV FSm,FSn	-	-	12	16	
When input operand is 0,¥,NaN or negative.	-	-	9	13	
FDIV FSm1,FSm2,FSn	-	-	12	16	
When input operand is 0,¥,NaN or negative.	-	-	9	13	
FDIV imm32,FSm,FSn	-	-	13	17	
When input operand is 0,¥,NaN or negative.	-	-	10	14	
FMADD FSm1,FSm2,FSm3,FSn	-	-	1	4	
FMSUB FSm1,FSm2,FSm3,FSn	-	-	1	4	
FNMADD FSm1,FSm2,FSm3,FSn	-	-	1	4	
FNMSUB FSm1,FSm2,FSm3,FSn	-	-	1	4	
FMOV (Rm),FDn	-	-	1	4	
FMOV (Ri,Rm),FDn	-	-	1	4	
FMOV (d8,Rm),FDn	-	-	1	4	
FMOV (d24,Rm),FDn	-	-	2	5	
FMOV (d32,Rm),FDn	-	-	2	5	
FMOV (SP),FDn	-	-	1	4	
FMOV (d8,SP),FDn	-	-	1	4	
FMOV (d24,SP),FDn	-	-	2	5	
FMOV (d32,SP),FDn	-	-	2	5	
FMOV FDm,(Rn)	-	-	1	-	
FMOV FDm,(Ri,Rn)	-	-	1	-	
FMOV FDm,(d8,Rn)	-	-	1	-	
FMOV FDm,(d24,Rn)	-	-	2	-	
FMOV FDm,(d32,Rn)	-	-	2	-	
FMOV FDm,(SP)	-	-	1	-	
FMOV FDm,(d8,SP)	-	-	1	-	
FMOV FDm,(d24,SP)	-	-	2	-	
FMOV FDm,(d32,SP)	-	-	2	-	
FMOV (Rm+),FDn	-	-	1	4	
FMOV (Rm+,imm8),FDn	-	-	1	4	

MN103E SERIES THROUGHPUT & LATENCY

Assembly mnemonic	AM33-1		AM33-2/AM33-2A		NOTE
	Throughput	Latency	Throughput	Latency	
FMOV (Rm+,imm24),FDn			2	5	When Throughput is 1,the divergence is failure.
FMOV (Rm+,imm32),FDn			2	5	
FMOV FDm,(Rn+)			1	-	
FMOV FDm,(Rn+,imm8)			1	-	
FMOV FDm,(Rn+,imm24)			2	-	
FMOV FDm,(Rn+,imm32)			2	-	
FBEQ (d8,PC)			3	-	
FBNE (d8,PC)			3	-	
FBGT (d8,PC)			3	-	
FBGE (d8,PC)			3	-	
FBLT (d8,PC)			3	-	
FBLE (d8,PC)			3	-	
FBUO (d8,PC)			3	-	
FBLG (d8,PC)			3	-	
FBLEG (d8,PC)			3	-	
FBUG (d8,PC)			3	-	
FBUGE (d8,PC)			3	-	
FBUL (d8,PC)			3	-	
FBULE (d8,PC)			3	-	
FBUE (d8,PC)			3	-	
FLEQ			1	-	Throughput is 2 if branch is not taken.
FLNE			1	-	
FLGT			1	-	
FLGE			1	-	
FLLT			1	-	
FLLE			1	-	
FLUO			1	-	
FLLG			1	-	
FLLEG			1	-	
FLUG			1	-	
FLUGE			1	-	
FLUL			1	-	
FLULE			1	-	
FLUE			1	-	

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/N		
MOV	MOV Am->An	Am->An	-	-	-	1 1001 AmAn	
	MOV Dm->Dn	Dm->Dn	-	-	-	1 1000 DmDn	
	MOV Am,Dn	Am->Dn	-	-	-	2 1111 0001 1101 AmDn	
	MOV Dm,An	Dm->An	-	-	-	2 1111 0001 1110 DmAn	
	MOV Am,Rn	Am->Rn	-	-	-	2 1111 0101 00Am Rn..	
	MOV Rm,An	Rm->An	-	-	-	2 1111 0101 10Rm .An	
	MOV Dm,Rn	Dm->Rn	-	-	-	2 1111 0101 01Dm Rn..	
	MOV Rm,Dn	Rm->Dn	-	-	-	2 1111 0101 11Rm .Dn	
	MOV Rm,Rn	Rm->Rn	-	-	-	3 1111 1001 0000 1000 Rm.. Rn..	
	MOV imm8,An	(zero_ext)imm8->An	-	-	-	2 1001 AnAn imm8....	
	MOV imm16,An	(zero_ext)imm16->An	-	-	-	3 0010 01An imm16...	
	MOV imm32,An	imm32->An	-	-	-	6 1111 1100 1101 11An imm32...	
	MOV imm8,Dn	(sign_ext)imm8->Dn	-	-	-	2 1000 DnDnimm8....	
	MOV imm16,Dn	(sign_ext)imm16->Dn	-	-	-	3 0010 11Dn imm16...	
	MOV imm32,Dn	imm32->Dn	-	-	-	6 1111 1100 1100 11Dn imm32...	
	MOV imm8,Rn	(sign_ext)imm8->Rn	-	-	-	4 1111 1011 0000 1000 Rn.. ---- imm8....	
	MOV imm24,Rn	(sign_ext)imm24->Rn	-	-	-	6 1111 1101 0000 1000 Rn.. ---- imm24...	
	MOV imm32,Rn	imm32->Rn	-	-	-	7 1111 1110 0000 1000 Rn.. ---- imm32...	
	MOV MDR,Dn	MDR->Dn	-	-	-	2 1111 0010 1110 00Dn	
	MOV Dm,MDR	Dm->MDR	-	-	-	2 1111 0010 1111 Dm10	
	MOV SP,An	SP->An	-	-	-	1 0011 11An	
	MOV SP,Rn	SP->Rn	-	-	-	3 1111 1001 1110 1000 0000 Rn..	
	MOV Am,SP	Am->SP	-	-	-	2 1111 0010 1111 Am00	
	MOV Rm,SP	Rm->SP	-	-	-	3 1111 1001 1110 1000 0000 Rn..	
	MOV imm8,SP	(zero_ext)imm8->SP	-	-	-	4 1111 1011 1111 1000 0000 imm8....	
	MOV imm24,SP	(zero_ext)imm24->SP	-	-	-	6 1111 1101 1111 1000 0000 imm24...	
	MOV imm32,SP	imm32->SP	-	-	-	7 1111 1110 1111 1000 0000 imm32...	
	MOV MDRQ,Rn	MDRQ->Rn	0	0	Δ	3 1111 1001 1110 1000 0001 Rn..	
	MOV Rm,MDRQ	Rm->MDRQ	-	-	-	3 1111 1001 1111 1000 Rm.. 0001	
	MOV imm8,MDRQ	(zero_ext)imm8->MDRQ	-	-	-	4 1111 1011 1111 1000 0001 imm8....	
	MOV imm24,MDRQ	(zero_ext)imm24->MDRQ	-	-	-	6 1111 1101 1111 1000 0001 imm24...	
	MOV imm32,MDRQ	imm32->MDRQ	-	-	-	7 1111 1110 1111 1000 0001 imm32...	
	MOV MCRH,Rn	MCRH->Rn, MCVF->EPSW.V	-	-	-	3 1111 1001 1110 1000 0010 Rn..	
	MOV Rm,MCRH	Rm->MCRH, EPSW.V->MCVF	-	-	-	3 1111 1001 1111 1000 Rm.. 0010	
	MOV imm8,MCRH	(zero_ext)imm8->MCRH, EPSW.V->MCVF	-	-	-	4 1111 1011 1111 1000 0010 imm8....	
	MOV imm24,MCRH	(zero_ext)imm24->MCRH, EPSW.V->MCVF	-	-	-	6 1111 1101 1111 1000 0010 imm24...	

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code	
			V	F	C			
MOV	MOV imm32, MCRH	imm32->MCRH, EPSW.V->MCVF	-	-	-	7	1111 1110 1111 1000	0010 imm32...
	MOV MCRL, Rn	MCRL->Rn, MCVF->EPSW.V	Δ	0	?	3	1111 1001 1110 1000 0011 Rn..	
	MOV Rm, MCRL	Rm->MCRL, EPSW.V->MCVF	-	-	-	3	1111 1001 1111 1000 Rm..	0011
	MOV imm8, MCRL	(zero_ext)imm8->MCRL, EPSW.V->MCVF	-	-	-	4	1111 1011 1111 1000	0011 imm8...
	MOV imm24, MCRL	(zero_ext)imm24->MCRL, EPSW.V->MCVF	-	-	-	6	1111 1101 1111 1000	0011 imm24...
	MOV imm32, MCRL	imm32->MCRL, EPSW.V->MCVF	-	-	-	7	1111 1110 1111 1000	0011 imm32...
	MOV MCVF, Rn	MCVF->Rn[0], 0x00000000->Rn[31:1]	-	-	-	3	1111 1001 1110 1000 0100 Rn..	
	MOV Rm, MCVF	Rm[0]->MCVF	-	-	-	3	1111 1001 1111 1000 Rm..	0100
	MOV imm8, MCVF	imm8[0]->MCVF	-	-	-	4	1111 1011 1111 1000	0100 imm8...
	MOV imm24, MCVF	imm24[0]->MCVF	-	-	-	6	1111 1101 1111 1000	0100 imm24...
	MOV imm32, MCVF	imm32[0]->MCVF	-	-	-	7	1111 1110 1111 1000	0100 imm32...
	MOV PC, An	PC->An	-	-	-	2	1111 0000 0010 11An	
	MOV (Am), Dn	mem32(Am)->Dn	-	-	-	1	0111 DnAm	
	MOV (Am), An	mem32(Am)->An	-	-	-	2	1111 0000 0000 AnAm	
	MOV (Rm), Rn	mem32(Rm)->Rn	-	-	-	3	1111 1001 0000 1010 Rn.. Rm..	
	MOV (d8, Am), Dn	mem32(Am+(sign_ext)d8)->Dn	-	-	-	3	1111 1000 0000 DnAm d8.....	
	MOV (d16, Am), Dn	mem32(Am+(sign_ext)d16)->Dn	-	-	-	4	1111 1010 0000 DnAm d16.....	
	MOV (d32, Am), Dn	mem32(Am+d32)->Dn	-	-	-	6	1111 1100 0000 DnAm d32.....	
	MOV (d8, Am), An	mem32(Am+(sign_ext)d8)->An	-	-	-	3	1111 1000 0010 AnAm d8.....	
	MOV (d16, Am), An	mem32(Am+(sign_ext)d16)->An	-	-	-	4	1111 1010 0010 AnAm d16.....	
	MOV (d32, Am), An	mem32(Am+d32)->An	-	-	-	6	1111 1100 0010 AnAm d32.....	
	MOV (d8, Rm), Rn	mem32(Rm+(sign_ext)d8)->Rn	-	-	-	4	1111 1011 0000 1010 Rn.. Rm.. d8.....	
	MOV (d24, Rm), Rn	mem32(Rm+(sign_ext)d24)->Rn	-	-	-	6	1111 1101 0000 1010 Rn.. Rm.. d24.....	
	MOV (d32, Rm), Rn	mem32(Rm+d32)->Rn	-	-	-	7	1111 1110 0000 1010 Rn.. Rm.. d32.....	
	MOV (Di, Am), Dn	mem32(Di+Am)->Dn	-	-	-	2	1111 0011 00Dn DiAm	
	MOV (Di, Am), An	mem32(Di+Am)->An	-	-	-	2	1111 0011 10An DiAm	
	MOV (Ri, Rm), Rn	mem32(Ri+Rm)->Rn	-	-	-	4	1111 1011 1000 1110 Ri.. Rm.. Rn.. ----	
	MOV (abs16), Dn	mem32((zero_ext)abs16)->Dn	-	-	-	3	0011 00Dn abs16....	
	MOV (abs32), Dn	mem32(abs32)->Dn	-	-	-	6	1111 1100 1010 01Dn abs32...	
	MOV (abs16), An	mem32((zero_ext)abs16)->An	-	-	-	4	1111 1010 1010 00An abs16...	
	MOV (abs32), An	mem32(abs32)->An	-	-	-	6	1111 1100 1010 00An abs32...	
	MOV (abs8), Rn	mem32((zero_ext)abs8)->Rn	-	-	-	4	1111 1011 0000 1110 Rn.. ---- abs8...	
	MOV (abs24), Rn	mem32((zero_ext)abs24)->Rn	-	-	-	6	1111 1101 0000 1110 Rn.. ---- abs24...	
	MOV (abs32), Rn	mem32(abs32)->Rn	-	-	-	7	1111 1110 0000 1110 Rn.. ---- abs32...	
	MOV (d8, SP), Dn	mem32(SP+(zero_ext)d8)->Dn	-	-	-	2	0101 10Dn d8.....	

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code	Machine Code
			V	F	C		
			N	Z	F	Size	
MOV	MOV (d16,SP),Dn	mem32(SP+(zero_ext)d16)->Dn	-	-	-	4	1111 1010 1011 01Dn d16.....
	MOV (d32,SP),Dn	mem32(SP+d32)->Dn	-	-	-	6	1111 1100 1011 01Dn d32.....
	MOV (d8,SP),An	mem32(SP+(zero_ext)d8)->An	-	-	-	2	0101 11An d8.....
	MOV (d16,SP),An	mem32(SP+(zero_ext)d16)->An	-	-	-	4	1111 1010 1011 00An d16.....
	MOV (d32,SP),An	mem32(SP+d32)->An	-	-	-	6	1111 1100 1011 00An d32.....
	MOV (SP),Rn	mem32(SP)->Rn	-	-	-	3	1111 1001 1000 1010 Rn., ----
	MOV (d8,SP),Rn	mem32(SP+(zero_ext)d8)->Rn	-	-	-	4	1111 1011 1000 1010 Rn., ---- d8.....
	MOV (d24,SP),Rn	mem32(SP+(zero_ext)d24)->Rn	-	-	-	6	1111 1101 1000 1010 Rn., ---- d24.....
	MOV (d32,SP),Rn	mem32(SP+d32)->Rn	-	-	-	7	1111 1110 1000 1010 Rn., ---- d32.....
	MOV (d8,Am),SP	mem32(Am+(sign_ext)d8)	-	-	-	3	1111 1000 1111 00Am d8.....
	MOV Dm,(An)	Dm->mem32(An)	-	-	-	1	0110 DmAn
	MOV Am,(An)	Am->mem32(An)	-	-	-	2	1111 0000 0001 AmAn
	MOV Rm,(Rn)	Rm->mem32(Rn)	-	-	-	3	1111 1001 0001 1010 Rm., Rn.,
	MOV Dm,(d8,An)	Dm->mem32(An+(sign_ext)d8)	-	-	-	3	1111 1000 0001 DmAn d8.....
	MOV Dm,(d16,An)	Dm->mem32(An+(sign_ext)d16)	-	-	-	4	1111 1010 0001 DmAn d16.....
	MOV Dm,(d32,An)	Dm->mem32(An+d32)	-	-	-	6	1111 1100 0001 DmAn d32.....
	MOV Am,(d8,An)	Am->mem32(An+(sign_ext)d8)	-	-	-	3	1111 1000 0011 AmAn d8...•c
	MOV Am,(d16,An)	Am->mem32(An+(sign_ext)d16)	-	-	-	4	1111 1010 0011 AmAn d16.....
	MOV Am,(d32,An)	Am->mem32(An+d32)	-	-	-	6	1111 1100 0011 AmAn d32.....
	MOV Rm,(d8,Rn)	Rm->mem32(Rn+(sign_ext)d8)	-	-	-	4	1111 1011 0001 1010 Rm., Rn., d8.....
	MOV Rm,(d24,Rn)	Rm->mem32(Rn+(sign_ext)d24)	-	-	-	6	1111 1101 0001 1010 Rm., Rn., d24.....
	MOV Rm,(d32,Rn)	Rm->mem32(Rn+d32)	-	-	-	7	1111 1110 0001 1010 Rm., Rn., d32.....
	MOV Dm,(Di,An)	Dm->mem32(Di+An)	-	-	-	2	1111 0011 01Dm DiAn
	MOV Am,(Di,An)	Am->mem32(Di+An)	-	-	-	2	1111 0011 11Am DiAn
	MOV Rm,(Ri,Rn)	Rm->mem32(Ri+Rn)	-	-	-	4	1111 1011 1001 1110 Ri., Rn., ----
	MOV Dm,(abs16)	Dm->mem32((zero_ext)abs16)	-	-	-	3	0000 Dm01 abs16...
	MOV Dm,(abs32)	Dm->mem32((zero_ext)abs32)	-	-	-	6	1111 1100 1000 Dm01 abs32...
	MOV Am,(abs16)	Am->mem32((zero_ext)abs16)	-	-	-	4	1111 1010 1000 Am00 abs16...
	MOV Am,(abs32)	Am->mem32((zero_ext)abs32)	-	-	-	6	1111 1100 1000 Am00 abs32...
	MOV Rm,(abs8)	Rm->mem32((zero_ext)abs8)	-	-	-	4	1111 1011 0001 1110 Rm., ---- abs8.....
	MOV Rm,(abs24)	Rm->mem32((zero_ext)abs24)	-	-	-	6	1111 1101 0001 1110 Rm., ---- abs24...
	MOV Rm,(abs32)	Rm->mem32((zero_ext)abs32)	-	-	-	7	1111 1110 0001 1110 Rm., ---- abs32.....
	MOV Dm,(d8,SP)	Dm->mem32(SP+(zero_ext)d8)	-	-	-	2	0100 Dm10 d8.....
	MOV Dm,(d16,SP)	Dm->mem32(SP+(zero_ext)d16)	-	-	-	4	1111 1010 1001 Dm01 d16.....
	MOV Dm,(d32,SP)	Dm->mem32(SP+d32)	-	-	-	6	1111 1100 1001 Dm01 d32.....
	MOV Am,(d8,SP)	Am->mem32(SP+(zero_ext)d8)	-	-	-	2	0100 Am11 d8.....

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code	Machine Code
			V	F	C/N	ZF	
						Size	
MOV	MOV Am,(d16,SP)	Am->mem32(SP+(zero_ext)d16)	-	-	-	4	1111 1010 1001 Am00 d16.....
	MOV Am,(d32,SP)	Am->mem32(SP+d32)	-	-	-	6	1111 1100 1001 Am00 d32.....
	MOV Rm,(SP)	Rm->mem32(SP)	-	-	-	3	1111 1001 1001 1010 Rm.....
	MOV Rm,(d8,SP)	Rm->mem32(SP+(zero_ext)d8)	-	-	-	4	1111 1011 1001 1010 Rm..... d8.....
	MOV Rm,(d24,SP)	Rm->mem32(SP+(zero_ext)d24)	-	-	-	6	1111 1101 1001 1010 Rm..... d24.....
	MOV Rm,(d32,SP)	Rm->mem32(SP+d32)	-	-	-	7	1111 1110 1001 1010 Rm..... d32.....
	MOV SP,(d8,An)	SP->mem32(An+(sign_ext)d8)	-	-	-	3	1111 1000 1111 01An d8.....
	MOV (Rm+,imm8),Rn	mem32(Rm)->Rn,Rm+(sign_ext)imm8->Rm	-	-	-	4	1111 1011 0110 1010 Rn.. Rm.. imm8.....
	MOV (Rm+,imm24),Rn	mem32(Rm)->Rn,Rm+(sign_ext)imm24->Rm	-	-	-	6	1111 1101 0110 1010 Rn.. Rm.. imm24...
	MOV (Rm+,imm32),Rn	mem32(Rm)->Rn,Rm+imm32->Rm	-	-	-	7	1111 1110 0110 1010 Rn.. Rm.. imm32...
	MOV (Rm+),Rn	mem32(Rm)->Rn,Rm+4->Rm	-	-	-	3	1111 1001 0110 1010 Rn.. Rm..
	MOV Rm,(Rn+,imm8)	Rm->mem32(Rn),Rn+(sign_ext)imm8->Rn	-	-	-	4	1111 1011 0111 1010 Rm.. Rn.. imm8.....
	MOV Rm,(Rn+,imm24)	Rm->mem32(Rn),Rn+(sign_ext)imm24->Rn	-	-	-	6	1111 1101 0111 1010 Rm.. Rn.. imm24...
	MOV Rm,(Rn+,imm32)	Rm->mem32(Rn),Rn+imm32->Rn	-	-	-	7	1111 1110 0111 1010 Rm.. Rn.. imm32...
	MOV Rm,(Rn+)	Rm->mem32(Rn),Rn+4->Rn	-	-	-	3	1111 1001 0111 1010 Rm.. Rn..
	MOV MSP,An	MSP->An	-	-	-	2	1111 0000 0010 10An
	MOV Am,MSP	Am->MSP	-	-	-	2	1111 0000 0011 Am10
	MOV SSP,An	SSP->An	-	-	-	2	1111 0000 0010 01An
	MOV Am,SSP	Am->SSP	-	-	-	2	1111 0000 0011 Am01
	MOV USP,An	USP->An	-	-	-	2	1111 0000 0010 00An
MOVU	MOV Am,USP	Am->USP	-	-	-	2	1111 0000 0011 Am00
	MOV EPSW,Dn	EPSW->Dn	-	-	-	2	1111 0010 1110 11Dn
	MOV Dm,EPSPW	Dm->EPSPW	-	-	-	2	1111 0010 1111 Dm01
	MOV PSW,Dn	PSW->Dn	-	-	-	2	1111 0010 1110 01Dn
	MOV Dm,PSW	Dm->PSW	-	-	-	2	1111 0010 1111 Dm11
	MOVU imm8,Rn	(zero_ext)imm8->Rn	-	-	-	3	1111 1011 0001 1000 Rn.. ---- imm8.....
	MOVU imm24,Rn	(zero_ext)imm24->Rn	-	-	-	6	1111 1101 0001 1000 Rn.. ---- imm24...
	MOVU imm32,Rn	imm32->Rn	-	-	-	7	1111 1110 0001 1000 Rn.. ---- imm32.....
	MOVHU (Am),Dn	mem16(Am)->Dn[15:0],0x0000->Dn[31:16]	-	-	-	2	1111 0000 0110 DnAm
	MOVHU (Rm),Rn	mem16(Rm)->Rn[15:0],0x0000->Rn[31:16]	-	-	-	3	1111 1001 0100 1010 Rn.. Rm..
MOVHU	MOVHU (d8,Am),Dn	mem16(Am+(sign_ext)d8)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	3	1111 1000 0110 DnAm d8.....
	MOVHU (d16,Am),Dn	mem16(Am+(sign_ext)d16)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	4	1111 1010 0110 DnAm d16.....
	MOVHU (d32,Am),Dn	mem16(Am+d32)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	6	1111 1100 0110 DnAm d32.....
			-	-	-		

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/N		
MOVHU	MOVHU (d8,Rm),Rn	mem16(Rm+(sign_ext)d8)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	4	1111 1011 0100 1010 Rn.. Rm.. d8.....
	MOVHU (d24,Rm),Rn	mem16(Rm+(sign_ext)d16)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	6	1111 1101 0100 1010 Rn.. Rm.. d24.....
	MOVHU (d32,Rm),Rn	mem16(Rm+d32)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	7	1111 1110 0100 1010 Rn.. Rm.. d32.....
	MOVHU (Di,Am),Dn	mem16(Di+Am)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	2	1111 0100 10Dn DiAm
	MOVHU (Ri,Rm),Rn	mem16(Ri+Rm)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	4	1111 1011 1100 1110 Ri.. Rm.. Rn.. ----
	MOVHU (abs16),Dn	mem16((zero_ext)abs16)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	3	0011 10Dn abs16...
	MOVHU (abs32),Dn	mem16(abs32)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	6	1111 1100 1010 11Dn abs32...
	MOVHU (abs8),Rn	mem16((zero_ext)abs8)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	4	1111 1011 0100 1110 Rn.. ---- abs8....
	MOVHU (abs24),Rn	mem16((zero_ext)abs24)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	6	1111 1101 0100 1110 Rn.. ---- abs24...
	MOVHU (abs32),Rn	mem16(abs32)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	7	1111 1110 0100 1110 Rn.. ---- abs32...
	MOVHU (d8,SP),Dn	mem16(SP+(zero_ext)d8)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	3	1111 1000 1011 11Dn d8.....
	MOVHU (d16,SP),Dn	mem16(SP+(zero_ext)d16)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	4	1111 1010 1011 11Dn d16.....
	MOVHU (d32,SP),Dn	mem16(SP+d32)->Dn[15:0], 0x0000->Dn[31:16]	-	-	-	6	1111 1100 1011 11Dn d32.....
	MOVHU (SP),Rn	mem16(SP)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	3	1111 1001 1100 1010 Rn.. ----
	MOVHU (d8,SP),Rn	mem16(SP+(zero_ext)d8)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	4	1111 1011 1100 1010 Rn.. ---- d8.....
	MOVHU (d24,SP),Rn	mem16(SP+(zero_ext)d24)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	6	1111 1101 1100 1010 Rn.. ---- d24.....
	MOVHU (d32,SP),Rn	mem16(SP+d32)->Rn[15:0], 0x0000->Rn[31:16]	-	-	-	7	1111 1110 1100 1010 Rn.. ---- d32.....
	MOVHU Dm,(An)	Dm[15:0]->mem16(An)	-	-	-	2	1111 0000 0111 DmAn
	MOVHU Rm,(Rn)	Rm[15:0]->mem16(Rn)	-	-	-	3	1111 1001 0101 1010 Rm.. Rn..

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code	Machine Code
			V	F	C/N	ZF	
MOVHU	MOVHU Dm,(d8,An)	Dm[15:0]->mem16(An+(sign_ext)d8)	-	-	-	3	1111 1000 0111 DmAn d8.....
	MOVHU Dm,(d16,An)	Dm[15:0]->mem16(An+(sign_ext)d16)	-	-	-	4	1111 1010 0111 DmAn d16.....
	MOVHU Dm,(d32,An)	Dm[15:0]->mem16(An+d32)	-	-	-	6	1111 1100 0111 DmAn d32.....
	MOVHU Rm,(d8,Rn)	Rm[15:0]->mem16(Rn+(sign_ext)d8)	-	-	-	4	1111 1011 0101 1010 Rm.. Rn.. d8.....
	MOVHU Rm,(d24,Rn)	Rm[15:0]->mem16(Rn+(sign_ext)d24)	-	-	-	6	1111 1101 0101 1010 Rm.. Rn.. d24.....
	MOVHU Rm,(d32,Rn)	Rm[15:0]->mem16(Rn+d32)	-	-	-	7	1111 1110 0101 1010 Rm.. Rn.. d32.....
	MOVHU Dm,(Di,An)	Dm[15:0]->mem16(Di+An)	-	-	-	2	1111 0100 11Dm DiAn
	MOVHU Rm,(Ri,Rn)	Rm[15:0]->mem16(Ri+Rn)	-	-	-	4	1111 1011 1101 1110 Ri.. Rn.. Rm.. ----
	MOVHU Dm,(abs16)	Dm[15:0]->mem16((zero_ext)abs16)	-	-	-	3	0000 Dm11 abs16...
	MOVHU Dm,(abs32)	Dm[15:0]->mem16(abs32)	-	-	-	6	1111 1100 1000 Dm11 abs32...
	MOVHU Rm,(abs8)	Rm[15:0]->mem16((zero_ext)abs8)	-	-	-	4	1111 1011 0101 1110 Rm.. ---- abs8....
	MOVHU Rm,(abs24)	Rm[15:0]->mem16((zero_ext)abs24)	-	-	-	6	1111 1101 0101 1110 Rm.. ---- abs24...
	MOVHU Rm,(abs32)	Rm[15:0]->mem16(abs32)	-	-	-	7	1111 1110 0101 1110 Rm.. ---- abs32...
	MOVHU Dm,(d8,SP)	Dm[15:0]->mem16(SP+(zero_ext)d8)	-	-	-	3	1111 1000 1001 Dm11 d8.....
	MOVHU Dm,(d16,SP)	Dm[15:0]->mem16(SP+(zero_ext)d16)	-	-	-	4	1111 1010 1001 Dm11 d16.....
	MOVHU Dm,(d32,SP)	Dm[15:0]->mem16(SP+d32)	-	-	-	6	1111 1100 1001 Dm11 d32.....
	MOVHU Rm,(SP)	Rm[15:0]->mem16(SP)	-	-	-	3	1111 1001 1101 1010 Rm.. ----
	MOVHU Rm,(d8,SP)	Rm[15:0]->mem16(SP+(zero_ext)d8)	-	-	-	4	1111 1011 1101 1010 Rm.. ---- d8.....
	MOVHU Rm,(d24,SP)	Rm[15:0]->mem16(SP+(zero_ext)d24)	-	-	-	6	1111 1101 1101 1010 Rm.. ---- d24.....
	MOVHU Rm,(d32,SP)	Rm[15:0]->mem16(SP+d32)	-	-	-	7	1111 1110 1101 1010 Rm.. ---- d32.....
	MOVHU (Rm+,imm8),Rn	mem16(Rm)->Rn[15:0], 0x0000->Rn[31:16],Rm+(sign_ext)imm8->Rm	-	-	-	4	1111 1011 1110 1010 Rn.. Rm.. imm8....
	MOVHU (Rm+,imm24),Rn	mem16(Rm)->Rn[15:0], 0x0000->Rn[31:16],Rm+(sign_ext)imm24->Rm	-	-	-	6	1111 1101 1110 1010 Rn.. Rm.. imm24...
	MOVHU (Rm+,imm32),Rn	mem16(Rm)->Rn[15:0], 0x0000->Rn[31:16],Rm+imm32->Rm	-	-	-	7	1111 1110 1110 1010 Rn.. Rm.. imm32...
	MOVHU (Rm+),Rn	mem16(Rm)->Rn[15:0], 0x0000->Rn[31:16],Rm+2->Rm	-	-	-	3	1111 1001 1110 1010 Rn.. Rm..
	MOVHU Rm,(Rn+,imm8)	Rm[15:0]->mem16(Rn), Rn+(sign_ext)imm8->Rn	-	-	-	4	1111 1011 1111 1010 Rm.. Rn.. imm8....
	MOVHU Rm,(Rn+,imm24)	Rm[15:0]->mem16(Rn), Rn+(sign_ext)imm24->Rn	-	-	-	6	1111 1101 1111 1010 Rm.. Rn.. imm24...
	MOVHU Rm,(Rn+,imm32)	Rm[15:0]->mem16(Rn),Rn+imm32->Rn	-	-	-	7	1111 1110 1111 1010 Rm.. Rn.. imm32...
	MOVHU Rm,(Rn+)	Rm[15:0]->mem16(Rn),Rn+2->Rn	-	-	-	3	1111 1001 1111 1010 Rm.. Rn..
MOVBU	MOVBU (Am),Dn	mem8(Am)->Dn[7:0],0x000000->Dn[31:8]	-	-	-	2	1111 0000 0100 DnAm
	MOVBU (Rm),Rn	mem8(Rm)->Rn[7:0],0x000000->Rn[31:8]	-	-	-	3	1111 1001 0010 1010 Rn.. Rm..

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/N		
MOVBU	MOVBU (d8,Am),Dn	mem8(Am+(sign_ext)d8)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	3	1111 1000 0100 DnAm d8.....
	MOVBU (d16,Am),Dn	mem8(Am+(sign_ext)d16)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	4	1111 1010 0100 DnAm d16.....
	MOVBU (d32,Am),Dn	mem8(Am+d32)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	6	1111 1100 0100 DnAm d32.....
	MOVBU (d8,Rm),Rn	mem8(Rm+(sign_ext)d8)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	4	1111 1011 0010 1010 Rn., Rm., d8.....
	MOVBU (d24,Rm),Rn	mem8(Rm+(sign_ext)d24)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	6	1111 1101 0010 1010 Rn., Rm., d24.....
	MOVBU (d32,Rm),Rn	mem8(Rm+d32)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	7	1111 1110 0010 1010 Rn., Rm., d32.....
	MOVBU (Dl,Am),Dn	mem8(Dl+Am)->Dn[7:0],0x000000->Dn[31:8]	-	-	-	2	1111 0100 00Dn DlAm
	MOVBU (Ri,Rm),Rn	mem8(Ri+Rm)->Rn[7:0],0x000000->Rn[31:8]	-	-	-	4	1111 1011 1010 1110 Ri., Rm., Rn., ----
	MOVBU (abs16),Dn	mem8((zero_ext)abs16)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	3	0011 01Dn abs16...
	MOVBU (abs32),Dn	mem8(abs32)->Dn[7:0],0x000000->Dn[31:8]	-	-	-	6	1111 1100 1010 10Dn abs32...
	MOVBU (abs8),Rn	mem8((zero_ext)abs8)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	4	1111 1011 0010 1110 Rn., ---- abs8....
	MOVBU (abs24),Rn	mem8((zero_ext)abs24)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	6	1111 1101 0010 1110 Rn., ---- abs24...
	MOVBU (abs32),Rn	mem8(abs32)->Rn[7:0],0x000000->Rn[31:8]	-	-	-	7	1111 1110 0010 1110 Rn., ---- abs32...
	MOVBU (d8,SP),Dn	mem8(SP+(zero_ext)d8)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	3	1111 1000 1011 10Dn d8.....
	MOVBU (d16,SP),Dn	mem8(SP+(zero_ext)d16)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	4	1111 1010 1011 10Dn d16....
	MOVBU (d32,SP),Dn	mem8(SP+d32)->Dn[7:0], 0x000000->Dn[31:8]	-	-	-	6	1111 1100 1011 10Dn d32.....
	MOVBU (SP),Rn	mem8(SP)->Rn[7:0],0x000000->Rn[31:8]	-	-	-	3	1111 1001 1010 1010 Rn., ----
	MOVBU (d8,SP),Rn	mem8(SP+(zero_ext)d8)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	4	1111 1011 1010 1010 Rn., ---- d8.....
	MOVBU (d24,SP),Rn	mem8(SP+(zero_ext)d24)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	6	1111 1101 1010 1010 Rn., ---- d24.....
	MOVBU (d32,SP),Rn	mem8(SP+d32)->Rn[7:0], 0x000000->Rn[31:8]	-	-	-	7	1111 1110 1010 1010 Rn., ---- d32.....
	MOVBU Dm,(An)	Dm[7:0]->mem8(An)	-	-	-	2	1111 0000 0101 DmAn

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/N		
MOVBU	MOVBU Rm,(Rn)	Rm[7:0]->mem8(Rn)	-	-	-	3	1111 1001 0011 1010 Rm.. Rn..
	MOVBU Dm,(d8,An)	Dm[7:0]->mem8(An+(sign_ext)d8)	-	-	-	3	1111 1000 0101 DmAn d8.....
	MOVBU Dm,(d16,An)	Dm[7:0]->mem8(An+(sign_ext)d16)	-	-	-	4	1111 1010 0101 DmAn d16.....
	MOVBU Dm,(d32,An)	Dm[7:0]->mem8(An+d32)	-	-	-	6	1111 1100 0101 DmAn d32.....
	MOVBU Rm,(d8,Rn)	Rm[7:0]->mem8(Rn+(sign_ext)d8)	-	-	-	4	1111 1011 0011 1010 Rm.. Rn.. d8.....
	MOVBU Rm,(d24,Rn)	Rm[7:0]->mem8(Rn+(sign_ext)d24)	-	-	-	6	1111 1101 0011 1010 Rm.. Rn.. d24.....
	MOVBU Rm,(d32,Rn)	Rm[7:0]->mem8(Rn+d32)	-	-	-	7	1111 1110 0011 1010 Rm.. Rn.. d32.....
	MOVBU Dm,(Di,An)	Dm[7:0]->mem8(Di+An)	-	-	-	2	1111 0100 01Dm DiAn
	MOVBU Rm,(Ri,Rn)	Rm[7:0]->mem8(Ri+Rn)	-	-	-	4	1111 1011 1011 1110 Ri.. Rn.. Rm.. ----
	MOVBU Dm,(abs16)	Dm[7:0]->mem8((zero_ext)abs16)	-	-	-	3	0000 Dm10 abs16...
	MOVBU Dm,(abs32)	Dm[7:0]->mem8(abs32)	-	-	-	6	1111 1100 1000 Dm10 abs32...
	MOVBU Rm,(abs8)	Rm[7:0]->mem8((zero_ext)abs8)	-	-	-	4	1111 1011 0011 1110 Rm.. ---- abs8.....
	MOVBU Rm,(abs24)	Rm[7:0]->mem8((zero_ext)abs24)	-	-	-	6	1111 1101 0011 1110 Rm.. ---- abs24...
	MOVBU Rm,(abs32)	Rm[7:0]->mem8(abs32)	-	-	-	7	1111 1110 0011 1110 Rm.. ---- abs32...
	MOVBU Dm,(d8,SP)	Dm[7:0]->mem8(SP+(zero_ext)d8)	-	-	-	3	1111 1000 1001 Dm10 d8.....
	MOVBU Dm,(d16,SP)	Dm[7:0]->mem8(SP+(zero_ext)d16)	-	-	-	4	1111 1010 1001 Dm10 d16.....
	MOVBU Dm,(d32,SP)	Dm[7:0]->mem8(SP+d32)	-	-	-	6	1111 1100 1001 Dm10 d32.....
	MOVBU Rm,(SP)	Rm[7:0]->mem8(SP)	-	-	-	3	1111 1001 1011 1010 Rm.. ----
	MOVBU Rm,(d8,SP)	Rm[7:0]->mem8(SP+(zero_ext)d8)	-	-	-	4	1111 1011 1011 1010 Rm.. ---- d8.....
	MOVBU Rm,(d24,SP)	Rm[7:0]->mem8(SP+(zero_ext)d24)	-	-	-	6	1111 1101 1011 1010 Rm.. ---- d24.....
MOV	MOVBU Rm,(d32,SP)	Rm[7:0]->mem8(SP+d32)	-	-	-	7	1111 1110 1011 1010 Rm.. ---- d32.....
	MOV M (SP),[reg1,...,regN]	See "MOV M" page	-	-	-	2	1100 1110 regs.....
	MOV M [reg1,...,regN],(SP)	See "MOV M" page	-	-	-	2	1100 1111 regs.....
	MOV M (USP),[reg1,...,regN]	See "MOV M" page	-	-	-	3	1111 1000 1100 1110 regs....
EXT	EXT Dn	See "MOV M" page	-	-	-	3	1111 1000 1100 1111 regs....
	EXT Rn	(sign_ext)(Dn)->{MDR,Dn} if(Rn.bp31=0)0x0->MDR ,else 0xFFFFFFF->MDR	-	-	-	2	1111 0010 1101 00Dn
	EXTH Dn	if(Dn.bp15=0)Dn&0xFFFF->Dn ,else Dn 0xFFFF0000->Dn	-	-	-	1	0001 10Dn
EXTH	EXTH Rn	if(Rn.bp15=0)Rn&0xFFFF->Dn ,else Rn 0xFFFF0000->Rn	-	-	-	3	1111 1001 0100 1000 Rn.. Rn..
	EXTH Rm,Rn	if(Rm.bp15=0)Rm&0xFFFF->Rn ,else Rm 0xFFFF0000->Rn	-	-	-	3	1111 1001 0100 1000 Rm.. Rn..
	EXTHU Dn	Dn&0xFFFF->Dn	-	-	-	1	0001 11Dn
EXTHU	EXTHU Rn	Rn&0xFFFF->Rn	-	-	-	3	1111 1001 0101 1000 Rm.. Rn..

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/N		
EXTBU	EXTBU Rm,Rn	Rm&0xFFFF->Rn	-	-	-	3	1111 1001 0101 1000 Rm.. Rn..
	EXTB Dn	if(Dn.bp7=0)Dn&0xFF->Dn ,else Dn 0xFF000000->Dn	-	-	-	1	0001 00Dn
	EXTB Rn	if(Rn.bp7=0)Rn&0xFF->Dn ,else Rn 0xFF000000->Rn	-	-	-	3	1111 1001 0010 1000 Rn.. Rn..
	EXTB Rm,Rn	if(Rm.bp7=0)Rm&0xFF->Rn ,else Rm 0xFF000000->Rn	-	-	-	3	1111 1001 0010 1000 Rm.. Rn..
EXTBU	EXTBU Dn	Dn&0xFF->Dn	-	-	-	1	0001 01Dn
	EXTBU Rn	Rn&0xFF->Rn	-	-	-	3	1111 1001 0011 1000 Rn.. Rn..
	EXTBU Rm,Rn	Rm&0xFF0->Rn	-	-	-	3	1111 1001 0011 1000 Rm.. Rn..
	CLR Dn	0x00000000->Dn	-	-	-	1	0000 Dn00
CLR	CLR Rn	0x00000000->Rn	-	-	-	3	1111 1001 0110 1000 ---- Rn..
	DCPF (Rm)	mem_cline(Rm)->[DataCache]	-	-	-	3	1111 1001 1010 0110 Rm.. ----
	DCPF (d8,Rm)	mem_cline(Rm+(sign_ext)d8)->[DataCache]	-	-	-	4	1111 1011 1010 0111 Rm.. ---- d8.....
	DCPF (d24,Rm)	mem_cline(Rm+(sign_ext)d24)->[DataCache]	-	-	-	6	1111 1011 1010 0111 Rm.. ---- d24....
DCPF	DCPF (d32,Rm)	mem_cline(Rm+d32)->[DataCache]	-	-	-	7	1111 1110 0100 0110 Rm.. ---- d32....
	DCPF (Ri,Rm)	mem_cline(Rm+Ri)->[DataCache]	-	-	-	4	1111 1011 1010 0110 Ri.. Rm.. ---- ----
	DCPF (SP)	mem_cline(SP)->[DataCache]	-	-	-	3	1111 1001 1010 0111 ---- ----
	ADD Dm,Dn	Dm+Dn->Dn	Δ	Δ	Δ	1	1110 DmDn
ADD	ADD Dm,An	Dm+An->An	Δ	Δ	Δ	2	1111 0001 0110 DmAn
	ADD Am,Dn	Am+Dn->Dn	Δ	Δ	Δ	2	1111 0001 0101 AmDn
	ADD Am,An	Am+An->An	Δ	Δ	Δ	2	1111 0001 0111 AmAn
	ADD Rm,Rn	Rm+Rn->Rn	Δ	Δ	Δ	3	1111 1001 0111 1000 Rm.. Rn..
	ADD Rm,Rn,Rd	Rm+Rn->Rd	Δ	Δ	Δ	4	1111 1011 0111 1100 Rm.. Rn.. Rd.. ----
	ADD imm8,Dn	(sign_ext)imm8+Dn->Dn	Δ	Δ	Δ	2	0010 10Dn imm8....
	ADD imm16,Dn	(sign_ext)imm16+Dn->Dn	Δ	Δ	Δ	4	1111 1010 1100 00Dn imm16...
	ADD imm32,Dn	imm32+Dn->Dn	Δ	Δ	Δ	6	1111 1100 1100 00Dn imm32...
	ADD imm8,An	(sign_ext)imm8+An->An	Δ	Δ	Δ	2	0010 00An imm8....
	ADD imm16,An	(sign_ext)imm16+An->An	Δ	Δ	Δ	4	1111 1010 1101 00An imm16...
	ADD imm32,An	imm32+An->An	Δ	Δ	Δ	6	1111 1100 1101 00An imm32...
	ADD imm8,Rn	(sign_ext)imm8+Rn->Rn	Δ	Δ	Δ	4	1111 1011 0111 1000 Rn.. imm8....
	ADD imm24,Rn	(sign_ext)imm24+Rn->Rn	Δ	Δ	Δ	6	1111 1101 0111 1000 Rn.. imm24...
	ADD imm32,Rn	imm32+Rn->Rn	-	-	-	7	1111 1110 0111 1000 Rn.. imm32...
	ADD imm8,SP	(sign_ext)imm8+SP->SP	-	-	-	3	1111 1000 1111 1110 imm8....
	ADD imm16,SP	(sign_ext)imm16+SP->SP	-	-	-	4	1111 1010 1111 1110 imm16...
	ADD imm32,SP	imm32+SP->SP	Δ	Δ	Δ	6	1111 1100 1111 1110 imm32...

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/N		
ADDC	ADDC Dm, Dn	Dm+Dn+EPSW.C->Dn	Δ	Δ	Δ	2	1111 0001 0100 DmDn
	ADDC Rm, Rn	Rm+Rn+EPSW.C->Rn	Δ	Δ	Δ	3	1111 1001 1000 1000 Rm.. Rn..
	ADDC Rm, Rn, Rd	Rm+Rn+EPSW.C->Rd	Δ	Δ	Δ	4	1111 1011 1000 1100 Rm.. Rn.. Rd..
	ADDC imm8, Rn	(sign_ext)imm8+Rn+EPSW.C->Rn	Δ	Δ	Δ	4	1111 1011 1000 1000 Rn.. Rm.. imm8....
	ADDC imm24, Rn	(sign_ext)imm24+Rn+EPSW.C->Rn	Δ	Δ	Δ	6	1111 1101 1000 1000 Rn.. Rm.. imm24...
	ADDC imm32, Rn	imm32+Rn+EPSW.C->Rn	Δ	Δ	Δ	7	1111 1110 1000 1000 Rn.. Rm.. imm32...
	SUB Dm, Dn	Dn-Dm->Dn	Δ	Δ	Δ	2	1111 0001 0000 DmDn
SUB	SUB Dm, An	An-Dm->An	Δ	Δ	Δ	2	1111 0001 0010 DmAn
	SUB Am, Dn	Dn-Am->Dn	Δ	Δ	Δ	2	1111 0001 0001 AmDn
	SUB Am, An	An-Am->An	Δ	Δ	Δ	2	1111 0001 0011 AmAn
	SUB Rm, Rn	Rn-Rm->Rn	Δ	Δ	Δ	3	1111 1001 1001 1000 Rm.. Rn..
	SUB Rm, Rn, Rd	Rn-Rm->Rd	Δ	Δ	Δ	4	1111 1011 1001 1100 Rm.. Rn.. Rd..
	SUB imm32, Dn	Dn-imm32->Dn	Δ	Δ	Δ	6	1111 1100 1100 00Dn imm32...
	SUB imm32, An	An-imm32->An	Δ	Δ	Δ	6	1111 1100 1101 01An imm32...
SUBC	SUB imm8, Rn	Rn-(sign_ext)imm8->Rn	Δ	Δ	Δ	4	1111 1011 1001 1000 Rn.. Rm.. imm8....
	SUB imm24, Rn	Rn-(sign_ext)imm24->Rn	Δ	Δ	Δ	6	1111 1101 1001 1000 Rn.. Rm.. imm24...
	SUB imm32, Rn	Rn-imm32->Rn	Δ	Δ	Δ	7	1111 1110 1001 1000 Rn.. Rm.. imm32...
	SUBC Dm, Dn	Dn-Dm-EPSW.C->Dn	Δ	Δ	Δ	2	1111 0001 1000 DmDn
	SUBC Rm, Rn	Rn-Rm-EPSW.C->Rn	Δ	Δ	Δ	3	1111 1001 1010 1000 Rm.. Rn..
	SUBC Rm, Rn, Rd	Rn-Rm-EPSW.C->Rd	Δ	Δ	Δ	4	1111 1011 1010 1100 Rm.. Rn.. Rd..
	SUBC imm8, Rn	Rn-(sign_ext)imm8-EPSW.C->Rn	Δ	Δ	Δ	4	1111 1011 1010 1000 Rn.. Rm.. imm8....
MUL	SUBC imm24, Rn	Rn-(sign_ext)imm24-EPSW.C->Rn	Δ	Δ	Δ	6	1111 1101 1010 1000 Rn.. Rm.. imm24...
	SUBC imm32, Rn	Rn-imm32-EPSW.C->Rn	Δ	Δ	Δ	7	1111 1110 1010 1000 Rn.. Rm.. imm32...
	MUL Dm, Dn	Dm*Dn->{MDR, Dn}	?	?	?	2	1111 0010 0100 DmDn
	MUL Rm, Rn	Rm*Rn->{MDR, Rn}	?	?	?	3	1111 1001 1010 1001 Rm.. Rn..
	MUL Rm, Rn, Rd1, Rd2	Rm*Rn->{Rd1, Rd2}	?	?	?	4	1111 1011 1010 1101 Rm.. Rn.. Rd1. Rd2.
	MUL imm8, Rn	(sign_ext)imm8*Rn->{MDR, Rn}	?	?	?	4	1111 1011 1010 1001 Rn.. Rm.. imm8....
	MUL imm24, Rn	(sign_ext)imm24*Rn->{MDR, Rn}	?	?	?	6	1111 1101 1010 1001 Rn.. Rm.. imm24...
MULU	MUL imm32, Rn	imm32*Rn->{MDR, Rn}	?	?	?	7	1111 1110 1010 1001 Rn.. Rm.. imm32...
	MULU Dm, Dn	Dm*Dn->{MDR, Dn}	?	?	?	2	1111 0010 0101 DmDn
	MULU Rm, Rn	Rm*Rn->{MDR, Rn}	?	?	?	3	1111 1001 1011 1001 Rm.. Rn..
	MULU Rm, Rn, Rd1, Rd2	Rm*Rn->{MDR, Rn}	?	?	?	4	1111 1011 1011 1101 Rm.. Rn.. Rd1. Rd2.
	MULU imm8, Rn	(zero_ext)imm8*Rn->{MDR, Rn}	?	?	?	4	1111 1011 1011 1001 Rn.. Rm.. imm8....
	MULU imm24, Rn	(zero_ext)imm24*Rn->{MDR, Rn}	?	?	?	6	1111 1101 1011 1001 Rn.. Rm.. imm24...
	MULU imm32, Rn	imm32*Rn->{MDR, Rn}	?	?	?	7	1111 1110 1011 1001 Rn.. Rm.. imm32...
DIV	DIV Dm, Dn	{MDR, Dn}/Dm->Dn, {MDR, Dn}%Dm->MDR	0/1	?	Δ?	Δ?	2 1111 0010 0110 DmDn

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C		
DIVU	DIV Rm,Rn	{MDR,Rn}/Rm->Rn,{MDR,Rn}%Rm->MDR	0/1	?	Δ?	Δ?	3 1111 1001 1100 1001 Rm.. Rn..
	DIVU Dm,Dn	{MDR,Dn}/Dm->Dn,{MDR,Dn}%Dm->MDR	0/1	?	Δ?	Δ?	2 1111 0010 0111 DmDn
	DIVU Rm,Rn	{MDR,Rn}/Rm->Rn,{MDR,Rn}%Rm->MDR	0/1	?	Δ?	Δ?	3 1111 1001 1101 1001 Rm.. Rn..
INC	INC Dn	Dn+1->Dn	Δ	Δ	Δ	Δ	1 0100 Dn00
	INC An	An+1->An	-	-	-	-	1 0100 An01
	INC Rn	Rn+1->Rn	Δ	Δ	Δ	Δ	3 1111 1001 1011 1000 Rm.. Rn..
INC4	INC4 An	An+4->An	-	-	-	-	1 0101 00An
	INC4 Rn	Rn+4->Rn	Δ	Δ	Δ	Δ	3 1111 1001 1100 1000 Rm.. Rn..
	CMP Dm,Dn	Dn-Dm : EPSW	Δ	Δ	Δ	Δ	1 1010 DmDn
CMP	CMP Dm,An	An-Dm : EPSW	Δ	Δ	Δ	Δ	2 1111 0001 1010 DmAn
	CMP Am,Dn	Dn-Am : EPSW	Δ	Δ	Δ	Δ	2 1111 0001 1001 AmDn
	CMP Am,An	Am-An : EPSW	Δ	Δ	Δ	Δ	1 1011 AmAn
CMP	CMP Rm,Rn	Rm-Rn : EPSW	Δ	Δ	Δ	Δ	3 1111 1001 1101 1000 Rm.. Rn..
	CMP imm8,Dn	Dn-(sign_ext)imm8 : EPSW	Δ	Δ	Δ	Δ	2 1010 DnDn imm8...
	CMP imm16,Dn	Dn-(sign_ext)imm16 : EPSW	Δ	Δ	Δ	Δ	4 1111 1010 1100 10Dn imm16...
CMP	CMP imm32,Dn	Dn-imm32 : EPSW	Δ	Δ	Δ	Δ	6 1111 1100 1100 10Dn imm32...
	CMP imm8,An	An-(zero_ext)imm8 : EPSW	Δ	Δ	Δ	Δ	2 1011 AnAn imm8....
	CMP imm16,An	An-(zero_ext)imm16 : EPSW	Δ	Δ	Δ	Δ	4 1111 1010 1101 10An imm16...
CMP	CMP imm32,An	An-imm32 : EPSW	Δ	Δ	Δ	Δ	6 1111 1100 1101 10An imm32...
	CMP imm8,Rn	Rn-(sign_ext)imm8 : EPSW	Δ	Δ	Δ	Δ	4 1111 1011 1101 1000 ---- Rn.. imm8....
	CMP imm24,Rn	Rn-(sign_ext)imm24 : EPSW	Δ	Δ	Δ	Δ	6 7 1111 1101 1101 1000 ---- Rn.. imm24...
CMP	CMP imm32,Rn	Rn-imm32 : EPSW	Δ	Δ	Δ	Δ	2 1111 1110 1101 1000 ---- Rn.. imm32....
	AND Dm,Dn	Dm&Dn->Dn	0	0	Δ	Δ	3 1111 0010 0000 DmDn
	AND Rm,Rn	Rm&Rn->Rn	0	0	Δ	Δ	4 3 1111 1001 0000 1001 Rm.. Rn..
AND	AND Rm,Rn,Rd	Rm&Rn->Rd	0	0	Δ	Δ	4 1111 1011 0000 1101 Rm.. Rn.. Rd.. ----
	AND imm8,Dn	(zero_ext)imm8&Dn->Dn	0	0	Δ	Δ	6 1111 1000 1110 00Dn imm8....
	AND imm16,Dn	(zero_ext)imm16&Dn->Dn	0	0	Δ	Δ	4 1111 1010 1110 00Dn imm16...
AND	AND imm32,Dn	imm32&Dn->Dn	0	0	Δ	Δ	6 1111 1100 1110 00Dn imm32...
	AND imm8,Rn	(zero_ext)imm8&Rn->Rn	0	0	Δ	Δ	7 1111 1011 0000 1001 Rn.. Rm.. imm8....
	AND imm24,Rn	(zero_ext)imm24&Rn->Rn	0	0	Δ	Δ	4 6 1111 1101 0000 1001 Rn.. Rm.. imm24...
AND	AND imm32,Rn	imm32&Rn->Rn	0	0	Δ	Δ	2 3 1111 1110 0000 1001 Rn.. Rm.. imm32....
	AND imm16,PSW	imm16 & EPSW[15:0]->EPSW[15:0]	Δ	Δ	Δ	Δ	4 1111 1010 1111 1100 imm16...
	AND imm32,EPSPW	imm32 & EPSW -> EPSW	Δ	Δ	Δ	Δ	1111 1100 1111 1100 imm32...
OR	OR Dm,Dn	Dm Dn->Dn	0	0	Δ	Δ	1111 0010 0001 DmDn
	OR Rm,Rn	Rm Rn->Rn	0	0	Δ	Δ	1111 1001 0001 1001 Rm.. Rn..
	OR Rm,Rn,Rd	Rm Rn->Rd	0	0	Δ	Δ	1111 1011 0001 1101 Rm.. Rn.. Rd.. ----

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/NF/ZF		
OR	OR imm8,Dn	(zero_ext)imm8 Dn->Dn	0	0	Δ	3	1111 1000 1110 01Dn imm8....
	OR imm16,Dn	(zero_ext)imm16 Dn->Dn	0	0	Δ	4	1111 1010 1110 01Dn imm16...
	OR imm32,Dn	imm32 Dn->Dn	0	0	Δ	6	1111 1100 1110 01Dn imm32...
	OR imm8,Rn	(zero_ext)imm8 Rn->Rn	0	0	Δ	4	1111 1011 0001 1001 Rn.. Rn.. imm8....
	OR imm24,Rn	(zero_ext)imm24 Rn->Rn	0	0	Δ	6	1111 1101 0001 1001 Rn.. Rn.. imm24...
	OR imm32,Rn	imm32 Rn->Rn	0	0	Δ	7	1111 1110 0001 1001 Rn.. Rn.. imm32....
	OR imm16,PSW	imm16 EPSW[15:0]->EPSW[15:0]	Δ	Δ	Δ	4	1111 1010 1111 1101 imm16...
	OR imm32,EPSPW	imm32 EPSPW -> EPSPW	Δ	Δ	Δ	6	1111 1100 1111 1101 imm32....
	XOR Dm,Dn	Dm^Dn->Dn	0	0	Δ	2	1111 0010 0010 DmDn
	XOR Rm,Rn	Rm^Rn->Rn	0	0	Δ	3	1111 1001 0010 1001 Rm.. Rn..
XOR	XOR Rm,Rn,Rd	Rm^Rn->Rd	0	0	Δ	4	1111 1011 0010 1101 Rm.. Rn.. Rd.. ----
	XOR imm16,Dn	(zero_ext)imm16^Dn->Dn	0	0	Δ	4	1111 1010 1110 10Dn imm16...
	XOR imm32,Dn	imm32^Dn->Dn	0	0	Δ	6	1111 1100 1110 10Dn imm32...
	XOR imm8,Rn	(zero_ext)imm8^Rn->Rn	0	0	Δ	4	1111 1011 0010 1001 Rn.. Rn.. imm8....
	XOR imm24,Rn	(zero_ext)imm24^Rn->Rn	0	0	Δ	6	1111 1101 0010 1001 Rn.. Rn.. imm24...
	XOR imm32,Rn	imm32^Rn->Rn	0	0	Δ	7	1111 1110 0010 1001 Rn.. Rn.. imm32....
	NOT Dn	Dn ^ 0xFFFFFFFF -> Dn	0	0	Δ	2	1111 0010 0011 00Dn
	NOT Rn	Rn ^ 0xFFFFFFFF -> Rn	0	0	Δ	3	1111 1001 0011 1001 Rn.. Rn..
	BTST imm8,Dn	(zero_ext)imm8&Dn : EPSPW	0	0	Δ	3	1111 1000 1110 11Dn imm8....
	BTST imm16,Dn	(zero_ext)imm16&Dn : EPSPW	0	0	Δ	4	1111 1010 1110 11Dn imm16...
BTST	BTST imm32,Dn	imm32 & Dn : EPSPW	0	0	Δ	6	1111 1100 1110 11Dn imm32...
	BTST imm8,Rn	(zero_ext)imm8&Rn : EPSPW	0	0	Δ	4	1111 1011 1110 1001 ---- Rn.. imm8....
	BTST imm24,Rn	(zero_ext)imm24&Rn : EPSPW	0	0	Δ	6	1111 1101 1110 1001 ---- Rn.. imm24...
	BTST imm32,Rn	imm32 & Rn : EPSPW	0	0	Δ	7	1111 1110 1110 1001 ---- Rn.. imm32....
	BTST imm8,(d8,An)	mem8(An+(sign_ext)d8)->TMP[7:0], 0x000000->TMP[31:8], (sign_ext)imm8 & TMP : EPSPW	0	0	Δ	5	1111 1110 1000 0010 abs16... imm8....
	BTST imm8,(abs16)	mem8((zero_ext)abs16)->TMP[7:0], 0x000000->TMP[31:8], (sign_ext)imm8 & TMP : EPSPW	0	0	Δ	5	1111 1110 1000 0010 abs16... imm8....
	BTST imm8,(abs32)	mem8(abs32)->TMP[7:0], 0x000000->TMP[31:8], (sign_ext)imm8 & TMP : EPSPW	0	0	Δ	7	1111 1110 0000 0010 abs32... imm8....
	BSET Dm,(An)	mem8(An)->TMP[7:0], 0x000000->TMP[31:8], TMP&Dn:EPSPW ,(TMP Dn)[7:0]->mem8(An)	0	0	Δ	2	1111 0000 1000 DmAn

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	C	F	ZF	
BSET	BSET imm8,(d8,An)	mem8(An+(sign_ext)d8)->TMP[7:0], 0x000000->TMP[31:8], TMP&(sign_ext)imm8 : EPSW, (TMP[(zero_ext)imm8][7:0] ->mem8(An+(sign_ext)d8)	0	0	0	Δ	4 1111 1010 1111 00An d8..... imm8....
	BSET imm8,(abs16)	mem8((zero_ext)abs16)->TMP[7:0], 0x000000->TMP[31:8], TMP&(sign_ext)imm8 : EPSW, (TMP[(zero_ext)imm8][7:0] ->mem8((zero_ext)abs16)	0	0	0	Δ	5 1111 1110 1000 0000 abs16... imm8....
	BSET imm8,(abs32)	mem8(abs32)->TMP[7:0], 0x000000->TMP[31:8], TMP&(sign_ext)imm8:EPSW, (TMP[(zero_ext)imm8][7:0] ->mem8(abs32)	0	0	0	Δ	7 1111 1110 0000 0000 abs32... imm8....
	BCLR Dm,(An)	mem8(An)->TMP[7:0], 0x000000->TMP[31:8], TMP&Dn:EPSW, (TMP&(Dn^0xFFFFFFFF))[7:0] ->mem8(An)	0	0	0	Δ	2 1111 0000 1001 DmAn
BCLR	BCLR imm8,(d8,An)	mem8(An+(sign_ext)d8)->TMP[7:0], 0x000000->TMP[31:8], TMP&(sign_ext)imm8 : EPSW, (TMP[(zero_ext)imm8^0xFFFFFFFF][7:0] ->mem8(An+(sign_ext)d8)	0	0	0	Δ	4 1111 1010 1111 01An d8..... imm8....
	BCLR imm8,(abs16)	mem8((zero_ext)abs16)->TMP[7:0], 0x000000->TMP[31:8], TMP&(sign_ext)imm8 : EPSW, (TMP[(zero_ext)imm8^0xFFFFFFFF][7:0] ->mem8((zero_ext)abs16)	0	0	0	Δ	5 1111 1110 1000 0001 abs16... imm8....
	BCLR imm8,(abs32)	mem8(abs32)->TMP[7:0], 0x000000->TMP[31:8], TMP&(sign_ext)imm8:EPSW, (TMP[(zero_ext)imm8^0xFFFFFFFF][7:0] ->mem8(abs32)	0	0	0	Δ	7 1111 1110 0000 0001 abs32... imm8....

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code	
			VF	CF	NF	ZF		
ASR	ASR Dm,Dn	IF((Dm&0x0000001F)!=0),Dn[0]->EPSW.C,(Dn>>(Dm&0x0000001F))(sign_ext)->Dn, else shift operation is not performed	?	Δ?	Δ	Δ	2	1111 0010 1011 DmDn
	ASR Rm,Rn	IF((Rm&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(Rm&0x0000001F))(sign_ext)->Rn, else shift operation is not performed	?	Δ?	Δ	Δ	3	1111 1001 0100 1001 Rm.. Rn..
	ASR Rm,Rn,Rd	IF((Rm&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(Rm&0x0000001F))(sign_ext)->Rd, else shift operation is not performed	?	Δ?	Δ	Δ	4	1111 1011 0100 1101 Rm.. Rn.. Rd.. ----
	ASR imm8,Dn	IF((imm8&0x0000001F)!=0),Dn[0]->EPSW.C,(Dn>>(imm8&0x0000001F))(sign_ext)->Dn, else shift operation is not performed	?	Δ?	Δ	Δ	3	1111 1000 1100 10Dn imm8...
	ASR imm8,Rn	IF((imm8&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(imm8&0x0000001F))(sign_ext)->Rn, else shift operation is not performed	?	Δ?	Δ	Δ	4	1111 1011 0100 1001 Rn.. Rm.. imm8....
	ASR imm24,Rn	IF((imm24&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(imm24&0x0000001F))(sign_ext)->Rn, else shift operation is not performed	?	Δ?	Δ	Δ	6	1111 1101 0100 1001 Rn.. Rm.. imm24...
	ASR imm32,Rn	IF((imm32&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(imm32&0x0000001F))(sign_ext)->Rn, else shift operation is not performed	?	Δ?	Δ	Δ	7	1111 1110 0100 1001 Rn.. Rm.. imm32...
LSR	LSR Dm,Dn	IF((Dm&0x0000001F)!=0),Dn[0]->EPSW.C,(Dn>>(Dm&0x0000001F))(zero_ext)->Dn, else shift operation is not performed	?	Δ?	Δ	Δ	2	1111 0010 1010 DmDn
	LSR Rm,Rn	IF((Rm&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(Rm&0x0000001F))(zero_ext)->Rn, else shift operation is not performed	?	Δ?	Δ	Δ	3	1111 1001 0101 1001 Rm.. Rn..
	LSR Rm,Rn,Rd	IF((Rm&0x0000001F)!=0),Rn[0]->EPSW.C,(Rn>>(Rm&0x0000001F))(zero_ext)->Rd, else shift operation is not performed	?	Δ?	Δ	Δ	4	1111 1011 0101 1101 Rm.. Rn.. Rd.. ----

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			VF	CF	NF	ZF	
LSR	LSR imm8,Dn	IF((imm8&0x0000001F)!=0), Dn[0]->EPSW.C, (Dn>>(imm8&0x0000001F))(zero_ext)->Dn, else shift operation is not performed	?/	Δ	Δ	Δ	3 1111 1000 1100 01Dn imm8....
	LSR imm8,Rn	IF((imm8&0x0000001F)!=0), Rn[0]->EPSW.C, (Rn>>(imm8&0x0000001F))(zero_ext)->Rn, else shift operation is not performed	?	Δ	Δ	Δ	4 1111 1011 0101 1001 Rn.. Rm.. imm8....
	LSR imm24,Rn	IF((imm24&0x0000001F)!=0), Rn[0]->EPSW.C, (Rn>>(imm24&0x0000001F))(zero_ext)->Rn, else shift operation is not performed	?	Δ	Δ	Δ	6 1111 1101 0101 1001 Rn.. Rm.. imm24...
	LSR imm32,Rn	IF((imm32&0x0000001F)!=0), Rn[0]->EPSW.C, (Rn>>(imm32&0x0000001F))(zero_ext)->Rn, else shift operation is not performed	?	Δ	Δ	Δ	7 1111 1110 0101 1001 Rn.. Rm.. imm32...
	ASL Dm,Dn	IF((Dm&0x00001F)!=0), (Dn<<(Dm&0x00001F))->Dn else shift operation is not performed	?	Δ	Δ	Δ	2 1111 0010 1001 DmDn
ASL	ASL Rm,Rn	IF((Rm&0x00001F)!=0), (Rn<<(Rm&0x00001F))->Rn else shift operation is not performed	?	Δ	Δ	Δ	3 1111 1001 0110 1001 Rm.. Rn..
	ASL Rm,Rn,Rd	IF((Rm&0x00001F)!=0), (Rn<<(Rm&0x00001F))->Rd else shift operation is not performed	?	Δ	Δ	Δ	4 1111 1011 0110 1101 Rm.. Rn.. Rd.. ----
	ASL imm8,Dn	IF((imm8&0x00001F)!=0), (Dn<<(imm8&0x00001F))->Dn, imm=0 shift operation is not performed	?	Δ	Δ	Δ	3 1111 1000 1100 00Dn imm8....
	ASL imm8,Rn	IF((imm8&0x00001F)!=0), (Rn<<(imm8&0x00001F))->Rn, imm=0 shift operation is not performed	?	Δ	Δ	Δ	4 1111 1011 0110 1001 Rn.. Rm.. imm8....
	ASL imm24,Rn	IF((imm24&0x00001F)!=0), (Rn<<(imm24&0x00001F))->Rn, imm=0 shift operation is not performed	?	Δ	Δ	Δ	6 1111 1101 0110 1001 Rn.. Rm.. imm24...
ASL2	ASL imm32,Rn	IF((imm32&0x00001F)!=0), (Rn<<(imm32&0x00001F))->Rn, imm=0 shift operation is not performed	?	Δ	Δ	Δ	7 1111 1110 0110 1001 Rn.. Rm.. imm32...
	ASL2 Dn	Dn<<2->Dn	?	Δ	Δ	Δ	1 0101 01Dn

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C		
ASL2	ASL2 Rn	Rn<<2->Rn	?	?	Δ	3	1111 1001 0111 1001 Rn.. Rn..
ROR	ROR Dn	{EPSW.C,Dn[31:0]}->{Dn[31:0],EPSW.C}	0	Δ	Δ	2	1111 0010 1000 01Dn
	ROR Rn	{EPSW.C,Rn[31:0]}->{Rn[31:0],EPSW.C}	0	Δ	Δ	3	1111 0010 1000 1001 Rn.. Rn..
ROL	ROL Dn	{Dn[31:0],EPSW.C}>{EPSW.C,Dn[31:0]}	0	Δ	Δ	2	1111 0010 1000 00Dn
	ROL Rn	{Rn[31:0],EPSW.C}>{EPSW.C,Rn[31:0]}	0	Δ	Δ	3	1111 1001 1001 1001 Rn.. Rn..
Bcc	BEQ label	IF(Z=1orZFlagSet), PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 1000 d8.....
	BNE label	IF(Z!=1orZFlagCleared), PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 1001 d8.....
BGT	BGT label	IF(~(Z (N^V))), PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0001 d8.....
	BGE label	IF(~(N^V)),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0010 d8.....
BLE	BLE label	IF(Z (N^V)),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0011 d8.....
	BLT label	IF(N^V),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0000 d8.....
BHI	BHI label	IF(~(C Z)),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0101 d8.....
	BCC label	IF(~C),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0110 d8.....
BLS	BLS label	IF(C Z),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0111 d8.....
	BCS label	IF(C),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	2	1100 0100 d8.....
BVC	BVC label	IF(~V),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	3	1111 1000 1110 1000 d8.....
	BVS label	IF(V),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	3	1111 1000 1110 1001 d8.....
BNC	BNC label	IF(~N),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	3	1111 1000 1110 1010 d8.....
	BNS label	IF(N),PC+((sign_ext)d8)->nPC, else PC+CodeSize->nPC	-	-	-	3	1111 1000 1110 1011 d8.....
BRA	BRA label	PC+((sign_ext)d8)->nPC	-	-	-	2	1100 1010 d8.....
			-	-	-	2	1100 1010 d8.....

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Machine Code
			V	F	C	N		
Lcc	LEQ	IF(Z),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 1000	
	LNE	IF(~Z),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 1001	
	LGT	IF(~(Z (N^V))),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0001	
	LGE	IF(~(N^V)),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0010	
	LLE	IF(Z (N^V)),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0011	
	LLT	IF(N^V),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0000	
	LHI	IF(~(C Z)),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0101	
	LCC	IF(~C),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0110	
	LLS	IF(C Z),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0111	
	LCS	IF(C),LAR-4->nPC,else PC+1->nPC	-	-	-	-	1 1101 0100	
	LRA	LAR-4->nPC	-	-	-	-	1 1101 1010	
	SETLB	set loop buffer	-	-	-	-	1 1101 1011	
	JMP	An->nPC	-	-	-	-	2 1111 0000 1111 01An	
	JMP	label(d16,PC)	-	-	-	-	3 1100 1100 d16.....	
CALL	JMP	label(d32,PC)	-	-	-	-	5 1101 1100 d32.....	
	CALL	PC+(sign_ext)d16->nPC,PC+CodeSize->(SP),PC+CodeSize->MDR, Multiple address specified by regs-> Lower address memory following(SP-1), SP-(zero_ext)imm8->SP	-	-	-	-	5 1100 1101 d16..... regs.... imm8....	
	CALL	PC+d32->nPC,PC+CodeSize->(SP),PC+CodeSize->MDR, Multiple address specified by regs-> Lower address memory following(SP-1), SP-(zero_ext)imm8->SP	-	-	-	-	7 1101 1101 d32..... regs.... imm8....	
	CALLS	An->nPC,PC+CodeSize->mem32(SP),PC+CodeSize->MDR	-	-	-	-	2 1111 0000 1111 00An	
	CALLS	PC+(sign_ext)d16->nPC,PC+CodeSize->(SP),PC+CodeSize->MDR	-	-	-	-	4 1111 1010 1111 1111 d16.....	
	CALLS	PC+d32->nPC,PC+CodeSize->(SP),PC+CodeSize->MDR	-	-	-	-	6 1111 1100 1111 1111 d32.....	
	CALLS	(An)	-	-	-	-		
	CALLS	(d16,PC)	-	-	-	-		
	CALLS	(d32,PC)	-	-	-	-		
	CALLS		-	-	-	-		

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code	
			V	F	NF	ZF		
RET	RET	(SP+(zero_ext)imm8)->nPC, Lower address memory following(SP-1)-> Multiple registers specified by regs, SP+(zero_ext)imm8->SP	-	-	-	-	3	1101 1111 regs.... imm8....
RETF	RETF	MDR->nPC(next instruction PC), Lower address memory following(SP-1)-> Multiple registers specified by regs, SP+(zero_ext)imm8->SP	-	-	-	-	3	1101 1110 regs.... imm8....
RETS	RETS	mem32(SP)->nPC	-	-	-	-	2	1111 0000 1111 1100
RTI	RTI	mem32(SP)->EPSW,mem32(SP+4)->nPC, SP+8->SP	-	-	-	-	2	1111 0000 1111 1101
TRAP	TRAP	{TBR[31:24],0x000010}->nPC, PC+CodeSize->mem32(SP)	-	-	-	-	2	1111 0000 1111 1110
NOP	NOP	PC+CodeSize->nPC	-	-	-	-	1	1100 1011
SYSCALL	SYSCALL imm4	PC->mem32(SP),EPSW->mem32(SP-4), SP-8->SP,	-	-	-	-	2	1111 0000 1110 imm4
PI	PI	{TBR[31:24],(0x00300 imm4x8)}->nPC PC->mem32(SP-4),EPSW->mem32(SP-8), SP-8->SP,{TBR[31:24],0x000008}->nPC	-	-	-	-	1	1111 1111
DMULH	DMULH Rm,Rn	(sign_ext)Rm[31:16]*((sign_ext)Rn[31:16]) ->MDRQ, ((sign_ext)Rm[15:0]*((sign_ext)Rn[15:0]) ->Rn	-	-	-	-	3	1111 1001 1000 1011 Rm.. Rn..
		(sign_ext)Rm[31:16]*((sign_ext)Rn[31:16]) ->Rd1, ((sign_ext)Rm[15:0]*((sign_ext)Rn[15:0]) ->Rn2	-	-	-	-	4	1111 1011 1000 1111 Rm.. Rn.. Rd1. Rd2.
DMULH	imm32,Rn	(sign_ext)imm32[31:16]*((sign_ext)Rn[31:16]) ->MDRQ, ((sign_ext)imm32[15:0]*((sign_ext)Rn[15:0]) ->Rn	-	-	-	-	7	1111 1110 1000 1011 Rn.. Rm.. imm32...
		((zero_ext)Rm[31:16]*((zero_ext)Rn[31:16]) ->MDRQ, ((zero_ext)Rm[15:0]*((zero_ext)Rn[15:0]) ->Rn	-	-	-	-	3	1111 1001 1001 1011 Rm.. Rn..

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code	Machine Code
			V	F	CF	ZF	
DMULHU	DMULHU Rm,Rn,Rd1,Rd2	((zero_ext)Rm[31:16])*((zero_ext)Rn[31:16]) ->Rd1, ((zero_ext)Rm[15:0])*((zero_ext)Rn[15:0]) ->Rd2	-	-	-	4	1111 1011 1001 1111 Rm.. Rn.. Rd1. Rd2..
	DMULHU imm32,Rn	((zero_ext)imm32[31:16])*((zero_ext)Rn[31:16]) ->MDRQ, ((zero_ext)imm32[15:0])*((zero_ext)Rn[15:0]) ->Rn	-	-	-	7	1111 1110 1001 1011 Rn.. Rm.. imm32....
	DMACH Rm,Rn	((sign_ext)Rm[31:16])*(sign_ext)Rn[31:16]) +((sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]) +MCRL->MCRL	-	-	-	3	1111 1001 0110 1011 Rm.. Rn..
	DMACH Rm,Rn,Rd	((sign_ext)Rm[31:16])*(sign_ext)Rn[31:16]) +((sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]) +Rd->Rd	Δ	-	-	4	1111 1011 0110 1111 Rm.. Rn.. Rd.. ----
DMACHU	DMACHU imm32,Rn	((sign_ext)imm32[31:16])*(sign_ext)Rn[31:16]) +((sign_ext)imm32[15:0]*(sign_ext)Rn[15:0]) +MCRL->MCRL	-	-	-	7	1111 1110 0110 1011 ---- Rn.. imm32....
	DMACHU Rm,Rn	((zero_ext)Rm[31:16])*(zero_ext)Rn[31:16]) +((zero_ext)Rm[15:0]*(zero_ext)Rn[15:0]) +MCRL->MCRL	-	-	-	3	1111 1001 0111 1011 Rm.. Rn..
	DMACHU Rm,Rn,Rd	((zero_ext)Rm[31:16])*(zero_ext)Rn[31:16]) +((zero_ext)Rm[15:0]*(zero_ext)Rn[15:0]) +Rd->Rd	Δ	-	-	4	1111 1011 0111 1111 Rm.. Rn.. Rd.. ----
	DMACHU imm32,Rn	((zero_ext)imm32[31:16])*(zero_ext)Rn[31:16]) +((zero_ext)imm32[15:0]*(zero_ext)Rn[15:0]) +MCRL->MCRL	-	-	-	7	1111 1110 0111 1011 ---- Rn.. imm32....
MAC	MAC Rm,Rn	(Rm*Rn)+{MCRH,MCRL}->{MCRH,MCRL}	-	-	-	3	1111 1001 0000 1011 Rm.. Rn..
	MAC Rm,Rn,Rd1,Rd2	(Rm*Rn)+{Rd1,Rd2}->{Rd1,Rd2}	-	-	-	4	1111 1011 0000 1111 Rm.. Rn.Rd1..Rd2.....
	MAC imm8,Rn	((sign_ext)imm8*Rn)+{MCRH,MCRL}->{MCRH,MCRL}	-	-	-	4	1111 1011 0000 1011 Rn..imm8....

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code
			V	F	C/NF/ZF		
MAC	MAC imm24, Rn	((sign_ext)imm24*Rn)+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	6	1111 1101 0000 1011 ---- Rn.. imm24...
	MAC imm32, Rn	((imm32*Rn)+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	7	1111 1110 0000 1011 ---- Rn.. imm32...
	MACU Rm, Rn	(Rm*Rn)+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	3	1111 1001 0001 1011 Rm.. Rn..
	MACU Rm, Rn, Rd1, Rd2	(Rm*Rn)+{Rd1, Rd2}->{Rd1, Rd2}	Δ	-	-	4	1111 1011 0001 1111 Rm.. Rn.. Rd1. Rd2.
	MACU imm8, Rn	((zero_ext)imm8*Rn)+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	4	1111 1011 0001 1011 ---- Rn.. imm8...
MACH	MACU imm24, Rn	((zero_ext)imm24*Rn)+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	6	1111 1101 0001 1011 ---- Rn.. imm24...
	MACU imm32, Rn	((imm32*Rn)+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	7	1111 1110 0001 1011 ---- Rn.. imm32...
	MACH Rm, Rn	((sign_ext)Rm[15:0])*((sign_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	3	1111 1001 0100 1011 Rm.. Rn..
	MACH Rm, Rn, Rd1, Rd2	((sign_ext)Rm[15:0])*((sign_ext)Rn[15:0]))+{Rd1, Rd2}->{Rd1, Rd2}	Δ	-	-	4	1111 1011 0100 1111 Rm.. Rn.. Rd1. Rd2.
	MACH imm8, Rn	((sign_ext)imm8[15:0])*((sign_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	4	1111 1011 0100 1011 ---- Rn.. imm8...
MACHU	MACHU imm24, Rn	((sign_ext)imm24[15:0])*((sign_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	6	1111 1101 0100 1011 ---- Rn.. imm24...
	MACHU imm32, Rn	((imm32[15:0])*((sign_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	7	1111 1110 0100 1011 ---- Rn.. imm32...
	MACHU Rm, Rn	((zero_ext)Rm[15:0])*((zero_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	3	1111 1001 0101 1011 Rm.. Rn..
	MACHU Rm, Rn, Rd1, Rd2	((zero_ext)Rm[15:0])*((zero_ext)Rn[15:0]))+{Rd1, Rd2}->{Rd1, Rd2}	Δ	-	-	4	1111 1011 0101 1111 Rm.. Rn.. Rd1. Rd2.
	MACHU imm8, Rn	((zero_ext)imm8[15:0])*((zero_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	4	1111 1011 0101 1011 ---- Rn.. imm8...
MACB	MACHU imm24, Rn	((zero_ext)imm24[15:0])*((zero_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	6	1111 1101 0101 1011 ---- Rn.. imm24...
	MACHU imm32, Rn	((imm32[15:0])*((zero_ext)Rn[15:0]))+{MCRH, MCRL}->{MCRH, MCRL}	-	-	-	7	1111 1110 0101 1011 ---- Rn.. imm32...
	MACB Rm, Rn	((sign_ext)Rm[7:0])*((sign_ext)Rn[7:0]))+MCRH->MCRH	-	-	-	3	1111 1001 0010 1011 Rm.. Rn..
	MACB Rm, Rn, Rd	((sign_ext)Rm[7:0])*((sign_ext)Rn[7:0]))+Rd->Rd	Δ	-	-	4	1111 1011 0010 1111 Rm.. Rn.. Rd.. ----

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Machine Code
			V	F	C	NF		
MACB	MACB imm8,Rn	((sign_ext)imm8[7:0])*((sign_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	4	1111 1011 0010 1011 ---- Rn.. imm8....
	MACB imm24,Rn	((sign_ext)imm24[7:0])*((sign_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	6	1111 1101 0010 1011 ---- Rn.. imm24...
	MACB imm32,Rn	((sign_ext)imm32[7:0])*((sign_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	7	1111 1110 0010 1011 ---- Rn.. imm32...
	MACBU Rm,Rn	((zero_ext)Rm[7:0])*((zero_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	3	1111 1001 0011 1011 Rm.. Rn..
MACBU	MACBU Rm,Rn,Rd	((zero_ext)Rm[7:0])*((zero_ext)Rn[7:0])) +Rd->Rd	Δ	-	-	-	4	1111 1011 0011 1111 Rm.. Rn.. Rd.. ----
	MACBU imm8,Rn	((zero_ext)imm8[7:0])*((zero_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	4	1111 1011 0011 1011 ---- Rn.. imm8....
	MACBU imm24,Rn	((zero_ext)imm24[7:0])*((zero_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	6	1111 1101 0011 1011 ---- Rn.. imm24...
	MACBU imm32,Rn	((zero_ext)imm32[7:0])*((zero_ext)Rn[7:0])) +MCRL->MCRL	-	-	-	-	7	1111 1110 0011 1011 ---- Rn.. imm32...
SWHW	SWHW Rm,Rn	Rm[31:16]->Rn[15:0],Rm[15:0]->Rn[31:16]	-	-	-	-	3	1111 1001 1110 1011 Rm.. Rn..
SWAP	SWAP Rm,Rn	Rm[31:24]->Rn[7:0],Rm[23:16]->Rn[15:8], Rm[15:8]->Rn[23:16],Rm[7:0]->Rn[31:24]	-	-	-	-	3	1111 1001 1100 1011 Rm.. Rn..
SWAPH	SWAPH Rm,Rn	Rm[31:24]->Rn[23:16],Rm[23:16]->Rn[31:24], Rm[15:8]->Rn[7:0],Rm[7:0]->Rn[15:8]	-	-	-	-	3	1111 1001 1101 1011 Rm.. Rn..
SAT16	SAT16Rm,Rn	if(Rm>=0x7FFF)then 0x7FFF->Rn, else if(Rm<=0xFFFF8000)then 0xFFFF8000->Rn,else Rm,->Rn	?	Δ	Δ		3	1111 1001 1010 1011 Rm.. Rn..
SAT24	SAT24Rm,Rn	if(Rm>=0x7FFFFFFF)then 0x7FFFFFFF->Rn, else if(Rm<=0xFF800000)then 0xFF800000->Rn,else Rm,->Rn	?	Δ	Δ		4	1111 1011 1010 1111 Rm.. Rn.. ----
MCSTE	MCSTE Rm,Rn	See "MCSTE" page	0/1	0	?	?	3	1111 1001 1011 1011 Rm.. Rn..
	MCSTE imm8,Rn	See "MCSTE" page	0/1	0	?	?	4	1111 1011 1011 1011 ---- Rn.. imm8....
BSCH	BSCH Rm,Rn	See "BSCH" page	?	0/1	?	?	3	1111 1001 1111 1011 Rm.. Rn..
	BSCH Rm,Rn,Rd	See "BSCH" page	?	0/1	?	?	4	1111 1011 1111 1111 Rm.. Rd.. ----

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag		Code Size	Machine Code
			VF	CF		
LIW	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rm2+Rn2->Rn2	-	-	4	1111 0111 0000 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with (sign_ext)imm4+Rn2->Rn2	-	-	4	1111 0111 0001 0000 Rm1. Rn1. imm4 Rn2.
	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rn2-Rm2->Rn2	-	-	4	1111 0111 0010 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with Rn2-(sign_ext)imm4->Rn2	-	-	4	1111 0111 0011 0000 Rm1. Rn1. imm4 Rn2.
	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rn2-Rm2:EPSW	Δ	Δ	Δ	1111 0111 0100 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with Rn2-(sign_ext)imm4:EPSW	Δ	Δ	Δ	1111 0111 0101 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rm2->Rn2	-	-	4	1111 0111 0110 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with (sign_ext)imm4->Rn2	-	-	4	1111 0111 0111 0000 Rm1. Rn1. imm4 Rn2.
	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1000 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1001 0000 Rm1. Rn1. imm4 Rn2.
	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1010 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1011 0000 Rm1. Rn1. imm4 Rn2.
	ADD_ADD Rm1,Rn1,Rm2,Rn2	[Rm1+Rn1->Rn1]with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 0000 Rm1. Rn1. Rm2. Rn2.
	ADD_ADD Rm1,Rn1,imm4,Rn2	[Rm1+Rn1->Rn1]with Rn2<<(zero_ext)imm4->Rn2	-	-	4	1111 0111 1101 0000 Rm1. Rn1. imm4 Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rm2+Rn2->Rn2	-	-	4	1111 0111 0000 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with (sign_ext)imm4 ² +Rn2->Rn2	-	-	4	1111 0111 0001 0100 imm4 Rn1. imm4 Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rn2-Rm2->Rn2	-	-	4	1111 0111 0010 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with Rn2-(sign_ext)imm4 ² ->Rn2	-	-	4	1111 0111 0011 0100 imm4 Rn1. imm4 Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rn2-Rm2:EPSW	Δ	Δ	Δ	1111 0111 0100 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with Rn2-(sign_ext)imm4 ² :EPSW	Δ	Δ	Δ	1111 0111 0101 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rm2->Rn2	-	-	4	1111 0111 0110 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with (sign_ext)imm4 ² ->Rn2	-	-	4	1111 0111 0111 0100 imm4 Rn1. imm4 Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1000 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with Rn2>>(zero_ext)imm4 ² ->Rn2	-	-	4	1111 0111 1001 0100 imm4 Rn1. imm4 Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1010 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with Rn2>>(zero_ext)imm4 ² ->Rn2	-	-	4	1111 0111 1011 0100 imm4 Rn1. imm4 Rn2.
	ADD_ADD imm4,Rn1,Rm2,Rn2	[(sign_ext)imm4+Rn1->Rn1]with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 0100 imm4 Rn1. Rm2. Rn2.
	ADD_ADD imm4,Rn1,imm4,Rn2	[(sign_ext)imm4 ¹ +Rn1->Rn1]with Rn2<<(zero_ext)imm4 ² ->Rn2	-	-	4	1111 0111 1101 0100 imm4 Rn1. imm4 Rn2.
	CMP_ADD Rm1,Rn1,Rm2,Rn2	[Rn1-Rm1:EPSW]with Rm2+Rn2->Rn2	-	-	4	1111 0111 0000 0001 Rm1. Rn1. Rm2. Rn2.
	CMP_ADD Rm1,Rn1,imm4,Rn2	[Rn1-Rm1:EPSW]with (sign_ext)imm4+Rn2->Rn2	-	-	4	1111 0111 0001 0001 Rm1. Rn1. imm4 Rn2.
	CMP_ADD Rm1,Rn1,Rm2,Rn2	[Rn1-Rm1:EPSW]with Rn2-Rm2->Rn2	-	-	4	1111 0111 0010 0001 Rm1. Rn1. Rm2. Rn2.
	CMP_ADD Rm1,Rn1,imm4,Rn2	[Rn1-Rm1:EPSW]with Rn2-(sign_ext)imm4->Rn2	-	-	4	1111 0111 0011 0001 Rm1. Rn1. imm4 Rn2.
	CMP_ADD Rm1,Rn1,Rm2,Rn2	[Rn1-Rm1:EPSW]with Rm2->Rn2	-	-	4	1111 0111 0110 0001 Rm1. Rn1. Rm2. Rn2.
	CMP_ADD Rm1,Rn1,imm4,Rn2	[Rn1-Rm1:EPSW]with (sign_ext)imm4->Rn2	-	-	4	1111 0111 0111 0001 Rm1. Rn1. imm4 Rn2.
	CMP_ADD Rm1,Rn1,Rm2,Rn2	[Rn1-Rm1:EPSW]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1000 0001 Rm1. Rn1. Rm2. Rn2.

imm4*1:1st operand
imm4*2:3rd operand

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag		Code Size	Machine Code
			V	F		
LIW	CMP_AS _R Rm1, Rn1, imm4, Rn2	[Rn1-Rm1:EP _{SW}]with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1001 0001 Rm1. Rn1. imm4 Rn2.
	CMP_L _{SR} Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1:EP _{SW}]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1010 0001 Rm1. Rn1. Rm2. Rn2.
	CMP_L _{SR} Rm1, Rn1, imm4, Rn2	[Rn1-Rm1:EP _{SW}]with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1011 0001 Rm1. Rn1. imm4 Rn2.
	CMP_AS _L Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1:EP _{SW}]with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 0001 Rm1. Rn1. Rm2. Rn2.
	CMP_AS _L Rm1, Rn1, imm4, Rn2	[Rn1-Rm1:EP _{SW}]with Rn2<<(zero_ext)imm4->Rn2	-	-	4	1111 0111 1101 0001 Rm1. Rn1. imm4 Rn2.
	CMP_ADD imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4:EP _{SW}]with Rm2+Rn2->Rn2	Δ	Δ	Δ	1111 0111 0000 0101 imm4 Rn1. Rm2. Rn2.
	CMP_ADD imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ :EP _{SW}]with (sign_ext)imm4 ² +Rn2->Rn2	Δ	Δ	Δ	1111 0111 0001 0101 imm4 Rn1. imm4 Rn2.
	CMP_SUB imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4:EP _{SW}]with Rn2-Rm2->Rn2	Δ	Δ	Δ	1111 0111 0010 0101 imm4 Rn1. Rm2. Rn2.
	CMP_SUB imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ :EP _{SW}]with Rn2-(sign_ext)imm4 ² ->Rn2	Δ	Δ	Δ	1111 0111 0011 0101 imm4 Rn1. imm4 Rn2.
	CMP_MOV imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4:EP _{SW}]with Rm2->Rn2	Δ	Δ	Δ	1111 0111 0110 0101 imm4 Rn1. Rm2. Rn2.
	CMP_MOV imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ :EP _{SW}]with (sign_ext)imm4 ² ->Rn2	Δ	Δ	Δ	1111 0111 0111 0101 imm4 Rn1. imm4 Rn2.
	CMP_AS _R imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4:EP _{SW}]with Rn2>>Rm2->Rn2	Δ	Δ	Δ	1111 0111 1000 0101 imm4 Rn1. Rm2. Rn2.
	CMP_AS _R imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ :EP _{SW}]with Rn2>>(sign_ext)imm4 ² ->Rn2	Δ	Δ	Δ	1111 0111 1001 0101 imm4 Rn1. imm4 Rn2.
	CMP_L _{SR} imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4:EP _{SW}]with Rn2>>Rm2->Rn2	Δ	Δ	Δ	1111 0111 1010 0101 imm4 Rn1. Rm2. Rn2.
	CMP_L _{SR} imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ :EP _{SW}]with Rn2>>(zero_ext)imm4 ² ->Rn2	Δ	Δ	Δ	1111 0111 1011 0101 imm4 Rn1. imm4 Rn2.
	CMP_AS _L imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4:EP _{SW}]with Rn2<<Rm2->Rn2	Δ	Δ	Δ	1111 0111 1100 0101 imm4 Rn1. Rm2. Rn2.
	CMP_AS _L imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ :EP _{SW}]with Rn2<<(zero_ext)imm4 ² ->Rn2	Δ	Δ	Δ	1111 0111 1101 0101 imm4 Rn1. imm4 Rn2.
	SUB_ADD Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rm2+Rn2->Rn2	-	-	4	1111 0111 0000 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_ADD Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with (sign_ext)imm4+Rn2->Rn2	-	-	4	1111 0111 0001 0010 Rm1. Rn1. imm4 Rn2.
	SUB_SUB Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rn2-Rm2->Rn2	-	-	4	1111 0111 0010 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_SUB Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with Rn2-(sign_ext)imm4->Rn2	-	-	4	1111 0111 0011 0010 Rm1. Rn1. imm4 Rn2.
	SUB_C _{MP} Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rn2-Rm2:EP _{SW}	Δ	Δ	Δ	1111 0111 0100 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_C _{MP} Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with Rn2-(sign_ext)imm4:EP _{SW}	Δ	Δ	Δ	1111 0111 0101 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_MOV Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rm2->Rn2	-	-	4	1111 0111 0110 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_MOV Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with (sign_ext)imm4->Rn2	-	-	4	1111 0111 0111 0010 Rm1. Rn1. imm4 Rn2.
	SUB_AS _R Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1000 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_AS _R Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1001 0010 Rm1. Rn1. imm4 Rn2.
	SUB_L _{SR} Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1010 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_L _{SR} Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1011 0010 Rm1. Rn1. imm4 Rn2.
	SUB_AS _L Rm1, Rn1, Rm2, Rn2	[Rn1-Rm1->Rn1]with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 0010 Rm1. Rn1. Rm2. Rn2.
	SUB_AS _L Rm1, Rn1, imm4, Rn2	[Rn1-Rm1->Rn1]with Rn2<<(zero_ext)imm4->Rn2	-	-	4	1111 0111 1101 0010 Rm1. Rn1. imm4 Rn2.
	SUB_ADD imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rm2+Rn2->Rn2	-	-	4	1111 0111 0000 0110 imm4 Rn1. Rm2. Rn2.
	SUB_ADD imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ ->Rn1]with (sign_ext)imm4 ² +Rn2->Rn2	-	-	4	1111 0111 0001 0110 imm4 Rn1. imm4 Rn2.
	SUB_SUB imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rn2-Rm2->Rn2	-	-	4	1111 0111 0010 0110 imm4 Rn1. Rm2. Rn2.
	SUB_SUB imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4 ¹ ->Rn1]with Rn2-(sign_ext)imm4 ² ->Rn2	-	-	4	1111 0111 0011 0110 imm4 Rn1. imm4 Rn2.
	SUB_C _{MP} imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rn2-Rm2:EP _{SW}	Δ	Δ	Δ	1111 0111 0100 0110 imm4 Rn1. Rm2. Rn2.

imm4¹:1st operand
imm4²:3rd operand

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag		Code Size	Machine Code	
			V	FC			
LIW	SUB_CMP imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4*1->Rn1]with Rn2-(sign_ext)imm4*2:EPSW	Δ	Δ	Δ	4	1111 0111 0101 0110 imm4 Rn1. imm4 Rn2.
	SUB_MOV imm4, Rn1, Rn2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rm2->Rn2	-	-	-	4	1111 0111 0110 0110 imm4 Rn1. Rm2. Rn2.
	SUB_MOV imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4*1->Rn1]with (sign_ext)imm4*2->Rn2	-	-	-	4	1111 0111 0111 0110 imm4 Rn1. imm4 Rn2.
	SUB_ASR imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rn2>>Rm2->Rn2	-	-	-	4	1111 0111 1000 0110 imm4 Rn1. Rm2. Rn2.
	SUB_ASR imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4*1->Rn1]with Rn2>>(zero_ext)imm4*2->Rn2	-	-	-	44	1111 0111 1001 0110 imm4 Rn1. imm4 Rn2.
	SUB_LSR imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rn2>>Rm2->Rn2	-	-	-	4	1111 0111 1010 0110 imm4 Rn1. Rm2. Rn2.
	SUB_LSR imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4*1->Rn1]with Rn2>>(zero_ext)imm4*2->Rn2	-	-	-	4	1111 0111 1011 0110 imm4 Rn1. imm4 Rn2.
	SUB_ASL imm4, Rn1, Rm2, Rn2	[Rn1-(sign_ext)imm4->Rn1]with Rn2<<Rm2->Rn2	-	-	-	4	1111 0111 1100 0110 imm4 Rn1. Rm2. Rn2.
	SUB_ASL imm4, Rn1, imm4, Rn2	[Rn1-(sign_ext)imm4*1->Rn1]with Rn2<<(zero_ext)imm4*2->Rn2	-	-	-	4	1111 0111 1101 0110 imm4 Rn1. imm4 Rn2.
	MOV_ADD Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rm2+Rn2->Rn2	-	-	-	4	1111 0111 0000 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_ADD Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with (sign_ext)imm4+Rn2->Rn2	-	-	-	4	1111 0111 0001 0011 Rm1. Rn1. imm4 Rn2.
	MOV_SUB Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rn2-Rm2->Rn2	-	-	-	4	1111 0111 0010 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_SUB Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with Rn2-(sign_ext)imm4->Rn2	-	-	-	4	1111 0111 0011 0011 Rm1. Rn1. imm4 Rn2.
	MOV_CMP Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rn2-Rm2:EPSW	Δ	Δ	Δ	4	1111 0111 0100 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_CMP Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with Rn2-(sign_ext)imm4:EPSW	Δ	Δ	Δ	4	1111 0111 0101 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_MOV Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rm2->Rn2	-	-	-	4	1111 0111 0110 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_MOV Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with (sign_ext)imm4->Rn2	-	-	-	4	1111 0111 0111 0011 Rm1. Rn1. imm4 Rn2.
	MOV_ASR Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	4	1111 0111 1000 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_ASR Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	4	1111 0111 1001 0011 Rm1. Rn1. imm4 Rn2.
	MOV_LSR Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	4	1111 0111 1010 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_LSR Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	4	1111 0111 1011 0011 Rm1. Rn1. imm4 Rn2.
	MOV_ASL Rm1, Rn1, Rm2, Rn2	[Rm1->Rn1]with Rn2<<Rm2->Rn2	-	-	-	4	1111 0111 1100 0011 Rm1. Rn1. Rm2. Rn2.
	MOV_ASL Rm1, Rn1, imm4, Rn2	[Rm1->Rn1]with Rn2<<(zero_ext)imm4->Rn2	-	-	-	4	1111 0111 1101 0011 Rm1. Rn1. imm4 Rn2.
	MOV_ADD imm4, Rn1, Rm2, Rn2	(sign_ext)imm4->Rn1]with Rm2+Rn2->Rn2	-	-	-	4	1111 0111 0000 0111 imm4 Rn1. Rm2. Rn2.
	MOV_ADD imm4, Rn1, imm4, Rn2	(sign_ext)imm4*1->Rn1]with (sign_ext)imm4*2+Rn2->Rn2	-	-	-	4	1111 0111 0001 0111 imm4 Rn1. imm4 Rn2.
	MOV_SUB imm4, Rn1, Rm2, Rn2	(sign_ext)imm4->Rn1]with Rn2-Rm2->Rn2	-	-	-	4	1111 0111 0010 0111 imm4 Rn1. Rm2. Rn2.
	MOV_SUB imm4, Rn1, imm4, Rn2	(sign_ext)imm4*1->Rn1]with Rn2-(sign_ext)imm4*2->Rn2	-	-	-	4	1111 0111 0011 0111 imm4 Rn1. imm4 Rn2.
	MOV_CMP imm4, Rn1, Rm2, Rn2	(sign_ext)imm4->Rn1]with Rn2-Rm2	Δ	Δ	Δ	4	1111 0111 0100 0111 imm4 Rn1. Rm2. Rn2.
	MOV_CMP imm4, Rn1, imm4, Rn2	(sign_ext)imm4*1->Rn1]with Rn2-(sign_ext)imm4*2	Δ	Δ	Δ	4	1111 0111 0101 0111 imm4 Rn1. imm4 Rn2.
	MOV_MOV imm4, Rn1, Rm2, Rn2	(sign_ext)imm4->Rn1]with Rm2->Rn2	-	-	-	4	1111 0111 0110 0111 imm4 Rn1. Rm2. Rn2.
	MOV_ASR imm4, Rn1, Rm2, Rn2	(sign_ext)imm4*1->Rn1]with (sign_ext)imm4*2->Rn2	-	-	-	4	1111 0111 0111 0111 imm4 Rn1. imm4 Rn2.
	MOV_ASR imm4, Rn1, imm4, Rn2	(sign_ext)imm4->Rn1]with Rn2>>Rm2->Rn2	-	-	-	4	1111 0111 1000 0111 imm4 Rn1. Rm2. Rn2.
	MOV_LSR imm4, Rn1, Rm2, Rn2	(sign_ext)imm4->Rn1]with Rn2>>Rm2->Rn2	-	-	-	4	1111 0111 1010 0111 imm4 Rn1. Rm2. Rn2.
	MOV_LSR imm4, Rn1, imm4, Rn2	(sign_ext)imm4*1->Rn1]with Rn2>>(zero_ext)imm4*2->Rn2	-	-	-	4	1111 0111 1011 0111 imm4 Rn1. imm4 Rn2.
	MOV_ASL imm4, Rn1, Rm2, Rn2	(sign_ext)imm4->Rn1]with Rn2<<Rm2->Rn2	-	-	-	4	1111 0111 1100 0111 imm4 Rn1. Rm2. Rn2.
imm4*1:1st operand imm4*2:3rd operand							

imm4*1: 1st operand
imm4*2: 3rd operand

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag		Code Size	Machine Code
			V	FC		
LIW	MOV_AS_L imm4,Rn1,imm4,Rn2	[(sign_ext)imm4*1->Rn1]with Rn2<<(zero_ext)imm4*2->Rn2	-	-	-	1111 0111 1101 0111 imm4 Rn1, imm4 Rn2.
	AND_ADD_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rm2+Rn2->Rn2	-	-	-	1111 0111 0000 1000 Rm1, Rn1, Rm2, Rn2.
	AND_ADD_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with (sign_ext)imm4+Rn2->Rn2	-	-	-	1111 0111 0001 1000 Rm1, Rn1, imm4 Rn2.
	AND_SUB_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rn2-Rm2->Rn2	-	-	-	1111 0111 0010 1000 Rm1, Rn1, Rm2, Rn2.
	AND_SUB_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with Rn2-(sign_ext)imm4->Rn2	-	-	-	1111 0111 0011 1000 Rm1, Rn1, imm4 Rn2.
	AND_CMP_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rn2-Rm2:EPSW	Δ	Δ	Δ	1111 0111 0100 1000 Rm1, Rn1, Rm2, Rn2.
	AND_CMP_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with Rn2-(sign_ext)imm4:EPSW	Δ	Δ	Δ	1111 0111 0101 1000 Rm1, Rn1, Rm2, Rn2.
	AND_MOV_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rm2->Rn2	-	-	-	1111 0111 0110 1000 Rm1, Rn1, Rm2, Rn2.
	AND_MOV_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with (sign_ext)imm4->Rn2	-	-	-	1111 0111 0111 1000 Rm1, Rn1, imm4 Rn2.
	AND_AS_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	1111 0111 1000 1000 Rm1, Rn1, Rm2, Rn2.
	AND_AS_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	1111 0111 1001 1000 Rm1, Rn1, imm4 Rn2.
	AND_LSR_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	1111 0111 1010 1000 Rm1, Rn1, Rm2, Rn2.
	AND_LSR_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	1111 0111 1011 1000 Rm1, Rn1, imm4 Rn2.
	AND_AS_L_Rm1,Rn1,Rm2,Rn2	[Rm1&Rn1->Rn1]with Rn2<<Rm2->Rn2	-	-	-	1111 0111 1100 1000 Rm1, Rn1, Rm2, Rn2.
	AND_AS_L_Rm1,Rn1,imm4,Rn2	[Rm1&Rn1->Rn1]with Rn2<<(zero_ext)imm4->Rn2	-	-	-	1111 0111 1101 1000 Rm1, Rn1, imm4 Rn2.
	OR_ADD_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rm2+Rn2->Rn2	-	-	-	1111 0111 0000 1100 Rm1, Rn1, Rm2, Rn2.
	OR_ADD_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with (sign_ext)imm4+Rn2->Rn2	-	-	-	1111 0111 0001 1100 Rm1, Rn1, imm4 Rn2.
	OR_SUB_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rn2-Rm2->Rn2	-	-	-	1111 0111 0010 1100 Rm1, Rn1, Rm2, Rn2.
	OR_SUB_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with Rn2-(sign_ext)imm4->Rn2	-	-	-	1111 0111 0011 1100 Rm1, Rn1, imm4 Rn2.
	OR_CMP_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rn2-Rm2:EPSW	Δ	Δ	Δ	1111 0111 0100 1100 Rm1, Rn1, Rm2, Rn2.
	OR_CMP_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with Rn2-(sign_ext)imm4:EPSW	Δ	Δ	Δ	1111 0111 0101 1100 Rm1, Rn1, Rm2, Rn2.
	OR_MOV_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rm2->Rn2	-	-	-	1111 0111 0110 1100 Rm1, Rn1, Rm2, Rn2.
	OR_MOV_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with (sign_ext)imm4->Rn2	-	-	-	1111 0111 0111 1100 Rm1, Rn1, imm4 Rn2.
	OR_AS_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	1111 0111 1000 1100 Rm1, Rn1, Rm2, Rn2.
	OR_AS_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	1111 0111 1001 1100 Rm1, Rn1, imm4 Rn2.
	OR_LSR_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	1111 0111 1010 1100 Rm1, Rn1, Rm2, Rn2.
	OR_LSR_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	1111 0111 1011 1100 Rm1, Rn1, imm4 Rn2.
	OR_AS_L_Rm1,Rn1,Rm2,Rn2	[Rm1 Rn1->Rn1]with Rn2<<Rm2->Rn2	-	-	-	1111 0111 1100 1100 Rm1, Rn1, Rm2, Rn2.
	OR_AS_L_Rm1,Rn1,imm4,Rn2	[Rm1 Rn1->Rn1]with Rn2<<(zero_ext)imm4->Rn2	-	-	-	1111 0111 1101 1100 Rm1, Rn1, imm4 Rn2.

imm4*1:1st operand
imm4*2:3rd operand

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag				Code Size	Machine Code
			V	F	C	NFZF		
LIW	XOR MOV Rm1,Rn1,imm4,Rn2	[Rm1^Rn1->Rn1]with (sign_ext)imm4->Rn2	-	-	-	-	4	1111 0111 0111 1010 Rm1. Rn1. imm4 Rn2.
	XOR ASR Rm1,Rn1,Rm2,Rn2	[Rm1^Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	-	4	1111 0111 1000 1010 Rm1. Rn1. Rm2. Rn2.
	XOR ASR Rm1,Rn1,imm4,Rn2	[Rm1^Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	-	4	1111 0111 1001 1010 Rm1. Rn1. imm4 Rn2.
	XOR LSR Rm1,Rn1,Rm2,Rn2	[Rm1^Rn1->Rn1]with Rn2>>Rm2->Rn2	-	-	-	-	4	1111 0111 1010 1010 Rm1. Rn1. Rm2. Rn2.
	XOR LSR Rm1,Rn1,imm4,Rn2	[Rm1^Rn1->Rn1]with Rn2>>(zero_ext)imm4->Rn2	-	-	-	-	4	1111 0111 1011 1010 Rm1. Rn1. imm4 Rn2.
	XOR ASL Rm1,Rn1,Rm2,Rn2	[Rm1^Rn1->Rn1]with Rn2<<Rm2->Rn2	-	-	-	-	4	1111 0111 1100 1010 Rm1. Rn1. Rm2. Rn2.
	XOR ASL Rm1,Rn1,imm4,Rn2	[Rm1^Rn1->Rn1]with Rn2<<(zero_ext)imm4->Rn2	-	-	-	-	4	1111 0111 1101 1010 Rm1. Rn1. imm4 Rn2.
	DMACH_ADD Rm1,Rn1,Rm2,Rn2	[(sign_ext)Rm1[31:16]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rm2->Rn2	-	-	-	-	4	1111 0111 0000 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_ADD Rm1,Rn1,imm4,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with (sign_ext)imm4->Rn2	-	-	-	-	4	1111 0111 0001 1001 Rm1. Rn1. imm4 Rn2.
	DMACH_SUB Rm1,Rn1,Rm2,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2->Rm2->Rn2	-	-	-	-	4	1111 0111 0010 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_SUB Rm1,Rn1,imm4,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2->Rm2->Rn2	-	-	-	-	4	1111 0111 0011 1001 Rm1. Rn1. imm4 Rn2.
	DMACH_CMP Rm1,Rn1,Rm2,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2->Rm2->Rn2	Δ	Δ	Δ	Δ	4	1111 0111 0100 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_CMP Rm1,Rn1,imm4,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2->Rm2->Rn2	Δ	Δ	Δ	Δ	4	1111 0111 0101 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_MOV Rm1,Rn1,Rm2,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rm2->Rn2	-	-	-	-	4	1111 0111 0110 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_MOV Rm1,Rn1,imm4,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with (sign_ext)imm4->Rn2	-	-	-	-	4	1111 0111 0111 1001 Rm1. Rn1. imm4 Rn2.
	DMACH_ASR Rm1,Rn1,Rm2,Rn2	[(sign_ext)Rm1[31:16]]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2>>Rm2->Rn2	-	-	-	-	4	1111 0111 1000 1001 Rm1. Rn1. Rm2. Rn2.

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag		Code Size	Machine Code
			V	FC		
LIW	DMACH_ASR Rm1, Rn1, imm4, Rn2	[(sign_ext)Rm[31:16]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1001 1001 Rm1. Rn1. imm4 Rn2.
	DMACH_LSR Rm1, Rn1, Rm2, Rn2	[(sign_ext)Rm[31:16]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1010 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_LSR Rm1, Rn1, imm4, Rn2	[(sign_ext)Rm[31:16]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1011 1001 Rm1. Rn1. imm4 Rn2.
	DMACH_AS L Rm1, Rn1, Rm2, Rn2	[(sign_ext)Rm[31:16]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 1001 Rm1. Rn1. Rm2. Rn2.
	DMACH_AS L Rm1, Rn1, imm4, Rn2	[(sign_ext)Rm[31:16]*(sign_ext)Rn[31:16]+(sign_ext)Rm[15:0]*(sign_ext)Rn[15:0]+MCRL->MCRL] with Rn2<<(zero_ext)imm4->Rn2	-	-	4	1111 0111 1101 1001 Rm1. Rn1. imm4 Rn2.
	SHW_ADD Rm1, Rn1, Rm2, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rm2+Rn2->Rn2	-	-	4	1111 0111 0000 1011 Rm1. Rn1. Rm2. Rn2.
	SHW_ADD Rm1, Rn1, imm4, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with (sign_ext)imm4+Rn2->Rn2	-	-	4	1111 0111 0001 1011 Rm1. Rn1. imm4 Rn2.
	SHW_SUB Rm1, Rn1, Rm2, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2-Rm2->Rn2	-	-	4	1111 0111 0010 1011 Rm1. Rn1. Rm2. Rn2.
	SHW_SUB Rm1, Rn1, imm4, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2-(sign_ext)imm4->Rn2	-	-	4	1111 0111 0011 1011 Rm1. Rn1. imm4 Rn2.
	SHW_CMP Rm1, Rn1, Rm2, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2-Rm2:EPSW	Δ	Δ	4	1111 0111 0100 1011 Rm1. Rn1. Rm2. Rn2.
	SHW_CMP Rm1, Rn1, imm4, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2-(sign_ext)imm4:EPSW	Δ	Δ	4	1111 0111 0101 1011 Rm1. Rn1. Rm2. Rn2.
	SHW_MOV Rm1, Rn1, Rm2, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rm2->Rn2	-	-	4	1111 0111 0110 1011 Rm1. Rn1. Rm2. Rn2.
	SHW_MOV Rm1, Rn1, imm4, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with (sign_ext)imm4->Rn	-	-	4	1111 0111 0111 1011 Rm1. Rn1. imm4 Rn2.
	SHW_ASR Rm1, Rn1, Rm2, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2>>Rm2->Rn2	-	-	4	1111 0111 1000 1011 Rm1. Rn1. Rm2. Rn2.
	SHW_ASR Rm1, Rn1, imm4, Rn2	Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2>>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1001 1011 Rm1. Rn1. imm4 Rn2.

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag		Code Size	Machine Code
			V	CF		
LIW	SWHW_LSR Rm1, Rn1, Rm2, Rn2	[Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2->>Rm2->Rn2	-	-	4	1111 0111 1010 1011 Rm1. Rn1. Rm2. Rn2.
	SWHW_LSR Rm1, Rn1, imm4, Rn2	[Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2->>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1011 1011 Rm1. Rn1. imm4 Rn2.
	SWHW_AS_L Rm1, Rn1, Rm2, Rn2	[Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 1011 Rm1. Rn1. Rm2. Rn2.
	SWHW_AS_L Rm1, Rn1, imm4, Rn2	[Rm1[15:0]->Rn1[31:16], Rm1[31:16]->Rn1[15:0]] with Rn2<<(zero_ext)imm4->Rn2	-	-	4	1111 0111 1101 1011 Rm1. Rn1. imm4 Rn2.
	SAT16_ADD Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rm2->Rn2->Rn2	-	-	4	1111 0111 0000 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_ADD Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with (sign_ext)imm4->Rn2->Rn2	-	-	4	1111 0111 0001 1101 Rm1. Rn1. imm4 Rn2.
	SAT16_SUB Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rn2->Rn2->Rn2	-	-	4	1111 0111 0010 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_SUB Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with Rn2-(sign_ext)imm4->Rn2	-	-	4	1111 0111 0011 1101 Rm1. Rn1. imm4 Rn2.
	SAT16_CMP Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rn2-Rm2:EPSW	Δ	Δ	4	1111 0111 0100 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_CMP Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with Rn2-(sign_ext)imm4:EPSW	Δ	Δ	4	1111 0111 0101 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_MOV Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rm2->Rn2	-	-	4	1111 0111 0110 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_MOV Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with (sign_ext)imm4->Rn2	-	-	4	1111 0111 0111 1101 Rm1. Rn1. imm4 Rn2.
	SAT16_ASR Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rn2->>Rm2->Rn2	-	-	4	1111 0111 1000 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_ASR Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with Rn2->>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1001 1101 Rm1. Rn1. imm4 Rn2.
	SAT16_LSR Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rn2->>Rm2->Rn2	-	-	4	1111 0111 1010 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_LSR Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with Rn2->>(zero_ext)imm4->Rn2	-	-	4	1111 0111 1011 1101 Rm1. Rn1. imm4 Rn2.
	SAT16_AS_L Rm1, Rn1, Rm2, Rn2	[(SAT16Op.)Rm->Rn]with Rn2<<Rm2->Rn2	-	-	4	1111 0111 1100 1101 Rm1. Rn1. Rm2. Rn2.
	SAT16_AS_L Rm1, Rn1, imm4, Rn2	[(SAT16Op.)Rm->Rn]with Rn2<<(zero_ext)imm4->Rn2	-	-	4	1111 0111 1101 1101 Rm1. Rn1. imm4 Rn2.
	MOV_LEQ (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with leq operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 1000
	MOV_LNE (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lne operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 1001
	MOV_LGT (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lgt operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0001
	MOV_LGE (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lge operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0010
	MOV_LLE (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lle operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0011
	MOV_LLT (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with llt operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0000
	MOV_LHI (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lhi operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0101
	MOV_LCC (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lcc operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0110
	MOV_LLS (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lls operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0111
	MOV_LCS (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lcs operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 0100
	MOV_LRA (Rm+, imm4), Rn	mem32(Rm)->Rn, Rm+4->Rm, with lra operation	-	-	4	1111 0111 1110 0000 Rn.. Rm.. imm4 1010

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	Flag			Code Size	Machine Code	
			V	F	C/NFZF			
UDF	UDF00 Dm,Dn (MULQ Dm,Dn)	Dm*Dn->{MDRQ,Dn}	?	?	Δ	Δ	2	1111 0110 0000 DmDn
	UDF00 imm8,Dn (MULQ imm8,Dn)	(sign_ext)imm8*Dn->{MDRQ,Dn}	?	?	Δ	Δ	3	1111 1001 0000 00Dn imm8....
	UDF00 imm16,Dn (MULQ imm16,Dn)	(sign_ext)imm16*Dn->{MDRQ,Dn}	?	?	Δ	Δ	4	1111 1011 0000 00Dn imm16...
	UDF00 imm32,Dn (MULQ imm32,Dn)	imm32*Dn->{MDRQ,Dn}	?	?	Δ	Δ	6	1111 1101 0000 00Dn imm32...
	UDF01 Dm,Dn (MULQU Dm,Dn)	Dm*Dn->{MDRQ,Dn}	?	?	Δ	Δ	2	1111 0110 0001 DmDn
	UDFU01 imm8,Dn (MULQU imm8,Dn)	(zero_ext)imm8*Dn->{MDRQ,Dn}	?	?	Δ	Δ	3	1111 1001 0001 00Dn imm8....
	UDFU01 imm16,Dn (MULQU imm16,Dn)	(zero_ext)imm16*Dn->{MDRQ,Dn}	?	?	Δ	Δ	4	1111 1011 0001 00Dn imm16...
	UDFU01 imm32,Dn (MULQU imm32,Dn)	imm32*Dn->{MDRQ,Dn}	?	?	Δ	Δ	6	1111 1101 0001 01Dn imm32....
	UDF02 Dm,Dn (MCST32 Dn / MCST16 Dn / MCST8 Dn)	Satulate Dm({MCRH,MCRL})->Dn	0/1	0	?	?	2	1111 0110 0010 DmDn
	UDF03 Dm,Dn (MCST9 Dn)	Satulate9(MCRL)->Dn	0/1	0	?	?	2	1111 0110 0011 DmDn
	UDF04 Dm,Dn (SAT16 Dm,Dn)	Satulate16(Dm)->Dn	?	?	Δ	Δ	2	1111 0110 0100 DmDn
	UDF05 Dm,Dn (SAT24 Dm,Dn)	Satulate24(Dm)->Dn	?	?	Δ	Δ	2	1111 0110 0101 DmDn
	UDF06 Dm,Dn (MCST48 Dn)	Satulate48({MCRH,MCRL})>>16->Dn	0/1	0	?	?	2	1111 0110 0110 --Dn
	UDF07 Dm,Dn (BSCH Dm,Dn)	bit search	?	0/1	?	?	2	1111 0110 0111 DmDn
	UDF08 Dm,Dn (SWAP Dm,Dn)	Dm[31:24]->Dn[7:0],Dm[23:16]->Dn[15:8] Dm[15:8]->Dn[23:16],Dm[7:0]->Dn[31:24]	-	-	-	-	2	1111 0110 1000 DmDn
	UDF09 Dm,Dn (SWAPH Dm,Dn)	Dm[31:24]->Dn[23:16],Dm[23:16]->Dn[31:24] Dm[15:8]->Dn[7:0],Dm[7:0]->Dn[15:8]	-	-	-	-	2	1111 0110 1001 DmDn
	UDF12 Dm,Dn (GETCHX Dn)	MCRH->Dn,MCVF->EPSW.V	Δ	0	?	?	2	1111 0110 1100 --Dn
	UDF13 Dm,Dn (GETCLX Dn)	MCRL->Dn,MCVF->EPSW.V	Δ	0	?	?	2	1111 0110 1101 --Dn
	UDF15 Dm,Dn (GETX Dn)	MCRQ->Dn	0	0	Δ	Δ	2	1111 0110 1111 --Dn

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	EC Flag		FCC Flag		Code Size	Machine Code	Note
			VF	ZF	OF	IF	LF		
FMOV	FMOV (Rm),FSn	mem32(Rm)->FSn	-	-	-	-	-	3 1111 1001 0010 000X Rm.. Sn..	FSn={X,Sn}
	FMOV (SP),FSn	mem32(SP)->FSn	-	-	-	-	-	3 1111 1001 0010 010X ---- Sn..	FSn={X,Sn}
	FMOV (Rm,Ri),FSn	mem32(Rm+Ri)->FSn	-	-	-	-	-	4 1111 1011 0010 0111 Ri.. Rm.. Sn..-Z-	FSn={Z,Sn}
	FMOV (d8,Rm),FSn	mem32(Rm+(sign_ext)d8)->FSn	-	-	-	-	-	4 1111 1011 0010 000X Rm.. Sn.. d8.....	FSn={X,Sn}
	FMOV (d24,Rm),FSn	mem32(Rm+(sign_ext)d24)->FSn	-	-	-	-	-	6 1111 1101 0010 000X Rm.. Sn.. d24.....	FSn={X,Sn}
	FMOV (d32,Rm),FSn	mem32(Rm+d32)->FSn	-	-	-	-	-	7 1111 1110 0010 000X Rm.. Sn.. d32.....	FSn={X,Sn}
	FMOV (d8,SP),FSn	mem32(SP+(zero_ext)d8)->FSn	-	-	-	-	-	4 1111 1011 0010 010X ---- Sn.. d8.....	FSn={X,Sn}
	FMOV (d24,SP),FSn	mem32(SP+(zero_ext)d24)->FSn	-	-	-	-	-	6 1111 1101 0010 010X ---- Sn.. d24.....	FSn={X,Sn}
	FMOV (d32,SP),FSn	mem32(SP+d32)->FSn	-	-	-	-	-	7 1111 1110 0010 010X ---- Sn.. d32.....	FSn={X,Sn}
	FMOV FSm,(Rn)	FSm->mem32(Rn)	-	-	-	-	-	3 1111 1001 0011 00Y0 Sm.. Rn..	FSm={Y,Sm}
	FMOV FSm,(SP)	FSm->mem32(SP)	-	-	-	-	-	3 1111 1001 0011 01Y0 Sm.. ----	FSm={Y,Sm}
	FMOV FSm,(Rn,Ri)	FSm->mem32(Rn+Ri)	-	-	-	-	-	4 1111 1011 0011 0111 Ri.. Rn.. Sm..-Z-	FSm={Z,Sm}
	FMOV FSm,(d8,Rn)	FSm->mem32(Rn+(sign_ext)d8)	-	-	-	-	-	4 1111 1011 0011 00Y0 Sm.. Rn.. d8.....	FSm={X,Sm}
	FMOV FSm,(d24,Rn)	FSm->mem32(Rn+(sign_ext)d24)	-	-	-	-	-	6 1111 1101 0011 00Y0 Sm.. Rn.. d24.....	FSm={Y,Sm}
	FMOV FSm,(d32,Rn)	FSm->mem32(Rn+d32)	-	-	-	-	-	7 1111 1110 0011 00Y0 Sm.. Rn.. d32.....	FSm={Y,Sm}
	FMOV FSm,(d8,SP)	FSm->mem32(SP+(zero_ext)d8)	-	-	-	-	-	4 1111 1011 0011 01Y0 Sm.. ---- d8.....	FSm={X,Sm}
	FMOV FSm,(d24,SP)	FSm->mem32(SP+(zero_ext)d24)	-	-	-	-	-	6 1111 1101 0011 01Y0 Sm.. ---- d24.....	FSm={Y,Sm}
	FMOV FSm,(d32,SP)	FSm->mem32(SP+d32)	-	-	-	-	-	7 1111 1110 0011 01Y0 Sm.. ---- d32.....	FSm={Y,Sm}
	FMOV (Rm+),FSn	mem32(Rm)->FSn,Rm+0x00000004->Rm	-	-	-	-	-	3 1111 1001 0010 001X Rm.. Sn..	FSn={X,Sn}
	FMOV (Rm+,imm8),FSn	mem32(Rm)->FSn,Rm+(sign_ext)imm8->Rm	-	-	-	-	-	4 1111 1011 0010 001X Rm.. Sn.. imm8....	FSn={X,Sn}
	FMOV (Rm+,imm24),FSn	mem32(Rm)->FSn,Rm+(sign_ext)imm24->Rm	-	-	-	-	-	6 1111 1101 0010 001X Rm.. Sn.. imm24...	FSn={X,Sn}
	FMOV (Rm+,imm32),FSn	mem32(Rm)->FSn,Rm+imm32->Rm	-	-	-	-	-	7 1111 1110 0010 001X Rm.. Sn.. imm32....	FSn={X,Sn}
	FMOV FSm,(Rn+)	FSm->mem32(Rn),Rn+0x00000004->Rn	-	-	-	-	-	3 1111 1001 0011 00Y1 Sm.. Rn..	FSm={Y,Sm}
	FMOV FSm,(Rn+,imm8)	FSm->mem32(Rn),Rn+(sign_ext)imm8->Rn	-	-	-	-	-	4 1111 1011 0011 00Y1 Sm.. Rn.. imm8....	FSm={X,Sm}
	FMOV FSm,(Rn+,imm24)	FSm->mem32(Rn),Rn+(sign_ext)imm24->Rn	-	-	-	-	-	6 1111 1101 0011 00Y1 Sm.. Rn.. imm24...	FSm={Y,Sm}
	FMOV FSm,(Rn+,imm32)	FSm->mem32(Rn),Rn+imm32->Rn	-	-	-	-	-	7 1111 1110 0011 00Y1 Sm.. Rn.. imm32....	FSm={Y,Sm}
	FMOV FSm,FSn	FSm->FSn	-	-	-	-	-	3 1111 1001 0100 00YX Sm.. Sn..	FSm={Y,Sm},FSn={X,Sn}
	FMOV FSm,Rn	FSm->Rn	-	-	-	-	-	3 1111 1001 0011 01Y1 Sm.. Rn..	FSm={Y,Sm}
	FMOV Rm,FSn	Rm->FSn	-	-	-	-	-	3 1111 1001 0010 011X Rm.. Sn..	FSn={X,Sn}
	FMOV imm32,FSn	imm32->FSn	-	-	-	-	-	7 1111 1110 0010 011X ---- Sn.. imm32...	FSn={X,Sn}
	FMOV FPCR,Rn	FPCR->Rn	-	-	-	-	-	3 1111 1001 1011 0111 ---- Rn..	
	FMOV Rm,FPCR	Rm->FPCR	-	-	-	-	-	3 1111 1001 1011 0101 Rm.. ----	
	FMOV imm32,FPCR	imm32->FPCR	Δ	Δ	Δ	Δ	Δ	6 1111 1101 1011 0101 imm32.....	
	FMOV (Rm),FDn	mem64(Rm)->FDn	-	-	-	-	-	3 1111 1001 1010 000X Rm.. fn.-	FDn={X,fn}<<1
	FMOV (Rm,Ri),FDn	mem64(Rm+Ri)->FDn	-	-	-	-	-	4 1111 1011 0100 0111 Ri.. Rm.. fn..-Z-	FDn={Z,fn}
	FMOV (d8,Rm),FDn	mem64(d8+Rm)->FDn	-	-	-	-	-	4 1111 1011 1010 000X Rm.. fn.- d8.....	FDn={X,fn}<<1

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	EC Flag		FCC Flag		Code Size	Machine Code	Note
			VF	ZF	OF	UF	IF		
FMOV	FMOV (d24,Rm),FDn	mem64(d24+Rm)->FDn	-	-	-	-	-	6 1111 1101 1010 000X Rm.. fn.- d24.....	FDn={X,fn}<<1
	FMOV (d32,Rm),FDn	mem64(d32+Rm)->FDn	-	-	-	-	-	7 1111 1110 0100 000X Rm.. fn.- d32.....	FDn={X,fn}<<1
	FMOV (SP),FDn	mem64(SP)->FDn	-	-	-	-	-	3 1111 1001 1010 010X ---- fn.-	FDn={X,fn}<<1
	FMOV (d8,SP),FDn	mem64(d8+SP)->FDn	-	-	-	-	-	4 1111 1011 1010 010X ---- fn.- d8.....	FDn={X,fn}<<1
	FMOV (d24,SP),FDn	mem64(d24+SP)->FDn	-	-	-	-	-	6 1111 1101 1010 010X ---- fn.- d24.....	FDn={X,fn}<<1
	FMOV (d32,SP),FDn	mem64(d32+SP)->FDn	-	-	-	-	-	7 1111 1110 0101 01Y0 fm.- ---- d32.....	FDn={X,fn}<<1
	FMOV FDM,(Rn)	FDM->mem64(Rn)	-	-	-	-	-	3 1111 1001 1011 00Y0 fm.- Rn..	FDm={Y,fn}<<1
	FMOV FDM,(Rn,Ri)	FDM->mem64(Rn+Ri)	-	-	-	-	-	4 1111 1011 0101 0111 Ri.. Rn.. fm.. -Z-	FDm={Z,fn}
	FMOV FDM,(d8,Rn)	FDM->mem64((sign_ext)d8+Rn)	-	-	-	-	-	4 1111 1011 1011 00Y0 fm.- Rn.. d8.....	FDm={Y,fn}<<1
	FMOV FDM,(d24,Rn)	FDM->mem64((sign_ext)d24+Rn)	-	-	-	-	-	6 1111 1101 1011 00Y0 fm.- Rn.. d24.....	FDm={Y,fn}<<1
	FMOV FDM,(d32,Rn)	FDM->mem64(d32+Rn)	-	-	-	-	-	7 1111 1110 0101 00Y0 fm.- Rn.. d32.....	FDm={Y,fn}<<1
	FMOV FDM,(SP)	FDM->mem64(SP)	-	-	-	-	-	3 1111 1001 1011 01Y0 fm.- ----	FDm={Y,fn}<<1
	FMOV FDM,(d8,SP)	FDM->mem64((zero_ext)d8+SP)	-	-	-	-	-	4 1111 1011 1011 01Y0 fm.- ---- d8.....	FDm={Y,fn}<<1
	FMOV FDM,(d24,SP)	FDM->mem64((zero_ext)d24+SP)	-	-	-	-	-	6 1111 1101 1011 01Y0 fm.- ---- d24.....	FDm={Y,fn}<<1
	FMOV FDM,(d32,SP)	FDM->mem64(d32+SP)	-	-	-	-	-	7 1111 1110 0101 01Y0 ---- fm.- d32.....	FDm={Y,fn}<<1
	FMOV (Rm+),FDn	mem64(Rm)->FDn,Rm+0x00000008->Rm	-	-	-	-	-	3 1111 1001 1010 001X Rm.. fn.-	FDn={X,fn}<<1
	FMOV (Rm+,imm8),FDn	mem64(Rm)->FDn, Rm+(sign_ext)imm8->Rm	-	-	-	-	-	4 1111 1011 1010 001X Rm.. fn.- imm8....	FDn={X,fn}<<1
	FMOV (Rm+,imm24),FDn	mem64(Rm)->FDn, Rm+(sign_ext)imm24->Rm	-	-	-	-	-	6 1111 1101 1010 001X Rm.. fn.- imm24...	FDn={X,fn}<<1
	FMOV (Rm+,imm32),FDn	mem64(Rm)->FDn,Rm+imm32->Rm	-	-	-	-	-	7 1111 1110 0100 001X Rm.. fn.- imm32...	FDn={X,fn}<<1
	FMOV FDM,(Rn+),imm8	FDM->mem64(Rn),Rn+0x00000008->Rn	-	-	-	-	-	3 1111 1001 1011 00Y1 fm.- Rn.. imm8....	FDm={Y,fn}<<1
	FMOV FDM,(Rn+,imm24)	FDM->mem64(Rn),Rn+(sign_ext)imm24->Rn	-	-	-	-	-	6 1111 1101 1011 00Y1 fm.- Rn.. imm24...	FDm={Y,fn}<<1
	FMOV FDM,(Ri+,imm32)	FDM->mem64(Rn),Rn+imm32->Rn	-	-	-	-	-	7 1111 1110 0101 00Y1 fm.- Rn.. imm32...	FDm={Y,fn}<<1
FABS	FABS FSn	[FSn]->FSn	Δ	Δ	Δ	Δ	Δ	3 1111 1001 0100 010X ---- Sn..	FSn={X,Sn}
FABS	FABS FSm,FSn	[FSm]->FSn	-	-	-	-	-	4 1111 1011 0100 0100 Sm.. ---- Sn.. X-Z-	FSm={X,Sm},FSn={Z,Sn}
FNEG	FNEG FSn	FSn*(-1)->FSn	-	-	-	-	-	3 1111 1001 0100 011X ---- Sn..	FSn={X,Sn}
FNEG	FNEG FSm,FSn	FSn*(-1)->FSn	-	-	-	-	-	4 1111 1011 0100 0110 Sm.. ---- Sn.. X-Z-	FSm={X,Sm},FSn={Z,Sn}
FRSQRT	FRSQRT FSn	1/Sqrt(FSn)->FSn	Δ	Δ	Δ	Δ	Δ	3 1111 1001 0101 000X ---- Sn..	FSn={X,Sn}
FRSQRT	FSm,FSn	1/Sqrt(FSm)->FSn	Δ	Δ	Δ	Δ	Δ	4 1111 1011 0101 0000 Sm.. ---- Sn.. X-Z-	FSm={X,Sm},FSn={Z,Sn}
FCMP	FSm1,FSm2	FSm2-FSm1:FPCR(EC)	Δ	Δ	Δ	Δ	Δ	3 1111 1001 0101 01YX Sm1. Sm2.	FSm1={Y,Sm1},FSm2={X,Sm2}
FCMP	imm32,FSm	FSm-imm32:FPCR(EC)	Δ	Δ	Δ	Δ	Δ	7 1111 1110 0011 01Y1 Sm.. ---- imm32...	FSm={Y,Sm}
FADD	FSm,FSn	FSm+FSn->FSn	Δ	Δ	Δ	Δ	Δ	3 1111 1001 0110 00YX Sm.. Sn..	FSm={Y,Sm},FSn={X,Sn}
FADD	FSm1,FSm2,FSn	FSm1+FSm2->FSn	Δ	Δ	Δ	Δ	Δ	4 1111 1011 0110 0000 Sm1. Sm2. Sn.. XYZ-	FSm1={X,Sm1},FSm2={Y,Sm2},FSn={Z,Sn}
FADD	imm32,FSm,FSn	FSm+imm32->FSn	Δ	Δ	Δ	Δ	Δ	7 1111 1110 0110 00YX Sm.. Sn.. imm32...	FSm={Y,Sm},FSn={X,Sn}

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	EC Flag		FCC Flag		Code Size	Machine Code	Note
			VF	ZF	OF	UF	IF		
FSUB	FSUB FSm,FSn	FSn-FSm->FSn	Δ	Δ	Δ	Δ	-	3 1111 1001 0110 0101 X Sm.. Sn..	FSm={Y,Sm},FSn={X,Sn}
	FSUB FSm1,FSm2,FSn	FSm2-FSm1->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 0110 0100 Sm1. Sm2. Sn.. XYZ-	FSm1={X,Sm1},FSm2={Y,Sm2},FSn={Z,Sn}
	FSUB imm32,FSm,FSn	FSm-imm32->FSn	Δ	Δ	Δ	Δ	-	7 1111 1110 0110 0101 X Sm.. Sn.. imm32...	FSm={Y,Sm},FSn={X,Sn}
FMUL	FMUL FSm,FSn	FSm*FSn->FSn	Δ	Δ	Δ	Δ	-	3 1111 1001 0111 0010 X Sm.. Sn..	FSm={Y,Sm},FSn={X,Sn}
	FMUL FSm1,FSm2,FSn	FSm1*FSm2->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 0111 0000 Sm1. Sm2. Sn.. XYZ-	FSm1={X,Sm1},FSm2={Y,Sm2},FSn={Z,Sn}
	FMUL imm32,FSm,FSn	FSm*imm32->FSn	Δ	Δ	Δ	Δ	-	7 1111 1110 0111 0010 X Sm.. Sn.. imm32...	FSm={Y,Sm},FSn={X,Sn}
FDIV	FDIV FSm,FSn	FSn/FSm->FSn	Δ	Δ	Δ	Δ	-	3 1111 1001 0111 0101 X Sm.. Sn..	FSm={Y,Sm},FSn={X,Sn}
	FDIV FSm1,FSm2,FSn	FSm2/FSm1->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 0111 0100 Sm1. Sm2. Sn.. XYZ-	FSm1={X,Sm1},FSm2={Y,Sm2},FSn={Z,Sn}
	FDIV imm32,FSm,FSn	FSm/imm32->FSn	Δ	Δ	Δ	Δ	-	7 1111 1110 0111 0101 X Sm.. Sn.. imm32...	FSm={Y,Sm},FSn={X,Sn}
FMADD	FMADD FSm1,FSm2,FSm3,FSn	((FSm1*FSm2)+FSm3)->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 1000 00Sn Sm1. Sm2. Sm3. XYZA	FSm1={X,Sm1},FSm2={Y,Sm2},FSm3={Z,Sm3},FSn={A,Sn}
	FMSUB FSm1,FSm2,FSm3,FSn	((FSm1*FSm2)-FSm3)->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 1000 01Sn Sm1. Sm2. Sm3. XYZA	FSm1={X,Sm1},FSm2={Y,Sm2},FSm3={Z,Sm3},FSn={A,Sn}
	FMADD FSm1,FSm2,FSm3,FSn	(-(FSm1*FSm2)+FSm3)->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 1001 00Sn Sm1. Sm2. Sm3. XYZA	FSm1={X,Sm1},FSm2={Y,Sm2},FSm3={Z,Sm3},FSn={A,Sn}
FMSUB	FMSUB FSm1,FSm2,FSm3,FSn	(-(FSm1*FSm2)-FSm3)->FSn	Δ	Δ	Δ	Δ	-	4 1111 1011 1001 01Sn Sm1. Sm2. Sm3. XYZA	FSm1={X,Sm1},FSm2={Y,Sm2},FSm3={Z,Sm3},FSn={A,Sn}
	FBEQ label	IF(E=1),PC+(sign_ext)d8->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	3 1111 1000 1101 0000 d8.....	
	FBNE label	IF(U=1orL=1orG=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC,	-	-	-	-	-	3 1111 1000 1101 0001 d8.....	
FBGT	FBGT label	IF(G=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC,	-	-	-	-	-	3 1111 1000 1101 0010 d8.....	
	FBGE label	IF(G=1orE=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC,	-	-	-	-	-	3 1111 1000 1101 0011 d8.....	
	FBLT label	IF(L=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	3 1111 1000 1101 0100 d8.....	
FBLE	FBLE label	IF(L=1orE=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	3 1111 1000 1101 0101 d8.....	
	FBUO label	IF(U=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	3 1111 1000 1101 0110 d8.....	
			-	-	-	-	-		

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	EC Flag				FCC Flag				Code Size	Machine Code
			V	F	Z	O	F	U	I	F		
FBCC	FBLG label	IF(L=1orG=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 0111 d8.....
	FBLEG label	IF(L=1orE=1orG=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 1000 d8.....
	FBUG label	IF(U=1orG=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 1001 d8.....
	FBUGE label	IF(U=1orG=1orE=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 1010 d8.....
	FBUL label	IF(U=1orL=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 1011 d8.....
	FBULE label	IF(U=1orL=1orE=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 1100 d8.....
	FBUE label	IF(U=1orE=1) PC+((sign_ext)d8)->nPC, else,PC+CodeSize->nPC	-	-	-	-	-	-	-	-	3	1111 1000 1101 1101 d8.....
	FLEQ	IF(E=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0000
FLCC	FLNE	IF(U=1orL=1orG=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0001
	FLGT	IF(G=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0010
	FLGE	IF(G=1orE=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0011
	FLLT	IF(L=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0100
	FLLE	IF(L=1orE=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0101

MN103E SERIES INSTRUCTION SET

Group	Mnemonic	Operation	EC Flag				FCC Flag				Code Size	Machine Code
			V	F	Z	O	F	I	F	L		
FLCC	FLUO	IF(U=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0110
	FLLG	IF(L=1orG=1),LAR-4->nPC, else,PC+1->nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 0111
	FLLEG	IF(L=1orE=1orG=1),LAR-4->nPC, else,PC+1-> nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 1000
	FLUG	IF(U=1orG=1),LAR-4->nPC, else,PC+1-> nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 1001
	FLUGE	IF(U=1orG=1orE=1),LAR-4->nPC, else,PC+1-> nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 1010
	FLUL	IF(U=1orL=1),LAR-4->nPC, else,PC+1-> nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 1011
	FLULE	IF(U=1orL=1orE=1),LAR-4->nPC, else,PC+1-> nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 1100
	FLUE	IF(U=1orE=1),LAR-4->nPC, else,PC+1-> nPC	-	-	-	-	-	-	-	-	2	1111 0000 1101 1101

MN103E SERIES INSTRUCTION MAP

1st Byte

	Lower				Upper				0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	CLR D0	MOV D0,(abs16)	MOVBU D0,(abs16)	MOVHU D0,(abs16)	CLR D1	MOV D1,(abs16)	MOVBU D1,(abs16)	MOVHU D1,(abs16)	CLR D2	MOV D2,(abs16)	MOVBU D2,(abs16)	MOVHU D2,(abs16)	CLR D3	MOV D3,(abs16)	MOVBU D3,(abs16)	MOVHU D3,(abs16)								
1	EXTB Dn				EXTBU Dn				EXTH Dn				EXTHU Dn											
2	ADD imm8,An				MOV imm16,An				ADD imm8,Dn				MOV imm16,Dn											
3	MOV (abs16),Dn				MOVBU (abs16),Dn				MOVHU (abs16),Dn				MOV SP,An											
4	INC D0	INC A0	MOV D0,(d8,SP)	MOV A0,(d8,SP)	INC D1	INC A1	MOV D1,(d8,SP)	MOV A1,(d8,SP)	INC D2	INC A2	MOV D2,(d8,SP)	MOV A2,(d8,SP)	INC D3	INC A3	MOV D3,(d8,SP)	MOV A3,(d8,SP)								
5	INC4 An				ASL2 Dn				MOV (d8,SP),Dn				MOV (d8,SP),An											
6	MOV Dm,(An)																							
7	MOV (Am),Dn																							
8	MOV Dm,Dn (m=n:MOV imm8,Dn)																							
9	MOV Am,An (m=n:MOV imm8,An)																							
A	CMP Dm,Dn (m=n:MOV imm8,Dn)																							
B	CMP Am,An (m=n:MOV imm8,An)																							
C	BLT (d8,PC)	BGT (d8,PC)	BGE (d8,PC)	BLE (d8,PC)	BCS (d8,PC)	BHI (d8,PC)	BCC (d8,PC)	BLS (d8,PC)	BEQ (d8,PC)	BNE (d8,PC)	BRA (d8,PC)	NOP	JMP (d16,PC)	CALL (d16,PC)	MOVM (SP),regs	MOVM regs,(SP)								
D	LLT	LGT	LGE	LLE	LCS	LHI	LCC	LLS	LEQ	LNE	LRA	SETLB	JMP (d32,PC)	CALL (d32,PC)	RETF	RET								
E	ADD Dm,Dn																							
F	2 byte code							4 byte code	3 byte code			4 byte code		6 byte code		7 byte code	PI							

F0 (2 byte Code)

2nd Byte

Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper																
0	MOV (Am),An															
1	MOV Am,(An)															
2	MOV USP,An				MOV SSP,An				MOV MSP,An				MOV PC,An			
3	MOV A0,USP	MOV A0,SSP	MOV A0,MSP		MOV A1,USP	MOV A1,SSP	MOV A1,MSP		MOV A2,USP	MOV A2,SSP	MOV A2,MSP		MOV A3,USP	MOV A3,SSP	MOV A3,MSP	
4	MOVBU (Am),Dn															
5	MOVBU Dm,(An)															
6	MOVHU (Am),Dn															
7	MOVHU Dm,(An)															
8	BSET Dm,(An)															
9	BCLR Dm,(An)															
A																
B																
C																
D	FLEQ	FLNE	FLGT	FLGE	FLLT	FLLE	FLUO	FLLG	FLLEG	FLUG	FLUGE	FLUL	FLULE	FLUE		
E	SYSCALL imm4															
F	CALLS (An)				JMP (An)								RETS	RTI	TRAP	

MN103E SERIES INSTRUCTION MAP

F1 (2 byte Code)

2nd Byte

Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper																
0	SUB Dm,Dn															
1	SUB Am,Dn															
2	SUB Dm,An															
3	SUB Am,An															
4	ADDC Dm,Dn															
5	ADD Am,Dn															
6	ADD Dm,An															
7	ADD Am,An															
8	SUBC Dm,Dn															
9	CMP Am,Dn															
A	CMP Dm,An															
B																
C																
D	MOV Am,Dn															
E	MOV Dm,An															
F																

F2 (2 byte Code)

2nd Byte

	Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	Upper																
0		AND Dm,Dn															
1		OR Dm,Dn															
2		XOR Dm,Dn															
3		NOT Dn															
4		MUL Dm,Dn															
5		MULU Dm,Dn															
6		DIV Dm,Dn															
7		DIVU Dm,Dn															
8		ROL Dn				ROR Dn											
9		ASL Dm,Dn															
A		LSR Dm,Dn															
B		ASR Dm,Dn															
C																	
D		EXT Dn															
E		MOV MDR,Dn				MOV PSW,Dn								MOV EPSW,Dn			
F		MOV A0,SP	MOV D0,EPSW	MOV D0,MDR	MOV D0,PSW	MOV A1,SP	MOV D1,EPSW	MOV D1,MDR	MOV D1,PSW	MOV A2,SP	MOV D2,EPSW	MOV D2,MDR	MOV D2,PSW	MOV A3,SP	MOV D3,EPSW	MOV D3,MDR	MOV D3,PSW

MN103E SERIES INSTRUCTION MAP

F3 (2 byte Code)

2nd Byte

Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper																
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

F4 (2 byte Code)

2nd Byte

Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper																
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

MN103E SERIES INSTRUCTION MAP

F5 (2 byte Code)

2nd Byte

Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper																
0	MOV Am,Rn															
1																
2																
3																
4	MOV Dm,Rn															
5																
6																
7																
8	MOV Rm,An															
9																
A																
B																
C	MOV Rm,Dn															
D																
E																
F																

F6 (2 byte Code)

2nd Byte

Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper																
0	UDF00 Dm,Dn (MULQ Dm,Dn)															
1	UDF01 Dm,Dn (MULQU Dm,Dn)															
2	UDF02 Dm,Dn (MCST32 Dn / MCST16 Dn / MCST8 Dn)															
3	UDF03 Dm,Dn (MCST9 Dn)															
4	UDF04 Dm,Dn (SAT16 Dm,Dn)															
5	UDF05 Dm,Dn (SAT24 Dm,Dn)															
6	UDF06 Dm,Dn (MCST48 Dn)															
7	UDF07 Dm,Dn (BSCH Dm,Dn)															
8	UDF08 Dm,Dn (SWAP Dm,Dn)															
9	UDF09 Dm,Dn (SWAPH Dm,Dn)															
A	UDF10 Dm,Dn															
B	UDF11 Dm,Dn															
C	UDF12 Dm,Dn (GETCHX Dn)															
D	UDF13 Dm,Dn (GETCLX Dn)															
E	UDF14 Dm,Dn															
F	UDF15 Dm,Dn (GETX Dn)															

MN103E SERIES INSTRUCTION MAP

F7 (4 byte Code)															
2nd Byte															
Lower Upper															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ADD_ADD operand1	CMP_ADD operand1	SUB_ADD operand1	MOV_ADD operand1	ADD_ADD operand2	CMP_ADD operand2	SUB_ADD operand2	MOV_ADD operand2	AND_ADD operand1	DMACH_ADD operand1	XOR_ADD operand1	SWHW_ADD operand1	OR_ADD operand1	SAT16_ADD operand1		
ADD_ADD operand3	CMP_ADD operand3	SUB_ADD operand3	MOV_ADD operand3	ADD_ADD operand4	CMP_ADD operand4	SUB_ADD operand4	MOV_ADD operand4	AND_ADD operand3	DMACH_ADD operand3	XOR_ADD operand3	SWHW_ADD operand3	OR_ADD operand3	SAT16_ADD operand3		
ADD_SUB operand1	CMP_SUB operand1	SUB_SUB operand1	MOV_SUB operand1	ADD_SUB operand2	CMP_SUB operand2	SUB_SUB operand2	MOV_SUB operand2	AND_SUB operand1	DMACH_SUB operand1	XOR_SUB operand1	SWHW_SUB operand1	OR_SUB operand1	SAT16_SUB operand1		
ADD_SUB operand3	CMP_SUB operand3	SUB_SUB operand3	MOV_SUB operand3	ADD_SUB operand4	CMP_SUB operand4	SUB_SUB operand4	MOV_SUB operand4	AND_SUB operand3	DMACH_SUB operand3	XOR_SUB operand3	SWHW_SUB operand3	OR_SUB operand3	SAT16_SUB operand3		
ADD_CMP operand1		SUB_CMP operand1	MOV_CMP operand1	ADD_CMP operand2			MOV_CMP operand2	AND_CMP operand1	DMACH_CMP operand1	XOR_CMP operand1	SWHW_CMP operand1	OR_CMP operand1	SAT16_CMP operand1		
ADD_CMP operand3		SUB_CMP operand3	MOV_CMP operand3	ADD_CMP operand4			MOV_CMP operand4	AND_CMP operand3	DMACH_CMP operand3	XOR_CMP operand3	SWHW_CMP operand3	OR_CMP operand3	SAT16_CMP operand3		
ADD_MOV operand1	CMP_MOV operand1	SUB_MOV operand1	MOV_MOV operand1	ADD_MOV operand2	CMP_MOV operand2	SUB_MOV operand2	MOV_MOV operand2	AND_MOV operand1	DMACH_MOV operand1	XOR_MOV operand1	SWHW_MOV operand1	OR_MOV operand1	SAT16_MOV operand1		
ADD_MOV operand3	CMP_MOV operand3	SUB_MOV operand3	MOV_MOV operand3	ADD_MOV operand4	CMP_MOV operand4	SUB_MOV operand4	MOV_MOV operand4	AND_MOV operand3	DMACH_MOV operand3	XOR_MOV operand3	SWHW_MOV operand3	OR_MOV operand3	SAT16_MOV operand3		
ADD_ASR operand1	CMP_ASR operand1	SUB_ASR operand1	MOV_ASR operand1	ADD_ASR operand2	CMP_ASR operand2	SUB_ASR operand2	MOV_ASR operand2	AND_ASR operand1	DMACH_ASR operand1	XOR_ASR operand1	SWHW_ASR operand1	OR_ASR operand1	SAT16_ASR operand1		
ADD_ASR operand3	CMP_ASR operand3	SUB_ASR operand3	MOV_ASR operand3	ADD_ASR operand4	CMP_ASR operand4	SUB_ASR operand4	MOV_ASR operand4	AND_ASR operand3	DMACH_ASR operand3	XOR_ASR operand3	SWHW_ASR operand3	OR_ASR operand3	SAT16_ASR operand3		
ADD_LSR operand1	CMP_LSR operand1	SUB_LSR operand1	MOV_LSR operand1	ADD_LSR operand2	CMP_LSR operand2	SUB_LSR operand2	MOV_LSR operand2	AND_LSR operand1	DMACH_LSR operand1	XOR_LSR operand1	SWHW_LSR operand1	OR_LSR operand1	SAT16_LSR operand1		
ADD_LSR operand3	CMP_LSR operand3	SUB_LSR operand3	MOV_LSR operand3	ADD_LSR operand4	CMP_LSR operand4	SUB_LSR operand4	MOV_LSR operand4	AND_LSR operand3	DMACH_LSR operand3	XOR_LSR operand3	SWHW_LSR operand3	OR_LSR operand3	SAT16_LSR operand3		
ADD_ASL operand1	CMP_ASL operand1	SUB_ASL operand1	MOV_ASL operand1	ADD_ASL operand2	CMP_ASL operand2	SUB_ASL operand2	MOV_ASL operand2	AND_ASL operand1	DMACH_ASL operand1	XOR_ASL operand1	SWHW_ASL operand1	OR_ASL operand1	SAT16_ASL operand1		
ADD_ASL operand3	CMP_ASL operand3	SUB_ASL operand3	MOV_ASL operand3	ADD_ASL operand4	CMP_ASL operand4	SUB_ASL operand4	MOV_ASL operand4	AND_ASL operand3	DMACH_ASL operand3	XOR_ASL operand3	SWHW_ASL operand3	OR_ASL operand3	SAT16_ASL operand3		
MOV_Lcc operand5															
F															

operand1 : Rm1,Rn1,Rm2,Rn2
operand2 : imm4,Rn1,Rm2,Rn2
operand3 : Rm1,Rn1,imm4,Rn2
operand4 : imm4,Rn1,imm4,Rn2
operand5 : (Rm+,imm4),Rn

MN103E SERIES INSTRUCTION MAP

F8 (3 byte Code)

2nd Byte

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F				
Lower																				
Upper																				
0	MOV (d8,Am),Dn																			
1	MOV Dm,(d8,An)																			
2	MOV (d8,Am),An																			
3	MOV Am,(d8,An)																			
4	MOVBU (d8,Am),Dn																			
5	MOVBU Dm,(d8,An)																			
6	MOVHU (d8,Am),Dn																			
7	MOVHU Dm,(d8,An)																			
8																				
9	MOVBU D0,(d8,SP)		MOVHU D0,(d8,SP)				MOVBU D1,(d8,SP)		MOVHU D1,(d8,SP)				MOVBU D2,(d8,SP)		MOVHU D2,(d8,SP)		MOVBU D3,(d8,SP)		MOVHU D3,(d8,SP)	
A																				
B									MOVBU (d8,SP),Dn				MOVHU (d8,SP),Dn							
C	ASL imm8,Dn				LSR imm8,Dn				ASR imm8,Dn						MOV (USP),regs		MOV (USP),regs			
D	FBEQ (d8,PC)	FBNE (d8,PC)	FBGT (d8,PC)	FBGE (d8,PC)	FBLT (d8,PC)	FBLE (d8,PC)	FBUO (d8,PC)	FBLG (d8,PC)	FBLEG (d8,PC)	FBUG (d8,PC)	FBUGE (d8,PC)	FBUL (d8,PC)	FBULE (d8,PC)	FBUE (d8,PC)						
E	AND imm8,Dn				OR imm8,Dn				BVC (d8,PC)	BVS (d8,PC)	BNC (d8,PC)	BNS (d8,PC)	BTST imm8,Dn							
F	MOV (d8,An),SP				MOV SP,(d8,An)										ADD imm8,SP					

F9 (3 byte Code)

2nd Byte

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Lower Upper																
0	UDF00 imm8,Dn (MULQ imm,Dn)								MOV Rm,Rn	AND Rm,Rn	MOV (Rm),Rn	MAC Rm,Rn				
1					UDF01 imm8,Dn (MULQU imm,Dn)				EXT Rn	OR Rm,Rn	MOV Rm,(Rn)	MACU Rm,Rn				
2	FMOV (Rm),FSn		FMOV (Rm+),FSn		FMOV (SP),FSn		FMOV Rm,FSn		EXTB Rn	XOR Rm,Rn	MOVBU (Rm),Rn	MACB Rm,Rn				
3	FMOV FSm,(Rn)	FMOV FSm,(Rn+)	FMOV FSm,(Rn)	FMOV FSm,(Rn+)	FMOV FSm,(SP)	FMOV FSm,Rn	FMOV FSm,(SP)	FMOV FSm,Rn	EXTBU Rn	NOT Rn	MOVBU Rm,(Rn)	MACBU Rm,Rn				
4	FMOV FSm,FSn				FABS FSn		FNEG FSn		EXTH Rn	ASR Rm,Rn	MOVHU (Rm),Rn	MACH Rm,Rn				
5	FRSQRT FSn		FSQRT FSn		FCMP FSm1,FSm2				EXTHU Rn	LSR Rm,Rn	MOVHU Rm,(Rn)	MACHU Rm,Rn				
6	FADD FSm,FSn				FSUB FSm,FSn				CLR Rn	ASL Rm,Rn	MOV (Rm+),Rn	DMACH Rm,Rn				
7	FMUL FSm,FSn				FDIV FSm,FSn				ADD Rm,Rn	ASL2 Rn	MOV Rm,(Rn+)	DMACHU Rm,Rn				
8									ADDC Rm,Rn	ROR Rn	MOV (SP),Rn	DMULH Rm,Rn				
9									SUB Rm,Rn	ROL Rn	MOV Rm,(SP)	DMULHU Rm,Rn				
A	FMOV (Rm),FDn		FMOV (Rm+),FDn		FMOV (SP),FDn		DCPF (Rm)	DCPF (SP)	SUBC Rm,Rn	MUL Rm,Rn	MOVBU (SP),Rn	SAT16 Rm,Rn				
B	FMOV FDm,(Rn)	FMOV FDm,(Rn+)	FMOV FDm,(Rn)	FMOV FDm,(Rn+)	FMOV FDm,(SP)	FMOV Rm,FPCR	FMOV FDm,(SP)	FPCR,Rn	INC Rn	MULU Rm,Rn	MOVBU Rm,(SP)	MCSTE Rm,Rn				
C	FMOV FDm,FDn				FABS FDn		FNEG FDn		INC4 Rn	DIV Rm,Rn	MOVHU (SP),Rn	SWAP Rm,Rn				
D	FRSQRT FDn		FSQRT FDn		FCMP FDm1,FDm2				CMP Rm,Rn	DIVU Rn	MOV Rm,(SP)	SWAPH Rm,Rn				
E	FADD FDm,FDn				FSUB FDm,FDn				MOV XRm,Rn		MOVHU (Rm+),Rn	SWHW Rm,Rn				
F	FMUL FDm,FDn				FDIV FDm,FDn				MOV Rm,XRn		MOVHU Rm,(Rn+)	BSCH Rm,Rn				

MN103E SERIES INSTRUCTION MAP

FA (4 byte Code)

2nd Byte

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		
Lower																		
Upper																		
0	MOV (d16,Am),Dn																	
1	MOV Dm,(d16,An)																	
2	MOV (d16,Am),An																	
3	MOV Am,(d16,An)																	
4	MOVB (d16,Am),Dn																	
5	MOVB Dm,(d16,An)																	
6	MOVHU (d16,Am),Dn																	
7	MOVHU Dm,(d16,An)																	
8	MOV A0,(abs16)					MOV A1,(abs16)					MOV A2,(abs16)					MOV A3,(abs16)		
9	MOV A0,(d16,SP)	MOV D0,(d16,SP)	MOVB D0,(d16,SP)	MOVHU D0,(d16,SP)	MOV A1,(d16,SP)	MOV D1,(d16,SP)	MOVB D1,(d16,SP)	MOVHU D1,(d16,SP)	MOV A2,(d16,SP)	MOV D2,(d16,SP)	MOVB D2,(d16,SP)	MOVHU D2,(d16,SP)	MOV A3,(d16,SP)	MOV D3,(d16,SP)	MOVB D3,(d16,SP)	MOVHU D3,(d16,SP)		
A	MOV (abs16),An																	
B	MOV (d16,SP),An				MOV (d16,SP),Dn				MOVB (d16,SP),Dn				MOVHU (d16,SP),Dn					
C	ADD imm16,Dn								CMP imm16,Dn									
D	ADD imm16,An								CMP imm16,An									
E	AND imm16,Dn				OR imm16,Dn				XOR imm16,Dn				BTST imm16,Dn					
F	BSET imm8,(d8,An)				BCLR imm8,(d8,An)				BTST imm8,(d8,An)				AND imm16,PSW	OR imm16,PSW	ADD imm16,SP	CALLS (d16,PC)		

384 INSTRUCTION MAP

FB (4 byte Code)																	
2nd Byte																	
Lower	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
Upper																	
0	UDF00 imm16,Dn (MULQ imm,Dn)				FMOV (d8,SP),FSn		FMOV (Rm+,imm8),FSn		UDF01 imm16,Dn (MULQU imm,Dn)		MOV imm8,Rn	AND imm8,Rn	MOV (d8,Rm),Rn	MAC imm8,Rn	AND Rm,Rn,Rd	MOV (abs8),Rn	MAC Rm,Rn,Rd1,Rd2
1	FMOV (d8,Rn),FSn		FMOV (Rm+,imm8),FSn		FMOV (d8,SP),FSn		FMOV (Ri,Rm),FSn		MOVU imm8,Rn		OR imm8,Rn	MOV Rm,(d8,Rn)	MACU imm8,Rn	OR Rm,Rn,Rd	MOV Rm,(abs8)	MACU Rm,Rn,Rd1,Rd2	
2	FMOV (d8,Rn),FSn		FMOV (Rm+,imm8),FSn		FMOV (d8,SP),FSn		FMOV (Ri,Rm),FSn				XOR imm8,Rn	MOVBU (d8,Rm),Rn	MACB imm8,Rn	XOR Rm,Rn,Rd	MOVBU (abs8),Rn	MACB Rm,Rn,Rd	
3	FMOV (d8,Rn),FSn		FMOV (Rm+,imm8),FSn		FMOV (d8,SP),FSn		FMOV (Ri,Rm),FSn					MOVBU Rm,(d8,Rn)	MACBU imm8,Rn		MOVBU Rm,(abs8)	MACBU Rm,Rn,Rd	
4	FTOI FSm,FSn		ITOF FSm,FSn		FABS FSm,FSn		FNEG FSm,FSn		FMOV FSm,(Ri,Rn)		ASR imm8,Rn	MOVHU (d8,Rm),Rn	MACH imm8,Rn	ASR Rm,Rn,Rd	MOVHU (abs8),Rn	MACH Rm,Rn,Rd1,Rd2	
5	FRSQRT FSm,FSn		FTOD FSm,FSn		FSQRT FSm,FSn		DTOF FSm,FSn		FMOV FSm,(Ri,Rn)		LSR imm8,Rn	MOVHU Rm,(d8,Rn)	MACHU imm8,Rn	LSR Rm,Rn,Rd	MOVHU Rm,(abs8)	MACHU Rm,Rn,Rd1,Rd2	
6	FADD FSm1,FSm2,FSn		FSUB FSm1,FSm2,FSn		FDIV FSm1,FSm2,FSn				ADD imm8,Rn		ASL imm8,Rn	MOV (Rm+,imm8),Rn		ASL Rm,Rn,Rd		DMACH Rm,Rn,Rd	
7	FMUL FSm1,FSm2,FSn		FDIV FSm1,FSm2,FSn						ADD imm8,Rn			MOV Rm,(Rn+,imm8)	ADD Rm,Rn,Rd			DMACHU Rm,Rn,Rd	
8	FMADD FSm1,FSm2,FSm3,FSn				FMSUB FSm1,FSm2,FSm3,FSn				FMSUB FSm1,FSm2,FSm3,FSn				ADDC Rm,Rn,Rd		MOV (Ri,Rm),Rn	DMULH Rm,Rn,Rd1,Rd2	
9	FNMADD FSm1,FSm2,FSm3,FSn				FNMSUB FSm1,FSm2,FSm3,FSn				FNMSUB FSm1,FSm2,FSm3,FSn				SUB Rm,Rn,Rd		MOV Rm,(Ri,Rn)	DMULHU Rm,Rn,Rd1,Rd2	
A	FMOV (d8,Rm),FDn		FMOV (Rm+,imm8),FDn		FMOV (d8,SP),FDn		DCPF (Ri,Rm)		SUBC imm8,Rn		MUL imm8,Rn	MOVBU (d8,SP),Rn	SUBC Rm,Rn,Rd	MUL Rm,Rn,Rd1,Rd2	MOVBU (Ri,Rm),Rn	SAT24 Rm,Rn	
B	FMOV (d8,Rn),FDn		FMOV (Rm+,imm8),FDn		FMOV (d8,SP),FDn		FMOV (Ri,Rm),FDn				MULU imm8,Rn	MOVBU Rm,(d8,SP)	MCSTE imm8,Rn		MOVBU Rm,(Ri,Rn)		
C					FABS FSm,FDn		FNEG FSm,FDn		CMP imm8,Rn			MOVHU (d8,SP),Rn			MOVHU (Ri,Rm),Rn		
D	FRSQRT FSm,FDn		FRSQRT FSm,FDn		FSQRT FSm,FDn		FSQRT FSm,FDn					MOVHU Rm,(d8,SP)			MOVHU Rm,(Ri,Rn)		
E	FADD FSm1,FDm2,FDn		FADD FSm1,FDm2,FDn		FSUB FSm1,FDm2,FDn		FSUB FSm1,FDm2,FDn				BTST imm8,Rn	MOVHU (Rm+,imm8),Rn					
F	FMUL FSm1,FDm2,FDn		FMUL FSm1,FDm2,FDn		FDIV FSm1,FDm2,FDn		FDIV FSm1,FDm2,FDn		MOV imm8,XRn			MOVHU Rm,(Rn+,imm8)				BSCH Rm,Rn,Rd	

MN103E SERIES INSTRUCTION MAP

FC (6 byte Code)

2nd Byte

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Lower																
Upper																
0	MOV (d32,Am),Dn															
1	MOV Dm,(d32,An)															
2	MOV (d32,Am),An															
3	MOV Am,(d32,An)															
4	MOVBu (d32,Am),Dn															
5	MOVBu Dm,(d32,An)															
6	MOVHu (d32,Am),Dn															
7	MOVHu Dm,(d32,An)															
8	MOV A0,(abs32)	MOV D0,(abs32)	MOVBu D0,(abs32)	MOVHu D0,(abs32)	MOV A1,(abs32)	MOV D1,(abs32)	MOVBu D1,(abs32)	MOVHu D1,(abs32)	MOV A2,(abs32)	MOV D2,(abs32)	MOVBu D2,(abs32)	MOVHu D2,(abs32)	MOV A3,(abs32)	MOV D3,(abs32)	MOVBu D3,(abs32)	MOVHu D3,(abs32)
9	MOV A0,(d32,SP)	MOV D0,(d32,SP)	MOVBu D0,(d32,SP)	MOVHu D0,(d32,SP)	MOV A1,(d32,SP)	MOV D1,(d32,SP)	MOVBu D1,(d32,SP)	MOVHu D1,(d32,SP)	MOV A2,(d32,SP)	MOV D2,(d32,SP)	MOVBu D2,(d32,SP)	MOVHu D2,(d32,SP)	MOV A3,(d32,SP)	MOV D3,(d32,SP)	MOVBu D3,(d32,SP)	MOVHu D3,(d32,SP)
A	MOV (abs32),An				MOV (abs32),Dn				MOVBu (abs32),Dn				MOVHu (abs32),Dn			
B	MOV (d32,SP),An				MOV (d32,SP),Dn				MOVBu (d32,SP),Dn				MOVHu (d32,SP),Dn			
C	ADD imm32,Dn				SUB imm32,Dn				CMP imm32,Dn				MOV imm32,Dn			
D	ADD imm32,An				SUB imm32,An				CMP imm32,An				MOV imm32,An			
E	AND imm32,Dn				OR imm32,Dn				XOR imm32,Dn				BTST imm32,Dn			
F													AND imm32,EPSt	OR imm32,EPSt	ADD imm32,SP	CALLS (d32,PC)

MN103E SERIES INSTRUCTION MAP

FD (6 byte Code)										2nd Byte						Lower Upper						F					

MN103E SERIES INSTRUCTION MAP

FE (7 byte Code / 5 byte Code)

2nd Byte

Lower
Upper

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BSET imm8, (abs32)	BCLR imm8, (abs32)	BTST imm8, (abs32)						MOV imm32, Rn	AND imm32, Rn	MOV (d32, Rm), Rn	MAC imm32, Rn			MOV (abs32), Rn	
1									MOVU imm32, Rn	OR imm32, Rn	MOV Rn, (d32, Rn)	MACU imm32, Rn			MOV Rm, (abs32)	
2	FMOV (d32, Rm), FSn	FMOV (Rn+, imm32), FSn	FMOV (Rn+, imm32), FSn	FMOV (d32, SP), FSn	FMOV (d32, SP), FSn	FMOV imm32, FSn				XOR imm32, Rn	MOVBU (d32, Rm), Rn	MACB imm32, Rn			MOVBU (abs32), Rn	
3	FMOV FSn, (d32, Rn)	FMOV FSn, (Rn+, imm32)	FMOV FSn, (Rn+, imm32)	FMOV FSn, (d32, SP)	FMOV FSn, (d32, SP)	FMOV FSn, (d32, SP)	FMOV FSn, (d32, SP)	FMOV FSn, (d32, SP)			MOVBU Rm, (d32, Rn)	MACBU imm32, Rn			MOVBU Rm, (abs32)	
4	FMOV (d32, Rm), FDn	FMOV (Rn+, imm32), FDn	FMOV (Rn+, imm32), FDn	FMOV (d32, SP), FDn	FMOV (d32, SP), FDn	DCPF (d32, Rm)				ASR imm32, Rn	MOVHU (d32, Rm), Rn	MACH imm32, Rn			MOVHU (abs32), Rn	
5	FMOV FDm, (d32, Rn)	FMOV FDm, (Rn+, imm32)	FMOV FDm, (Rn+, imm32)	FMOV FDm, (d32, SP)	FMOV FDm, (d32, SP)	FMOV FDm, (d32, SP)				LSR imm32, Rn	MOVHU Rm, (d32, Rn)	MACHU imm32, Rn			MOVHU Rm, (abs32)	
6	FADD imm32, FSn, FSn					FSUB imm32, FSn, FSn				ASL imm32, Rn	MOV (Rn+, imm32), Rn	DMACH imm32, Rn				
7	FMUL imm32, FSn, FSn					FDIV imm32, FSn, FSn			ADD imm32, Rn		MOV Rm, (Rn+, imm32)	DMACHU imm32, Rn				
8	BSET imm8, (abs16)	BCLR imm8, (abs16)	BTST imm8, (abs16)						ADDC imm32, Rn		MOV (d32, SP), Rn	DMULH imm32, Rn				
9									SUB imm32, Rn		MOV Rm, (d32, SP)	DMULHU imm32, Rn				
A									SUBC imm32, Rn	MUL imm32, Rn	MOVBU (d32, SP), Rn					
B										MULU imm32, Rn	MOVBU Rm, (d32, SP)					
C											MOVHU (d32, SP), Rn					
D									CMP imm32, Rn		MOVHU Rm, (d32, SP)					
E										BTST imm32, Rn	MOVHU (Rn+, imm32), Rn					
F									MOV imm32, XRn		MOVHU Rm, (Rn+, imm32)					

INDEX

A

add Am, An -----	76
add Am, Dn -----	76
add Dm, An -----	76
add Dm, Dn -----	76
add imm16, An -----	77
add imm16, Dn -----	77
add imm16, SP -----	77
add imm24, Rn -----	77
add imm32, An -----	77
add imm32, Dn -----	77
add imm32, Rn -----	77
add imm32, SP -----	77
add imm8, An -----	77
add imm8, Dn -----	77
add imm8, Rn -----	77
add imm8, SP -----	77
add Rm, Rn -----	76
add Rm, Rn, Rd -----	76
add_add imm4, Rn1, imm4, Rn2 -----	171, 197
add_add imm4, Rn1, Rm2, Rn2 -----	170, 195
add_add Rm1, Rn1, imm4, Rn2 -----	169, 196
add_add Rm1, Rn1, Rm2, Rn2 -----	168, 194
add_asl imm4, Rn1, imm4, Rn2 -----	171, 219
add_asl imm4, Rn1, Rm2, Rn2 -----	170, 217
add_asl Rm1, Rn1, imm4, Rn2 -----	169, 218
add_asl Rm1, Rn1, Rm2, Rn2 -----	168, 216
add_asr imm4, Rn1, imm4, Rn2 -----	171, 211
add_asr imm4, Rn1, Rm2, Rn2 -----	170, 209
add_asr Rm1, Rn1, imm4, Rn2 -----	169, 210
add_asr Rm1, Rn1, Rm2, Rn2 -----	168, 208
add_cmp imm4, Rn1, imm4, Rn2 -----	171, 203
add_cmp imm4, Rn1, Rm2, Rn2 -----	170, 202
add_cmp Rm1, Rn1, imm4, Rn2 -----	169, 203
add_cmp Rm1, Rn1, Rm2, Rn2 -----	168, 202
add_lsr imm4, Rn1, imm4, Rn2 -----	171, 215
add_lsr imm4, Rn1, Rm2, Rn2 -----	170, 213
add_lsr Rm1, Rn1, imm4, Rn2 -----	169, 214
add_lsr Rm1, Rn1, Rm2, Rn2 -----	168, 212
add_mov imm4, Rn1, imm4, Rn2 -----	171, 207
add_mov imm4, Rn1, Rm2, Rn2 -----	170, 205
add_mov Rm1, Rn1, imm4, Rn2 -----	169, 206
add_mov Rm1, Rn1, Rm2, Rn2 -----	168, 204
add_sub imm4, Rn1, imm4, Rn2 -----	171, 201
add_sub imm4, Rn1, Rm2, Rn2 -----	170, 199
add_sub Rm1, Rn1, imm4, Rn2 -----	169, 200
add_sub Rm1, Rn1, Rm2, Rn2 -----	168, 198
addc Dm, Dn -----	78
addc imm24, Rn -----	79
addc imm32, Rn -----	79
addc imm8, Rn -----	79
addc Rm, Rn -----	78
addc Rm, Rn, Rd -----	78
and Dm, Dn -----	94
and imm16, Dn -----	95
and imm16, PSW -----	96
and imm24, Rn -----	95
and imm32, Dn -----	95
and imm32, EPSW -----	96
and imm32, Rn -----	95
and imm8, Dn -----	95
and imm8, Rn -----	95
and Rm, Rn -----	94
and Rm, Rn, Rd -----	94, 97
and_add Rm1, Rn1, imm4, Rn2 -----	183, 196
and_add Rm1, Rn1, Rm2, Rn2 -----	182, 194
and_asl Rm1, Rn1, imm4, Rn2 -----	183, 218
and_asl Rm1, Rn1, Rm2, Rn2 -----	182, 216
and_asr Rm1, Rn1, imm4, Rn2 -----	183, 210
and_asr Rm1, Rn1, Rm2, Rn2 -----	182, 208
and_cmp Rm1, Rn1, imm4, Rn2 -----	183, 203
and_cmp Rm1, Rn1, Rm2, Rn2 -----	182, 202
and_lsr Rm1, Rn1, imm4, Rn2 -----	183, 214
and_lsr Rm1, Rn1, Rm2, Rn2 -----	182, 212
and_mov Rm1, Rn1, imm4, Rn2 -----	183, 206
and_mov Rm1, Rn1, Rm2, Rn2 -----	182, 204
and_sub Rm1, Rn1, imm4, Rn2 -----	200
and_sub Rm1, Rn1, imm4, Rn2 -----	183
and_sub Rm1, Rn1, Rm2, Rn2 -----	182, 198
asl imm8, Dn -----	116
asl Dm, Dn -----	114
asl imm24, Rn -----	116
asl imm32, Rn -----	116
asl imm8, Rn -----	116
asl Rm, Rn -----	114
asl Rm, Rn, Rd -----	115
asl2 Dn -----	117

asl2 Rn	117
asr imm8, Dn	110
asr Dm, Dn	108
asr imm24, Rn	110
asr imm32, Rn	110
asr imm8, Rn	110
asr Rm, Rn	108
asr Rm, Rn, Rd	109

B

bcc label	120
bclr imm8, (abs16)	107
bclr imm8, (abs32)	107
bclr imm8, (d8, An)	107
bclr Dm, (An)	106
bcs label	120
beq label	120
bge label	120
bgt label	120
bhi label	120
ble label	120
bls label	120
blt label	120
bnc label	120
bne label	120
bns label	120
bra label	120
bsch Rm, Rn	166
bsch Rm, Rn, Rd	167
bset imm8, (abs32)	105
bset imm8, (d8, An)	105
bset Dm, (An)	104
bset imm8, (abs16)	105
btst imm16, Dn	103
btst imm24, Rn	103
btst imm32, Dn	103
btst imm32, Rn	103
btst imm8, (abs16)	103
btst imm8, (abs32)	103
btst imm8, (d8, An)	103
btst imm8, Dn	103
btst imm8, Rn	103
bvc label	120
bvs label	120

C

CALL (d16, PC), regs, imm8	125
CALL (d32, PC), regs, imm8	126
call label, regs, imm8	125, 126
calls (An)	127
calls label	128
CALLS (d16, PC)	128
CALLS (d32, PC)	128
clr Dn	74
clr Rn	74
cmp imm16, An	93
cmp imm16, Dn	93
cmp imm24, Rn	93
cmp imm32, An	93
cmp imm32, Dn	93
cmp imm32, Rn	93
cmp imm8, An	93
cmp imm8, Dn	93
cmp imm8, Rn	93
cmp Am, An	92
cmp Am, Dn	92
cmp Dm, An	92
cmp Dm, Dn	92
cmp Rm, Rn	92
cmp_add imm4, Rn1, imm4, Rn2	173, 197
cmp_add imm4, Rn1, Rm2, Rn2	195
cmp_add Rm1, Rn1, imm4, Rn2	172, 196
cmp_add Rm1, Rn1, Rm2, Rn2	172, 194
cmp_asl imm4, Rn1, imm4, Rn2	173, 219
cmp_asl imm4, Rn1, Rm2, Rn2	217
cmp_asl Rm1, Rn1, imm4, Rn2	172, 218
cmp_asl Rm1, Rn1, Rm2, Rn2	172, 216
cmp_asr imm4, Rn1, imm4, Rn2	173, 211
cmp_asr imm4, Rn1, Rm2, Rn2	209
cmp_asr Rm1, Rn1, imm4, Rn2	172, 210
cmp_asr Rm1, Rn1, Rm2, Rn2	172, 208
cmp_lsr imm4, Rn1, imm4, Rn2	173, 215
cmp_lsr imm4, Rn1, Rm2, Rn2	213
cmp_lsr Rm1, Rn1, imm4, Rn2	172, 214
cmp_lsr Rm1, Rn1, Rm2, Rn2	172, 212
cmp_mov imm4, Rn1, imm4, Rn2	173, 207
cmp_mov imm4, Rn1, Rm2, Rn2	205
cmp_mov Rm1, Rn1, imm4, Rn2	172, 206
cmp_mov Rm1, Rn1, Rm2, Rn2	172, 204
cmp_sub imm4, Rn1, imm4, Rn2	173, 201
cmp_sub imm4, Rn1, Rm2, Rn2	199

cmp_sub Rm1, Rn1, imm4, Rn2 ----- 172, 200
 cmp_sub Rm1, Rn1, Rm2, Rn2 ----- 172, 198

D

dcpf (d24, Rm) ----- 75
 dcpf (d32, Rm) ----- 75
 dcpf (d8, Rm) ----- 75
 dcpf (Ri, Rm) ----- 75
 dcpf (Rm) ----- 75
 dcpf (SP) ----- 75
 div Rm, Rn ----- 88
 div Dm, Dn ----- 88
 divu Rm, Rn ----- 89
 divu Dm, Dn ----- 89
 dmach imm32, Rn ----- 142
 dmach Rm, Rn ----- 141
 dmach Rm, Rn, Rd ----- 141
 dmach_add Rm1, Rn1, imm4, Rn2 ----- 189
 dmach_add Rm1, Rn1, Rm2, Rn2 ----- 188, 194
 dmach_add Rm1, Rn1, imm4, Rn2 ----- 196
 dmach_asl Rm1, Rn1, imm4, Rn2 ----- 189, 218
 dmach_asl Rm1, Rn1, Rm2, Rn2 ----- 188, 216
 dmach_asr Rm1, Rn1, imm4, Rn2 ----- 189, 210
 dmach_asr Rm1, Rn1, Rm2, Rn2 ----- 188, 208
 dmach_cmp Rm1, Rn1, imm4, Rn2 ----- 189
 dmach_cmp Rm1, Rn1, imm4, Rn2 ----- 203
 dmach_cmp Rm1, Rn1, Rm2, Rn2 ----- 188, 202
 dmach_lsr Rm1, Rn1, imm4, Rn2 ----- 189, 214
 dmach_lsr Rm1, Rn1, Rm2, Rn2 ----- 188, 212
 dmach_mov Rm1, Rn1, imm4, Rn2 ----- 189
 dmach_mov Rm1, Rn1, Rm2, Rn2 ----- 188, 204
 dmach_mov Rm1, Rn1, imm4, Rn2 ----- 206
 dmach_sub Rm1, Rn1, imm4, Rn2 ----- 189
 dmach_sub Rm1, Rn1, Rm2, Rn2 ----- 188, 198
 dmach_sub Rm1, Rn1, imm4, Rn2 ----- 200
 dmachu imm32, Rn ----- 144
 dmachu Rm, Rn ----- 143
 dmachu Rm, Rn, Rd ----- 143
 dmulh imm32, Rn ----- 138
 dmulh Rm, Rn ----- 137
 dmulh Rm, Rn, Rd1, Rd2 ----- 137
 dmulhu imm32, Rn ----- 140
 dmulhu Rm, Rn ----- 139
 dmulhu Rm, Rn, Rd1, Rd2 ----- 139

E

ext Dn ----- 69
 ext Rn ----- 69
 extb Rm, Rn ----- 72
 extb Dn ----- 72
 extb Rn ----- 72
 extbu Rm, Rn ----- 73
 extbu Dn ----- 73
 extbu Rn ----- 73
 exth Rm, Rn ----- 70
 exth Dn ----- 70
 exth Rn ----- 70
 exthu Dn ----- 71
 exthu Rm, Rn ----- 71
 exthu Rn ----- 71

F

fabs FSm, FS n ----- 253
 fabs FS n ----- 253
 fadd FSm, FS n ----- 260
 fadd FSm1, FSm2, FS n ----- 260
 fadd imm32, FSm, FS n ----- 261
 fbeq label ----- 296
 fbge label ----- 296
 fbgt label ----- 296
 fble label ----- 296
 fbleg label ----- 296
 fblt label ----- 296
 fbne label ----- 296
 fbue label ----- 296
 fbug label ----- 296
 fbuge label ----- 296
 fbul label ----- 296
 fbule label ----- 296
 fbuo label ----- 296
 fcmp FSm1, FSm2 ----- 257
 fcmp imm32, FSm, FS n ----- 257
 fdiv FSm, FS n ----- 269
 fdiv FSm1, FSm2, FS n ----- 269
 fdiv imm32, FSm, FS n ----- 270
 fleq ----- 297
 flge ----- 297
 flgt ----- 297
 flle ----- 297
 flleg ----- 297

flg	297	fmov FdM,(Rn+,imm32)	252
fllt	297	fmov FdM,(Rn+,imm8)	252
flne	297	fmov FdM,(Rn+)	251
flue	297	fmov FdM,(SP)	248
flug	297	fmov FPCR,Rn	244
fluge	297	fmov FSm,(d24,Rn)	237
flul	297	fmov FSm,(d24,SP)	237
flule	297	fmov FSm,(d32,Rn)	237
fluo	297	fmov FSm,(d32,SP)	237
fmadd FSm1, FSm2, FSm3, FSn	272	fmov FSm,(d8,Rn)	237
fmov (d24,Rm), FSn	236	fmov FSm,(d8,SP)	237
fmov (d24,SP), FSn	236	fmov FSm,(Rn,Ri)	237
fmov (d32,Rm), FSn	236	fmov FSm,(Rn)	237
fmov (d32,SP), FSn	236	fmov FSm,(Rn+,imm24)	241
fmov (d8,Rm), FSn	236	fmov FSm,(Rn+,imm32)	241
fmov (d8,SP), FSn	236	fmov FSm,(Rn+,imm8)	241
fmov (Rm,Ri), FSn	236	fmov FSm,(SP)	237
fmov (Rm), FSn	236	fmov FSm,FSn	242
fmov (Rm+), FSn	238	fmov FSn,(Rn+)	240
fmov (SP), FSn	236	fmov imm32,FPCR	246
fmov FSm,Rn	242	fmov imm32,FSn	243
fmov (d24,Rm),FDn	247	fmov Rm,FPCR	245
fmov (d24,SP),FDn	247	fmov Rm,FSn	243
fmov (d32,Rm),FDn	247	fmsub FSm1, FSm2, FSm3, FSn	278
fmov (d32,SP),FDn	247	fmul FSm,FSn	266
fmov (d8,Rm),FDn	247	fmul FSm1, FSm2, FSn	266
fmov (d8,SP),FDn	247	fmul imm32, FSm, FSn	267
fmov (Rm,Ri),FDn	247	fneg FSm,FSn	254
fmov (Rm),FDn	247	fneg FSn	254
fmov (Rm+,imm24),FDn	250	fnmadd FSm1, FSm2, FSm3, FSn	284
fmov (Rm+,imm32),FDn	250	fnmsub FSm1, FSm2, FSm3, FSn	290
fmov (Rm+,imm8),FDn	250	frsqrtn FSm	255
fmov (Rm+),FDn	249	frsqrtn FSm,FSn	255
fmov (Rn+,imm24),FSn	239	fsub FSm1, FSm2, FSn	263
fmov (Rn+,imm32),FSn	239	fsub imm32, FSm, FSn	264
fmov (Rn+,imm8),FSn	239	fsud FSm,FSn	263
fmov (SP),FDn	247		
fmov FdM,(d24,Rn)	248	I	
fmov FdM,(d24,SP)	248	inc An	90
fmov FdM,(d32,Rn)	248	inc Dn	90
fmov FdM,(d32,SP)	248	inc Rn	90
fmov FdM,(d8,Rn)	248	inc4 An	91
fmov FdM,(d8,SP)	248	inc4 Rn	91
fmov FdM,(Rn,Ri)	248	J	
fmov FdM,(Rn)	248	jmp label	123, 124
fmov FdM,(Rn+,imm24)	252		

jmp (An)-----	123
JMP (d16, PC)-----	123
JMP (d32, PC)-----	124

L

lcc label-----	121
lcs label-----	121
leq label-----	121
lge label-----	121
lgt label-----	121
lhi label-----	121
lle label-----	121
lls label-----	121
llt label-----	121
lne label-----	121
lra label-----	121
lsr imm8, Dn-----	113
lsr Dm, Dn-----	111
lsr imm24, Rn-----	113
lsr imm32, Rn-----	113
lsr imm8, Rn-----	113
lsr Rm, Rn-----	111
lsr Rm, Rn, Rd-----	112

M

mac imm24, Rn-----	146
mac imm32, Rn-----	146
mac imm8, Rn-----	146
mac Rm, Rn-----	145
mac Rm, Rn, Rd1, Rd2-----	145
macb imm8, Rn-----	154
macb imm24, Rn-----	154
macb imm32, Rn-----	154
macb Rm, Rn-----	153
macb Rm, Rn, Rd-----	153
macbu imm8, Rn-----	156
macbu imm24, Rn-----	156
macbu imm32, Rn-----	156
macbu Rm, Rn-----	155
macbu Rm, Rn, Rd-----	155
mach imm8, Rn-----	150
mach imm24, Rn-----	150
mach imm32, Rn-----	150
mach Rm, Rn-----	149
mach Rm, Rn, Rd1, Rd2-----	149
machu imm8, Rn-----	152

machu imm24, Rn-----	152
machu imm32, Rn-----	152
machu Rm, Rn-----	151
machu Rm, Rn, Rd1, Rd2-----	151
macu imm24, Rn-----	148
macu imm32, Rn-----	148
macu imm8, Rn-----	148
macu Rm, Rn-----	147
macu Rm, Rn, Rd1, Rd2-----	147
mcste imm8, Rn-----	165
mcste Rm, Rn-----	163
mov (Rm+), Rn-----	50
mov Am, An-----	33
mov Am, Dn-----	33
mov Am, MSP-----	34
mov Am, Rn-----	33
mov Am, SP-----	40
mov Am, SSP-----	35
mov Am, USP-----	36
mov Dm, An-----	33
mov Dm, Dn-----	33
mov Dm, EPSW-----	37
mov Dm, MDR-----	39
mov Dm, PSW-----	38
mov Dm, Rn-----	33
mov EPSW, Dn-----	37
mov imm16, An-----	33
mov imm16, Dn-----	33
mov imm24, Rn-----	33
mov imm32, An-----	33
mov imm32, Dn-----	33
mov imm32, Rn-----	33
mov imm8, An-----	33
mov imm8, Dn-----	33
mov imm8, Rn-----	33
mov MCRH, Rn-----	42
mov MCRL, Rn-----	43
mov MCVF, Rn-----	44
mov MDR, Dn-----	39
mov MDRQ, Rn-----	41
mov MSP, An-----	34
mov PC, An-----	45
mov PSW, Dn-----	38
mov Rm, (Rn+)-----	51
mov Rm, An-----	33
mov Rm, Dn-----	33

mov Rm, MCRH	42	mov Am,(abs16)	48
mov Rm, MCRL	43	mov Am,(abs32)	48
mov Rm, MCVF	44	mov Am,(An)	48
mov Rm, MDRQ	41	mov Am,(d16,An)	48
mov Rm, Rn	33	mov Am,(d16,SP)	48
mov Rm, SP	40	mov Am,(d32,An)	48
mov SP, An	40	mov Am,(d32,SP)	49
mov SP, Rn	40	mov Am,(d8,An)	48
mov SSP, An	35	mov Am,(d8,SP)	48
mov USP, An	36	mov Am,(Di,An)	48
mov (abs16),An	46	mov Dm,(abs16)	48
mov (abs16),Dn	46	mov Dm,(abs32)	48
mov (abs24),Rn	46	mov Dm,(An)	48
mov (abs32),An	46	mov Dm,(d16,An)	48
mov (abs32),Dn	46	mov Dm,(d16,SP)	48
mov (abs32),Rn	46	mov Dm,(d32,An)	48
mov (abs8),Rn	46	mov Dm,(d32,SP)	48
mov (Am),An	46	mov Dm,(d8,An)	48
mov (Am),Dn	46	mov Dm,(d8,SP)	48
mov (d16,Am),An	46	mov Dm,(Di,An)	48
mov (d16,Am),Dn	46	mov imm24, MCRH	42
mov (d16,SP),An	46	mov imm24, MCRL	43
mov (d16,SP),Dn	46	mov imm24, MCVF	44
mov (d24,Rm),Rn	46	mov imm24, MDRQ	41
mov (d24,SP),Rn	47	mov imm24, SP	40
mov (d32,Am),An	46	mov imm32, MCRH	42
mov (d32,Am),Dn	46	mov imm32, MCRL	43
mov (d32,Rm),Rn	46	mov imm32, MCVF	44
mov (d32,SP),An	47	mov imm32, MDRQ	41
mov (d32,SP),Dn	46	mov imm32, SP	40
mov (d32,SP),Rn	47	mov imm8, MCRH	42
mov (d8,Am),An	46	mov imm8, MCRL	43
mov (d8,Am),Dn	46	mov imm8, MCVF	44
mov (d8,Am),SP	47	mov imm8, MDRQ	41
mov (d8,Rm),Rn	46	mov imm8, SP	40
mov (d8,SP),An	46	mov Rm, (Rn+, imm24)	51
mov (d8,SP),Dn	46	mov Rm, (Rn+, imm32)	51
mov (d8,SP),Rn	47	mov Rm, (Rn+, imm8)	51
mov (Di,Am),An	46	mov Rm,(abs24)	48
mov (Di,Am),Dn	46	mov Rm,(abs32)	48
mov (Ri,Rm),Rn	46	mov Rm,(abs8)	48
mov (Rm),Rn	46	mov Rm,(d24,Rn)	48
mov (Rm+, imm24), Rn	50	mov Rm,(d24,SP)	49
mov (Rm+, imm32), Rn	50	mov Rm,(d32,Rn)	48
mov (Rm+, imm8), Rn	50	mov Rm,(d32,SP)	49
mov (SP),Rn	47	mov Rm,(d8,Rn)	48
		mov Rm,(d8,SP)	49

mov Rm,(Ri,Rn) -----	48	movbu (abs32),Rn -----	59
mov Rm,(Rn) -----	48	movbu (abs8),Rn -----	59
mov Rm,(SP) -----	49	movbu (Am),Dn -----	59
mov SP,(d8,An) -----	49	movbu (d16,Am),Dn -----	59
mov_add imm4, Rn1, imm4, Rn2 -----	181, 197	movbu (d16,SP),Dn -----	59
mov_add imm4, Rn1, Rm2, Rn2 -----	180, 195	movbu (d24,Rm),Rn -----	59
mov_add Rm1, Rn1, imm4, Rn2 -----	179, 196	movbu (d24,SP),Rn -----	59
mov_add Rm1, Rn1, Rm2, Rn2 -----	178, 194	movbu (d32,Am),Dn -----	59
mov_asl imm4, Rn1, imm4, Rn2 -----	181, 219	movbu (d32,Rm),Rn -----	59
mov_asl imm4, Rn1, Rm2, Rn2 -----	180, 217	movbu (d32,SP),Dn -----	59
mov_asl Rm1, Rn1, imm4, Rn2 -----	179, 218	movbu (d32,SP),Rn -----	59
mov_asl Rm1, Rn1, Rm2, Rn2 -----	178, 216	movbu (d8,Am),Dn -----	59
mov_asr imm4, Rn1, imm4, Rn2 -----	181, 211	movbu (d8,Rm),Rn -----	59
mov_asr imm4, Rn1, Rm2, Rn2 -----	180, 209	movbu (d8,SP),Dn -----	59
mov_asr Rm1, Rn1, imm4, Rn2 -----	179, 210	movbu (d8,SP),Rn -----	59
mov_asr Rm1, Rn1, Rm2, Rn2 -----	178, 208	movbu (Di,Am),Dn -----	59
mov_cmp imm4, Rn1, imm4, Rn2 -----	181, 203	movbu (Ri,Rm),Rn -----	59
mov_cmp imm4, Rn1, Rm2, Rn2 -----	180, 202	movbu (Rm),Rn -----	59
mov_cmp Rm1, Rn1, imm4, Rn2 -----	179, 203	movbu (SP),Rn -----	59
mov_cmp Rm1, Rn1, Rm2, Rn2 -----	178, 202	movbu Dm,(abs16) -----	60
mov_lcc (Rm+, imm4), Rn -----	221	movbu Dm,(abs32) -----	60
mov_lcs (Rm+, imm4), Rn -----	221	movbu Dm,(An) -----	60
mov_leq (Rm+, imm4), Rn -----	221	movbu Dm,(d16,An) -----	60
mov_lge (Rm+, imm4), Rn -----	221	movbu Dm,(d16,SP) -----	60
mov_lgt (Rm+, imm4), Rn -----	221	movbu Dm,(d32,An) -----	60
mov_lhi (Rm+, imm4), Rn -----	221	movbu Dm,(d32,SP) -----	60
mov_lle (Rm+, imm4), Rn -----	221	movbu Dm,(d8,An) -----	60
mov_lls (Rm+, imm4), Rn -----	221	movbu Dm,(d8,SP) -----	60
mov_llt (Rm+, imm4), Rn -----	221	movbu Dm,(Di,An) -----	60
mov_lne (Rm+, imm4), Rn -----	221	movbu Rm,(abs24) -----	60
mov_lra (Rm+, imm4), Rn -----	221	movbu Rm,(abs32) -----	60
mov_lsr imm4, Rn1, imm4, Rn2 -----	181, 215	movbu Rm,(abs8) -----	60
mov_lsr imm4, Rn1, Rm2, Rn2 -----	180, 213	movbu Rm,(d24,Rn) -----	60
mov_lsr Rm1, Rn1, imm4, Rn2 -----	179, 214	movbu Rm,(d24,SP) -----	60
mov_lsr Rm1, Rn1, Rm2, Rn2 -----	178, 212	movbu Rm,(d32,Rn) -----	60
mov_mov imm4, Rn1, imm4, Rn2 -----	181, 207	movbu Rm,(d32,SP) -----	60
mov_mov imm4, Rn1, Rm2, Rn2 -----	180, 205	movbu Rm,(d8,Rn) -----	60
mov_mov Rm1, Rn1, imm4, Rn2 -----	179, 206	movbu Rm,(d8,SP) -----	60
mov_mov Rm1, Rn1, Rm2, Rn2 -----	178, 204	movbu Rm,(Ri,Rn) -----	60
mov_sub imm4, Rn1, imm4, Rn2 -----	181, 201	movbu Rm,(Rn) -----	60
mov_sub imm4, Rn1, Rm2, Rn2 -----	180, 199	movbu Rm,(SP) -----	60
mov_sub Rm1, Rn1, imm4, Rn2 -----	179, 200	movhu (Rm+), Rn -----	57
mov_sub Rm1, Rn1, Rm2, Rn2 -----	178, 198	movhu Rm, (Rn+) -----	58
movbu (abs16),Dn -----	59	movhu (abs16),Dn -----	53
movbu (abs24),Rn -----	59	movhu (abs24),Rn -----	53
movbu (abs32),Dn -----	59	movhu (abs32),Dn -----	53

movhu (abs32),Rn	53	movhu Rm(Rn+, imm8)	58
movhu (abs8),Rn	53	movm (SP), [reg1, reg2,,,regn]	61
movhu (Am),Dn	53	movm (USP),[reg1, reg2,,,regn]	65
movhu (d16,Am),Dn	53	movm [reg1, reg2,,,regn], (SP)	63
movhu (d16,SP),Dn	53	movm [reg1, reg2,,,regn], (USP)	67
movhu (d24,Rm),Rn	53	movu imm24, Rn	52
movhu (d24,SP),Rn	53	movu imm32, Rn	52
movhu (d32,Am),Dn	53	movu imm8, Rn	52
movhu (d32,Rm),Rn	53	mul Dm, Dn	84
movhu (d32,SP),Dn	53	mul imm24, Rn	85
movhu (d32,SP),Rn	53	mul imm32, Rn	85
movhu (d8,Am),Dn	53	mul imm8, Rn	85
movhu (d8,Rm),Rn	53	mul Rm, Rn	84
movhu (d8,SP),Dn	53	mul Rm, Rn, Rd1, Rd2	84
movhu (d8,SP),Rn	53	mulu Dm, Dn	86
movhu (Di,Am),Dn	53	mulu imm24, Rn	87
movhu (Ri,Rm),Rn	53	mulu imm32, Rn	87
movhu (Rm),Rn	53	mulu imm8, Rn	87
movhu (Rm+, imm24), Rn	57	mulu Rm, Rn	86
movhu (Rm+, imm32), Rn	57	mulu Rm, Rn, Rd1, Rd2	86
movhu (Rm+, imm8), Rn	57		
movhu (SP),Rn	53	N	
movhu Dm,(abs16)	55	nop	134
movhu Dm,(abs32)	55	not Dn	102
movhu Dm,(An)	55	not Rn	102
movhu Dm,(d16,An)	55		
movhu Dm,(d16,SP)	55	O	
movhu Dm,(d32,An)	55	or imm8, Rn	98
movhu Dm,(d32,SP)	55	or Dm, Dn	97
movhu Dm,(d8,An)	55	or imm16, Dn	98
movhu Dm,(d8,SP)	55	or imm16, PSW	99
movhu Dm,(Di,An)	55	or imm24, Rn	98
movhu Rm,(abs24)	55	or imm32, Dn	98
movhu Rm,(abs32)	55	or imm32, EPSW	99
movhu Rm,(abs8)	55	or imm32, Rn	98
movhu Rm,(d24,Rn)	55	or imm8, Dn	98
movhu Rm,(d24,SP)	55	or Rm, Rn	97
movhu Rm,(d32,Rn)	55	or_add Rm1, Rn1, imm4, Rn2	185, 196
movhu Rm,(d32,SP)	55	or_add Rm1, Rn1, Rm2, Rn2	184, 194
movhu Rm,(d8,Rn)	55	or_asl Rm1, Rn1, imm4, Rn2	185, 218
movhu Rm,(d8,SP)	55	or_asl Rm1, Rn1, Rm2, Rn2	184, 216
movhu Rm,(Ri,Rn)	55	or_asr Rm1, Rn1, imm4, Rn2	185, 210
movhu Rm,(Rn)	55	or_asr Rm1, Rn1, Rm2, Rn2	184, 208
movhu Rm,(SP)	55	or_cmp Rm1, Rn1, imm4, Rn2	185, 203
movhu Rm(Rn+, imm24)	58	or_cmp Rm1, Rn1, Rm2, Rn2	184, 202
movhu Rm(Rn+, imm32)	58	or_lsr Rm1, Rn1, imm4, Rn2	185, 214

or_lsr Rm1, Rn1, Rm2, Rn2 ----- 184, 212
 or_mov Rm1, Rn1, imm4, Rn2 ----- 185, 206
 or_mov Rm1, Rn1, Rm2, Rn2 ----- 184, 204
 or_sub Rm1, Rn1, imm4, Rn2 ----- 185, 200
 or_sub Rm1, Rn1, Rm2, Rn2 ----- 184, 198

P

pi ----- 136

R

ret ----- 129
 retf ----- 130
 rets ----- 131
 rol Dn ----- 119
 rol Rn ----- 119
 ror Dn ----- 118
 ror Rn ----- 118
 rti ----- 132

S

sat16 Rm, Rn ----- 160
 sat16_add Rm1, Rn1, imm4, Rn2 ----- 193, 196
 sat16_add Rm1, Rn1, Rm2, Rn2 ----- 192, 194
 sat16_asl Rm1, Rn1, imm4, Rn2 ----- 193, 218
 sat16_asl Rm1, Rn1, Rm2, Rn2 ----- 192, 216
 sat16_asr Rm1, Rn1, imm4, Rn2 ----- 193, 210
 sat16_asr Rm1, Rn1, Rm2, Rn2 ----- 192, 208
 sat16_cmp Rm1, Rn1, imm4, Rn2 ----- 193, 203
 sat16_cmp Rm1, Rn1, Rm2, Rn2 ----- 192, 202
 sat16_lsr Rm1, Rn1, imm4, Rn2 ----- 193, 214
 sat16_lsr Rm1, Rn1, Rm2, Rn2 ----- 192, 212
 sat16_mov Rm1, Rn1, imm4, Rn2 ----- 193, 206
 sat16_mov Rm1, Rn1, Rm2, Rn2 ----- 192, 204
 sat16_sub Rm1, Rn1, imm4, Rn2 ----- 193, 200
 sat16_sub Rm1, Rn1, Rm2, Rn2 ----- 192, 198
 sat24 Rm, Rn ----- 161
 setlb ----- 122
 sub imm24, An ----- 81
 sub imm32, Rn ----- 81
 sub imm8, Dn ----- 81
 sub Am, An ----- 80
 sub Am, Dn ----- 80
 sub Dm, An ----- 80
 sub Dm, Dn ----- 80
 sub imm32, An ----- 81
 sub imm32, Dn ----- 81

sub Rm, Rn ----- 80
 sub Rm, Rn, Rd ----- 80
 sub_add imm4, Rn1, imm4, Rn2 ----- 177, 197
 sub_add imm4, Rn1, Rm2, Rn2 ----- 176, 195
 sub_add Rm1, Rn1, imm4, Rn2 ----- 175, 196
 sub_add Rm1, Rn1, Rm2, Rn2 ----- 174, 194
 sub_asl imm4, Rn1, imm4, Rn2 ----- 177, 219
 sub_asl imm4, Rn1, Rm2, Rn2 ----- 176, 217
 sub_asl Rm1, Rn1, imm4, Rn2 ----- 175, 218
 sub_asl Rm1, Rn1, Rm2, Rn2 ----- 174, 216
 sub_asr imm4, Rn1, imm4, Rn2 ----- 177, 211
 sub_asr imm4, Rn1, Rm2, Rn2 ----- 176, 209
 sub_asr Rm1, Rn1, imm4, Rn2 ----- 175, 210
 sub_asr Rm1, Rn1, Rm2, Rn2 ----- 174, 208
 sub_cmp imm4, Rn1, imm4, Rn2 ----- 177, 203
 sub_cmp imm4, Rn1, Rm2, Rn2 ----- 176, 202
 sub_cmp Rm1, Rn1, imm4, Rn2 ----- 175, 203
 sub_cmp Rm1, Rn1, Rm2, Rn2 ----- 174, 202
 sub_lsr imm4, Rn1, imm4, Rn2 ----- 177, 215
 sub_lsr imm4, Rn1, Rm2, Rn2 ----- 176, 213
 sub_lsr Rm1, Rn1, imm4, Rn2 ----- 175, 214
 sub_lsr Rm1, Rn1, Rm2, Rn2 ----- 174, 212
 sub_mov imm4, Rn1, imm4, Rn2 ----- 177, 207
 sub_mov imm4, Rn1, Rm2, Rn2 ----- 176, 205
 sub_mov Rm1, Rn1, imm4, Rn2 ----- 175, 206
 sub_mov Rm1, Rn1, Rm2, Rn2 ----- 174, 204
 sub_sub imm4, Rn1, imm4, Rn2 ----- 177, 201
 sub_sub imm4, Rn1, Rm2, Rn2 ----- 176, 199
 sub_sub Rm1, Rn1, imm4, Rn2 ----- 175, 200
 sub_sub Rm1, Rn1, Rm2, Rn2 ----- 174, 198
 subc Dm, Dn ----- 82
 subc imm24, Rn ----- 83
 subc imm32, Rn ----- 83
 subc imm8, Rn ----- 83
 subc Rm, Rn ----- 82
 subc Rm, Rn, Rd ----- 82
 swap Rm, Rn ----- 158
 swaph Rm, Rn ----- 159
 swhw Rm, Rn ----- 157
 swhw_add Rm1, Rn1, imm4, Rn2 ----- 191, 196
 swhw_add Rm1, Rn1, Rm2, Rn2 ----- 190, 194
 swhw_asl Rm1, Rn1, imm4, Rn2 ----- 191, 218
 swhw_asl Rm1, Rn1, Rm2, Rn2 ----- 190, 216
 swhw_asr Rm1, Rn1, imm4, Rn2 ----- 191, 210
 swhw_asr Rm1, Rn1, Rm2, Rn2 ----- 190, 208
 swhw_cmp Rm1, Rn1, imm4, Rn2 ----- 191, 203

swhw_cmp Rm1, Rn1, Rm2, Rn2 ----- 190, 202
 swhw_lsr Rm1, Rn1, imm4, Rn2 ----- 191, 214
 swhw_lsr Rm1, Rn1, Rm2, Rn2 ----- 190, 212
 swhw_mov Rm1, Rn1, imm4, Rn2 ----- 191, 206
 swhw_mov Rm1, Rn1, Rm2, Rn2 ----- 190, 204
 swhw_sub Rm1, Rn1, imm4, Rn2 ----- 191, 200
 swhw_sub Rm1, Rn1, Rm2, Rn2 ----- 190, 198
 syscall imm4 ----- 135

T

trap ----- 133

U

udf00 Dm, Dn ----- 222
 udf00 imm16, Dn ----- 222
 udf00 imm32, Dn ----- 222
 udf00 imm8, Dn ----- 222
 udf01 Dm, Dn ----- 223
 udf02 Dm, Dn ----- 225
 udf03 Dm, Dn ----- 226
 udf04 Dm, Dn ----- 227
 udf05 Dm, Dn ----- 228
 udf06 Dm, Dn ----- 229
 udf07 Dm, Dn ----- 230
 udf08 Dm, Dn ----- 231
 udf09 Dm, Dn ----- 232
 udf12 Dm, Dn ----- 233
 udf13 Dm, Dn ----- 234
 udf15 Dm, Dn ----- 235
 udfu01 imm16, Dn ----- 223
 udfu01 imm32, Dn ----- 223
 udfu01 imm8, Dn ----- 223

X

xor imm24, Rn ----- 101
 xor Dm, Dn ----- 100
 xor imm16, Dn ----- 101
 xor imm32, Dn ----- 101
 xor imm32, Rn ----- 101
 xor imm8, Rn ----- 101
 xor Rm, Rn ----- 100
 xor Rm, Rn, Rd ----- 100
 xor_add Rm1, Rn1, imm4, Rn2 ----- 187, 196
 xor_add Rm1, Rn1, Rm2, Rn2 ----- 186, 194
 xor_asl Rm1, Rn1, imm4, Rn2 ----- 187, 218
 xor_asl Rm1, Rn1, Rm2, Rn2 ----- 186, 216

xor_asr Rm1, Rn1, imm4, Rn2 ----- 187, 210
 xor_asr Rm1, Rn1, Rm2, Rn2 ----- 186, 208
 xor_cmp Rm1, Rn1, imm4, Rn2 ----- 187, 203
 xor_cmp Rm1, Rn1, Rm2, Rn2 ----- 186, 202
 xor_lsr Rm1, Rn1, imm4, Rn2 ----- 187, 214
 xor_lsr Rm1, Rn1, Rm2, Rn2 ----- 186, 212
 xor_mov Rm1, Rn1, imm4, Rn2 ----- 187, 206
 xor_mov Rm1, Rn1, Rm2, Rn2 ----- 186, 204
 xor_sub Rm1, Rn1, imm4, Rn2 ----- 187, 200
 xor_sub Rm1, Rn1, Rm2, Rn2 ----- 186, 198

**MN103E Series
Instruction Manual**

March, 2003 2nd Edition

Issued by Matsushita Electric Industrial Co., Ltd.
© Matsushita Electric Industrial Co., Ltd.

Semiconductor Company, Matsushita Electric Industrial Co., Ltd.

Nagaokakyo, Kyoto 617-8520, Japan

Tel: (075) 951-8151

<http://panasonic.co.jp/semicon>

SALES OFFICES

■ NORTH AMERICA

● U.S.A. Sales Office:

Panasonic Industrial Company [PIC]

● New Jersey Office:

Two Panasonic Way Secaucus, New Jersey 07094 U.S.A.

Tel: 1-201-348-5257 Fax: 1-201-392-4652

● Chicago Office:

1707 N. Randall Road Elgin, Illinois 60123-7847 U.S.A.

Tel: 1-847-468-5720 Fax: 1-847-468-5725

● Milpitas Office:

1600 McCandless Drive Milpitas, California 95035 U.S.A.

Tel: 1-408-942-2912 Fax: 1-408-946-9063

● Atlanta Office:

1225 Northbrook Parkway Suite 1-151 Suwanee, GA 30024 U.S.A.

Tel: 1-770-338-6953 Fax: 1-770-338-6849

● San Diego Office:

9444 Balboa Avenue, Suite 185, San Diego, California 92123 U.S.A.

Tel: 1-619-503-2903 Fax: 1-858-715-5545

● Canada Sales Office:

Panasonic Canada Inc. [PCI]

5770 Ambler Drive 27 Mississauga, Ontario, L4W 2T3 CANADA

Tel: 1-905-238-2315 Fax: 1-905-238-2414

■ LATIN AMERICA

● Mexico Sales Office:

Panasonic de Mexico, S.A. de C.V. [PANAMEX]

Amores 1120 Col. Del Valle Delegacion Benito Juarez C.P. 03100 Mexico, D.F. MEXICO

Tel: 52-5-488-1000 Fax: 52-5-488-1073

● Guadalajara Office:

SUCURSAL GUADALAJARA

Av. Lazaro Cardenas 2305 Local G-102 Plaza Comercial Abastos; Col. Las Torres Guadalajara, Jal. 44920 MEXICO

Tel: 52-3-671-1205 Fax: 52-3-671-1256

● Brazil Sales Office:

Panasonic do Brasil Ltda. [PANABRAS]

Caixa Postal 1641, Sao Jose dos Campos, Estado de Sao Paulo

Tel: 55-12-335-9000 Fax: 55-12-331-3789

■ EUROPE

● Europe Sales Office:

Panasonic Industrial Europe GmbH [PIE]

● U.K. Sales Office:

Willoughby Road, Bracknell, Berks., RG12 8FP, THE UNITED KINGDOM

Tel: 44-1344-85-3671 Fax: 44-1344-85-3853

● Germany Sales Office:

Hans-Pinsel-Strasse 2 85540 Haar, GERMANY

Tel: 49-89-46159-119 Fax: 49-89-46159-195

■ ASIA

● Singapore Sales Office:

Panasonic Semiconductor of South Asia [PSSA]

300 Beach Road, #16-01, The Concourse, Singapore 199555 THE REPUBLIC OF SINGAPORE

Tel: 65-6390-3688 Fax: 65-6390-3689

● Malaysia Sales Office:

Panasonic Industrial Company (M) Sdn. Bhd. [PICM]

● Head Office:

Tingkat 16B, Menara PKNS Petaling Jaya, No.17, Jalan Yong Shook Lin 46050 Petaling Jaya, Selangor Darul Ehsan, MALAYSIA

Tel: 60-3-7951-6601 Fax: 60-3-7954-5968

● Penang Office:

Suite 20-07, 20th Floor, MWE Plaza, No.8, Lebuh Farquhar, 10200 Penang, MALAYSIA

Tel: 60-4-201-5113 Fax: 60-4-261-9989

● Johore Sales Office:

Menara Pelangi, Suite 8.3A, Level 8, No.2, Jalan Kuning Taman Pelangi, 80400 Johor Bahru, Johor, MALAYSIA

Tel: 60-7-331-3822 Fax: 60-7-355-3996

● Thailand Sales Office:

Panasonic Industrial (THAILAND) Ltd. [PICT]

252-133 Muang Thai-Phatra Complex Building, 31st Fl. Rachadaphisek Rd., Huaykwang, Bangkok 10320, THAILAND

Tel: 66-2-693-3428 Fax: 66-2-693-3422

● Philippines Sales Office:

[PISP]

Panasonic Industrial Sales Philippines Division of Matsushita Electric Philippines Corporation

102 Laguna Boulevard, Bo. Don Jose Laguna Technopark, Santa Rosa, Laguna 4026 PHILIPPINES

Tel: 63-2-520-8615 Fax: 63-2-520-8629

● India Sales Office:

National Panasonic India Ltd. [NPI]

E Block, 510, International Trade Tower Nehru Place, New Delhi 110019 INDIA

Tel: 91-11-629-2870 Fax: 91-11-629-2877

● Indonesia Sales Office:

P.T.MET & Gobel [M&G]

JL. Dewi Sartika (Cawang 2) Jakarta 13630, INDONESIA

Tel: 62-21-801-5666 Fax: 62-21-801-5675

● China Sales Office:

Panasonic Industrial (Shanghai) Co., Ltd. [PI(SH)]

Floor 12, Zhong Bao Mansion, 166 East Road Lujian Zui, PU Dong New District, Shanghai, 200120 CHINA

Tel: 86-21-5866-6114 Fax: 86-21-5866-8000

Panasonic Industrial (Tianjin) Co., Ltd. [PI(TJ)]

Room No.1001, Tianjin International Building 75, Nanjin Road, Tianjin 300050, CHINA

Tel: 86-22-2313-9771 Fax: 86-22-2313-9770

Panasonic SH Industrial Sales (Shenzhen) Co., Ltd.

[PSI(SZ)]

● Head Office:

7A-107, International Business & Exhibition Centre, Futian Free Trade Zone, Shenzhen 518048, CHINA

Tel: 86-755-8359-8500 Fax: 86-755-8359-8516

● Shum Yip Centre Office:

25F, Shum Yip Centre, #5045, East Shennan Road, Shenzhen, CHINA

Tel: 86-755-8211-0888 Fax: 86-755-8211-0884

Panasonic Shun Hing Industrial Sales (Hong Kong) Co., Ltd. [PSI(HK)]

11th Floor, Great Eagle Center 23 Harbour Road, Wanchai, HONG KONG

Tel: 852-2529-7322 Fax: 852-2865-3697

● Taiwan Sales Office:

Panasonic Industrial Sales (Taiwan) Co., Ltd. [PIST]

● Head Office:

6F, 550, Sec. 4, Chung Hsiao E. RD. Taipei, 110, TAIWAN

Tel: 886-2-2757-1900 Fax: 886-2-2757-1906

● Kaohsiung Office:

6th Floor, Hsin Kong Bldg. No.251, Chi Hsien 1st Road

Kaohsiung 800, TAIWAN

Tel: 886-7-346-3815 Fax: 886-7-236-8362

● Korea Sales Office:

Panasonic Industrial Korea Co., Ltd. [PIKL]

Kukje Center Bldg. 11th Fl., 191 Hangangro 2ga, Youngsan-ku, Seoul 140-702, KOREA

Tel: 82-2-795-9600 Fax: 82-2-795-1542