To all our customers

Regarding the change of names mentioned in the document, such as Mitsubishi Electric and Mitsubishi XX, to Renesas Technology Corp.

The semiconductor operations of Hitachi and Mitsubishi Electric were transferred to Renesas Technology Corporation on April 1st 2003. These operations include microcomputer, logic, analog and discrete devices, and memory chips other than DRAMs (flash memory, SRAMs etc.) Accordingly, although Mitsubishi Electric, Mitsubishi Electric Corporation, Mitsubishi Semiconductors, and other Mitsubishi brand names are mentioned in the document, these names have in fact all been changed to Renesas Technology Corp. Thank you for your understanding. Except for our corporate trademark, logo and corporate statement, no changes whatsoever have been made to the contents of the document, and these changes do not constitute any alteration to the contents of the document itself.

Note : Mitsubishi Electric will continue the business operations of high frequency & optical devices and power devices.

Renesas Technology Corp.
Customer Support Dept.
April 1, 2003

RENESAS
Renesas Technology Corp.

Mitsubishi 32-bit RISC Single-chip Microcomputers

M32R-FPU

Software Manual

http://www.infomicom.maec.co.jp/indexe.htm

Before using this material, please visit the above website to confirm that this is the most current document available.

RENESAS
Renesas Technology Corp.

Rev. 1.0
Revision date: Jan. 8, 2003

Keep safety first in your circuit designs!

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

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Mitsubishi semiconductor product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Mitsubishi Electric Corporation or a third party.
- Mitsubishi Electric Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Mitsubishi Electric Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Mitsubishi Electric Corporation or an authorized Mitsubishi Semiconductor product distributor for the latest product information before purchasing a product listed herein.
  The information described here may contain technical inaccuracies or typographical errors. Mitsubishi Electric Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
  Please also pay attention to information published by Mitsubishi Electric Corporation by various means, including the Mitsubishi Semiconductor home page (http://www.mitsubishichips.com).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Mitsubishi Electric Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Mitsubishi Electric Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Mitsubishi Electric Corporation or an authorized Mitsubishi Semiconductor product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Mitsubishi Electric Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
  Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Mitsubishi Electric Corporation or an authorized Mitsubishi Semiconductor product distributor for further details on these materials or the products contained therein.

REVISION HISTORY	M32R-FPU SOFTWARE MANUAL

Rev.	Date	Description	
		Page	Summary
1.0	Jan. 8, 03	-	First edition issued

Table of contents

CHAPTER 1   CPU PROGRAMMING MODEL

CHAPTER 2   INSTRUCTION SET

CHAPTER 3 INSTRUCTIONS

APPENDIX

INDEX

This page left blank intentionally.

CHAPTER 1

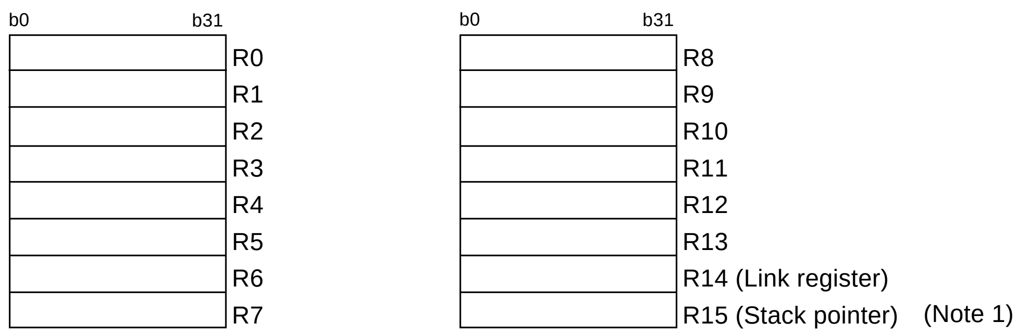
CPU PROGRAMMIING MODEL

1.1 CPU Register

The M32R family CPU, with a built-in FPU (herein referred to as M32R-FPU) has 16 general-purpose registers, 6 control registers, an accumulator and a program counter. The accumulator is of 56-bit configuration, and all other registers are a 32-bit configuration.

1.2 General-purpose Registers

The 16 general-purpose registers (R0 – R15) are of 32-bit width and are used to retain data and base addresses, as well as for integer calculations, floating-point operations, etc. R14 is used as the link register and R15 as the stack pointer. The link register is used to store the return address when executing a subroutine call instruction. The Interrupt Stack Pointer (SPI) and the User Stack Pointer (SPU) are alternately represented by R15 depending on the value of the Stack Mode (SM) bit in the Processor Status Word Register (PSW).
At reset release, the value of the general-purpose registers is undefined.



Note 1: The stack pointer functions as either the SPI or the SPU depending on the value of the SM bit in the PSW.

Figure 1.2.1 General-purpose Registers

1.3 Control Registers

There are 6 control registers which are the Processor Status Word Register (PSW), the Condition Bit Register (CBR), the Interrupt Stack Pointer (SPI), the User Stack Pointer (SPU), the Backup PC (BPC) and the Floating-point Status Register (FPSR). The dedicated **MVTC** and **MVFC** instructions are used for writing and reading these control registers.
In addition, the SM bit, IE bit and C bit of the PSW can also be set by the SETPSW instruction or the CLRPSW instruction.

CRn	b0 – b31	
CR0	**PSW**	**Processor Status Register**
CR1	**CBR**	**Condition Bit Register**
CR2	**SPI**	**Interrupt Stack Pointer**
CR3	**SPU**	**User Stack Pointer**
CR6	**BPC**	**Backup PC**
CR7	**FPSR**	**Floating-point Status Register**

Notes: • CRn (n = 0 - 3, 6 and 7) denotes the control register number.
• The dedicated MVTC and MVFC instructions are used for writing and reading these control registers.
• The SM bit, IE bit and C bit of the PSW can also be set by the SETPSW instruction or the CLRPSW instruction.

Figure 1.3.1 Control Registers

1.3.1 Processor Status Word Register: PSW (CR0)

b0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	b15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

b16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	b31
BSM	BIE						BC	SM	IE						C
?	?	0	0	0	0	0	?	0	0	0	0	0	0	0	0
BPSW field								PSW field							

< At reset release: "B'0000 0000 0000 0000 ??00 000? 0000 0000 >

b	Bit Name	Function	R	W
0-15	No function assigned. Fix to "0".		0	0
16	BSM Backup SM Bit	Saves value of SM bit when EIT occurs	R	W
17	BIE Backup IE Bit	Saves value of IE bit when EIT occurs	R	W
18-22	No function assigned. Fix to "0".		0	0
23	BC Backup C Bit	Saves value of C bit when EIT occurs	R	W
24	SM Stack Mode Bit	0: Uses R15 as the interrupt stack pointer 1: Uses R15 as the user stack pointer	R	W
25	IE Interrupt Enable Bit	0: Does not accept interrupt 1: Accepts interrupt	R	W
26-30	No function assigned. Fix to "0".		0	0
31	C Condition Bit	Indicates carry, borrow and overflow resulting from operations (instruction dependent)	R	W

The Processor Status Word Register (PSW) indicates the M32R-FPU status. It consists of the current PSW field which is regularly used, and the BPSW field where a copy of the PSW field is saved when EIT occurs.

The PSW field consists of the Stack Mode (SM) bit, the Interrupt Enable (IE) bit and the Condition (C) bit.

The BPSW field consists of the Backup Stack Mode (BSM) bit, the Backup Interrupt Enable (BIE) bit and the Backup Condition (BC) bit.

At reset release, BSM, BIE and BC are undefined. All other bits are "0".

1.3.2 Condition Bit Register: CBR (CR1)

The Condition Bit Register (CBR) is derived from the PSW register by extracting its Condition (C) bit. The value written to the PSW register's C bit is reflected in this register. The register can only be read. (Writing to the register with the **MVTC** instruction is ignored.)
At reset release, the value of CBR is "H'0000 0000".

	b0 … b30	b31
CBR	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	C

1.3.3 Interrupt Stack Pointer: SPI (CR2)
User Stack Pointer: SPU (CR3)

The Interrupt Stack Pointer (SPI) and the User Stack Pointer (SPU) retain the address of the current stack pointer. These registers can be accessed as the general-purpose register R15. R15 switches between representing the SPI and SPU depending on the value of the Stack Mode (SM) bit in the PSW.
At reset release, the value of the SPI and SPU are undefined.

	b0 … b31
SPI	SPI

	b0 … b31
SPU	SPU

1.3.4 Backup PC: BPC (CR6)

The Backup PC (BPC) is used to save the value of the Program Counter (PC) when an EIT occurs. Bit 31 is fixed to "0".
When an EIT occurs, the register sets either the PC value when the EIT occurred or the PC value for the next instruction depending on the type of EIT. The BPC value is loaded to the PC when the **RTE** instruction is executed. However, the values of the lower 2 bits of the PC are always "00" when returned (PC always returns to the word-aligned address).
At reset release, the value of the BPC is undefined.

	b0 … b30	b31
BPC	BPC	0

1.3.5 Floating-point Status Register: FPSR (CR7)

b0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	b15
FS 0	FX 0	FU 0	FZ 0	FO 0	FV 0	0	0	0	0	0	0	0	0	0	0

b16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	b31
0	EX 0	EU 0	EZ 0	EO 0	EV 0	0	DN 1	CE 0	CX 0	CU 0	CZ 0	CO 0	CV 0	RM 0	0

<At reset release: H0000 0100>

b	Bit Name	Function	R	W
0	FS Floating-point Exception Summary Bit	Reflects the logical sum of FU, FZ, FO and FV.	R	–
1	FX Inexact Exception Flag	Set to "1" when an inexact exception occurs (if EIT processing is unexecuted (Note 1)). Once set, the flag retains the value "1" until it is cleared to "0" in software.	R	W
2	FU Underflow Exception Flag	Set to "1" when an underflow exception occurs (if EIT processing is unexecuted (Note 1)). Once set, the flag retains the value "1" until it is cleared to "0" in software.	R	W
3	FZ Zero Divide Exception Flag	Set to "1" when a zero divide exception occurs (if EIT processing is unexecuted (Note 1)). Once set, the flag retains the value "1" until it is cleared to "0" in software.	R	W
4	FO Overflow Exception Flag	Set to "1" when an overflow exception occurs (if EIT processing is unexecuted (Note 1)). Once set, the flag retains the value "1" until it is cleared to "0" in software.	R	W
5	FV Invalid Operation Exception Flag	Set to "1" when an invalid operation exception occurs (if EIT processing is unexecuted (Note 1)). Once set, the flag retains the value "1" until it is cleared to "0" in software.	R	W
6–16	No function assigned. Fix to "0".		0	0
17	EX Inexact Exception Enable Bit	0: Mask EIT processing to be executed when an inexact exception occurs 1: Execute EIT processing when an inexact exception occurs	R	W
18	EU Underflow Exception Enable Bit	0: Mask EIT processing to be executed when an underflow exception occurs 1: Execute EIT processing when an underflow exception occurs	R	W
19	EZ Zero Divide Exception Enable Bit	0: Mask EIT processing to be executed when a zero divide exception occurs 1: Execute EIT processing when a zero divide exception occurs	R	W
20	EO Overflow Exception Enable Bit	0: Mask EIT processing to be executed when an overflow exception occurs 1: Execute EIT processing when an overflow exception occurs	R	W

21	EV Invalid Operation Exception Enable Bit	0: Mask EIT processing to be executed when an invalid operation exception occurs 1: Execute EIT processing when an invalid operation exception occurs	R W
22	No function assigned. Fix to "0".		0 0
23	DN Denormalized Number Zero Flash Bit (Note 2)	0: Handle the denormalized number as a denormalized number 1: Handle the denormalized number as zero	R W
24	CE Unimplemented Operation Exception Cause Bit	0: No unimplemented operation exception occurred . 1: An unimplemented operation exception occurred. When the bit is set to "1", the execution of an FPU operation instruction will clear it to "0".	R (Note 3)
25	CX Inexact Exception Cause Bit	0: No inexact exception occurred. 1: An inexact exception occurred. When the bit is set to "1", the execution of an FPU operation instruction will clear it to "0".	R (Note 3)
26	CU Underflow Exception Cause Bit	0: No underflow exception occurred. 1: An underflow exception occurred. When the bit is set to "1", the execution of an FPU operation instruction will clear it to "0".	R (Note 3)
27	CZ Zero Divide Exception Cause Bit	0: No zero divide exception occurred. 1: A zero divide exception occurred. When the bit is set to "1", the execution of an FPU operation instruction will clear it to "0".	R (Note 3)
28	CO Overflow Exception Cause Bit	0: No overflow exception occurred. 1: An overflow exception occurred. When the bit is set to "1", the execution of an FPU operation instruction will clear it to "0".	R (Note 3)
29	CV Invalid Operation Exception Cause Bit	0: No invalid operation exception occurred. 1: An invalid operation exception occurred. When the bit is set to "1", the execution of an FPU operation instruction will clear it to "0".	R (Note 3)
30, 31	RM Rounding Mode Selection Bit	00: Round to Nearest 01: Round toward Zero 10: Round toward +Infinity 11: Round toward -Infinity	R W

Note 1: ‘If EIT processing is unexecuted’ means whenever one of the exceptions occurs, enable bits 17 to 21 are set to "0" which masks the EIT processing so that it cannot be executed. If two exceptions occur at the same time and their corresponding exception enable bits are set differently (one enabled, and the other masked), EIT processing is executed. In this case, these two flags do not change state regardless of the enable bit settings.

Note 2: If a denormalized number is given to the operand when DN = "0", an unimplemented exception occurs.

Note 3: This bit is cleared by writing "0". Writing "1" has no effect (the bit retains the value it had before the write).

1.3.6 Floating-point Exceptions (FPE)

Floating-point Exception (FPE) occurs when Unimplemented Exception (UIPL) or one of the five exceptions specified in the IEEE754 standard (OVF/UDF/IXCT/ DIV0/IVLD) is detected. Each exception processing is outlined below.

(1) Overflow Exception (OVF)

The exception occurs when the absolute value of the operation result exceeds the largest describable precision in the floating-point format. The following table shows the operation results when an OVF occurs.

Rounding Mode	Sign of the Result	Operation Result (Content of the Destination Register)	
		When the OVF EIT processing is masked (Note 1)	When the OVF EIT processing is executed (Note 2)
–infinity	+	+MAX	No change
	–	–infinity	
+infinity	+	+infinity	
	–	–MAX	
0	+	+MAX	
	–	–MAX	
Nearest	+	+infinity	
	–	–infinity	

Note 1: When the Overflow Exception Enable (EO) bit (FPSR register bit 20) = "0"
Note 2: When the Overflow Exception Enable (EO) bit (FPSR register bit 20) = "1"
Note: • If an OVF occurs while EIT processing for OVF is masked, an IXCT occurs at the same time.
• +MAX = H'7F7F FFFF, –MAX = H'FF7F FFFF

(2) Underflow Exception (UDF)

The exception occurs when the absolute value of the operation result is less than the largest describable precision in the floating-point format. The following table shows the operation results when a UDF occurs.

Operation Result (Content of the Destination Register)	
When UDF EIT processing is masked (Note 1)	When UDF EIT processing is executed (Note 2)
DN = 0: An unimplemented exception occurs DN = 1: 0 is returned	No change

Note 1: When the Underflow Exception Enable (EU) bit (FPSR register bit 18) = "0"
Note 2: When the Underflow Exception Enable (EU) bit (FPSR register bit 18) = "1"

(3) Inexact Exception (IXCT)

The exception occurs when the operation result differs from a result led out with an infinite range of precision. The following table shows the operation results and the respective conditions in which each IXCT occurs.

Occurrence Condition	Operation Result (Content of the Destination Register)	
	When the IXCT EIT processing is masked (Note 1)	When the IXCT EIT processing is executed (Note 2)
Overflow occurs in OVF masked condition	Reference OVF operation results	No change
Rounding occurs	Rounded value	No change

Note 1: When the Inexact Exception Enable (EX) bit (FPSR register bit 17) = "0"
Note 2: When the Inexact Exception Enable (EX) bit (FPSR register bit 17) = "1"

(4) Zero Division Exception (DIV0)

The exception occurs when a finite nonzero value is divided by zero. The following table shows the operation results when a DIV0 occurs.

Dividend	Operation Result (Content of the Destination Register)	
	When the DIV0 EIT processing is masked (Note 1)	When the DIV0 EIT processing is executed (Note 2)
Nonzero finite value	±infinity (Sign is derived by exclusive-ORing the signs of divisor and dividend)	No change

Note 1: When the Zero Division Exception Enable (EZ) bit (FPSR register bit 19) = "0"
Note 2: When the Zero Division Exception Enable (EZ) bit (FPSR register bit 19) = "1"

Please note that the DIV0 EIT processing does not occur in the following conditions.

Dividend	Behavior
0	An invalid operation exception occurs
infinity	No exception occur (with the result "infinity")

(5) Invalid Operation Exception (IVLD)

The exception occurs when an invalid operation is executed. The following table shows the operation results and the respective conditions in which each IVLD occurs.

<table>
<tr><td colspan="2" rowspan="2">Occurrence Condition</td><td colspan="2">Operation Result (Content of the Destination Register)</td></tr>
<tr><td>When the IVLD EIT processing is masked (Note 1)</td><td>When the IVLD EIT processing is executed (Note 2)</td></tr>
<tr><td colspan="2">Operation for SNaN operand</td><td rowspan="4">QNaN</td><td rowspan="7">No change</td></tr>
<tr><td colspan="2">+infinity -(+infinity), -infinity -(-infinity)</td></tr>
<tr><td colspan="2">0 × infinity</td></tr>
<tr><td colspan="2">0 ÷ 0, infinity ÷ infinity</td></tr>
<tr><td rowspan="2">When an integer conversion overflowed
When NaN or Infinity was converted into an integer</td><td>When FTOI instruction was executed</td><td>Return value when pre-conversion signed bit is:
"0" = H’7FFF FFFF
"1" = H’8000 0000</td></tr>
<tr><td>When FTOS instruction was executed</td><td>Return value when pre-conversion signed bit is:
"0" = H’0000 7FFF
"1" = H’FFF 8000</td></tr>
<tr><td colspan="2">When < or > comparison was performed on NaN</td><td>Comparison results (comparison invalid)</td></tr>
</table>

Note 1: When the Invalid Operation Exception Enable (EV) bit (FPSR register bit 21) = "0"
Note 2: When the Invalid Operation Exception Enable (EV) bit (FPSR register bit 21) = "1"
Notes: • NaN (Not a Number)

SNaN (Signaling NaN): a NaN in which the MSB of the decimal fraction is “0”. When SNaN is used as the source operand in an operation, an IVLD occurs. SNaNs are useful in identifying program bugs when used as the initial value in a variable. However, SNaNs cannot be generated by hardware.

QNaN (Quiet NaN): a NaN in which the MSB of the decimal fraction is "1". Even when QNaN is used as the source operand in an operation, an IVLD will not occur (excluding comparison and format conversion). Because a result can be checked by the arithmetic operations, QNaN allows the user to debug without executing an EIT processing. QNaNs are created by hardware.

(6) Unimplemented Exception (UIPL)

The exception occurs when the Denormalized Number Zero Flash (DN) bit (FPSR register bit 23) = "0" and a denormalized number is given as an operation operand (Note 1).
Because the UIPL has no enable bits available, it cannot be masked when they occur. The destination register remains unchanged.

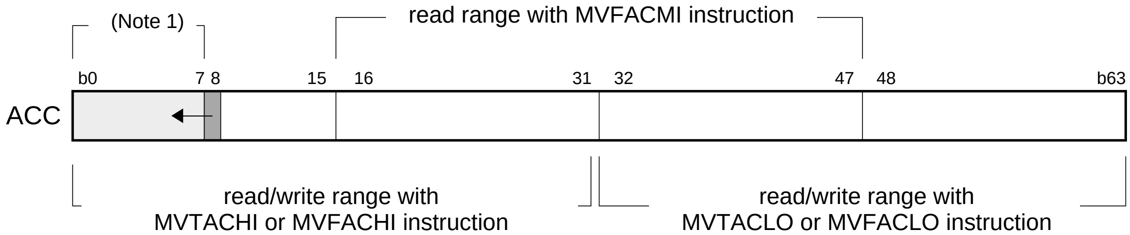
Note: • A UDF occurs when the intermediate result of an operation is a denormalized number, in which case if the DN bit (FPSR register bit 23) = "0", an UIPL occurs.

1.4 Accumulator

The Accumulator (ACC) is a 56-bit register used for DSP function instructions.
The accumulator is handled as a 64-bit register when accessed for read or write. When reading data from the accumulator, the value of bit 8 is sign-extended. When writing data to the accumulator, bits 0 to 7 are ignored. The accumulator is also used for the multiply instruction "MUL", in which case the accumulator value is destroyed by instruction execution.

Use the MVTACHI and MVTACLO instructions for writing to the accumulator. The MVTACHI and MVTACLO instructions write data to the high-order 32 bits (bits 0-31) and the low-order 32 bits (bits 32-63), respectively.

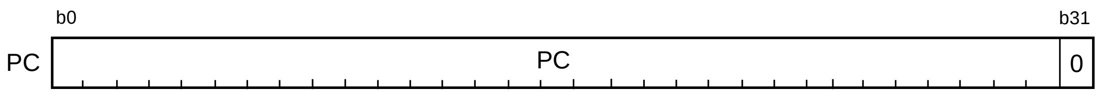
Use the MVFACHI, MVFACLO, and MVFACMI instructions for reading data from the accumulator. The MVFACHI, MVFACLO and MVFACMI instructions read data from the high-order 32 bits (bits 0-31), the low-order 32 bits (bits 32-63) and the middle 32 bits (bits 16-47), respectively.
At reset release, the value of accumulator is undefined.



Note 1: When read, bits 0 to 7 always show the sign-extended value of bit 8. Writing to this bit field is ignored.

1.5 Program Counter

The Program Counter (PC) is a 32-bit counter that retains the address of the instruction being executed. Since the M32R CPU instruction starts with even-numbered addresses, the LSB (bit 31) is always "0".
At reset release, the value of the PC is "H'0000 0000."



1.6 Data Format

1.6.1 Data Type

The data types that can be handled by the M32R-FPU instruction set are signed or unsigned 8, 16, and 32-bit integers and single-precision floating-point numbers. The signed integers are represented by 2's complements.

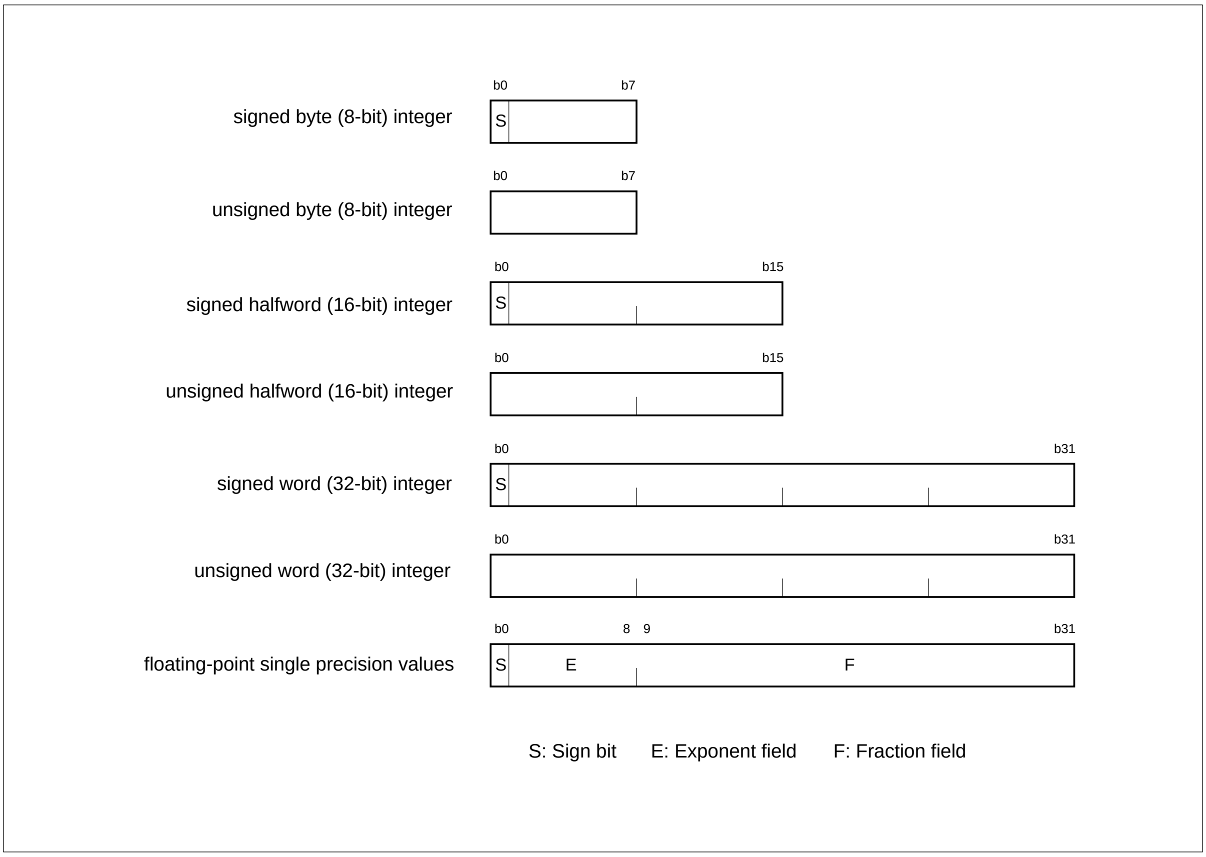


Figure 1.6.1 Data Type

1.6.2 Data Format

(1) Data format in a register

The data sizes in the M32R-FPU registers are always words (32 bits).
When loading byte (8-bit) or halfword (16-bit) data from memory into a register, the data is sign-extended (**LDB**, **LDH** instructions) or zero-extended (**LDUB**, **LDUH** instructions) to a word (32-bit) quantity before being loaded into the register.
When storing data from a register into a memory, the 32-bit data, the 16-bit data on the LSB side and the 8-bit data on the LSB side of the register are stored into memory by the **ST**, **STH** and **STB** instructions, respectively.

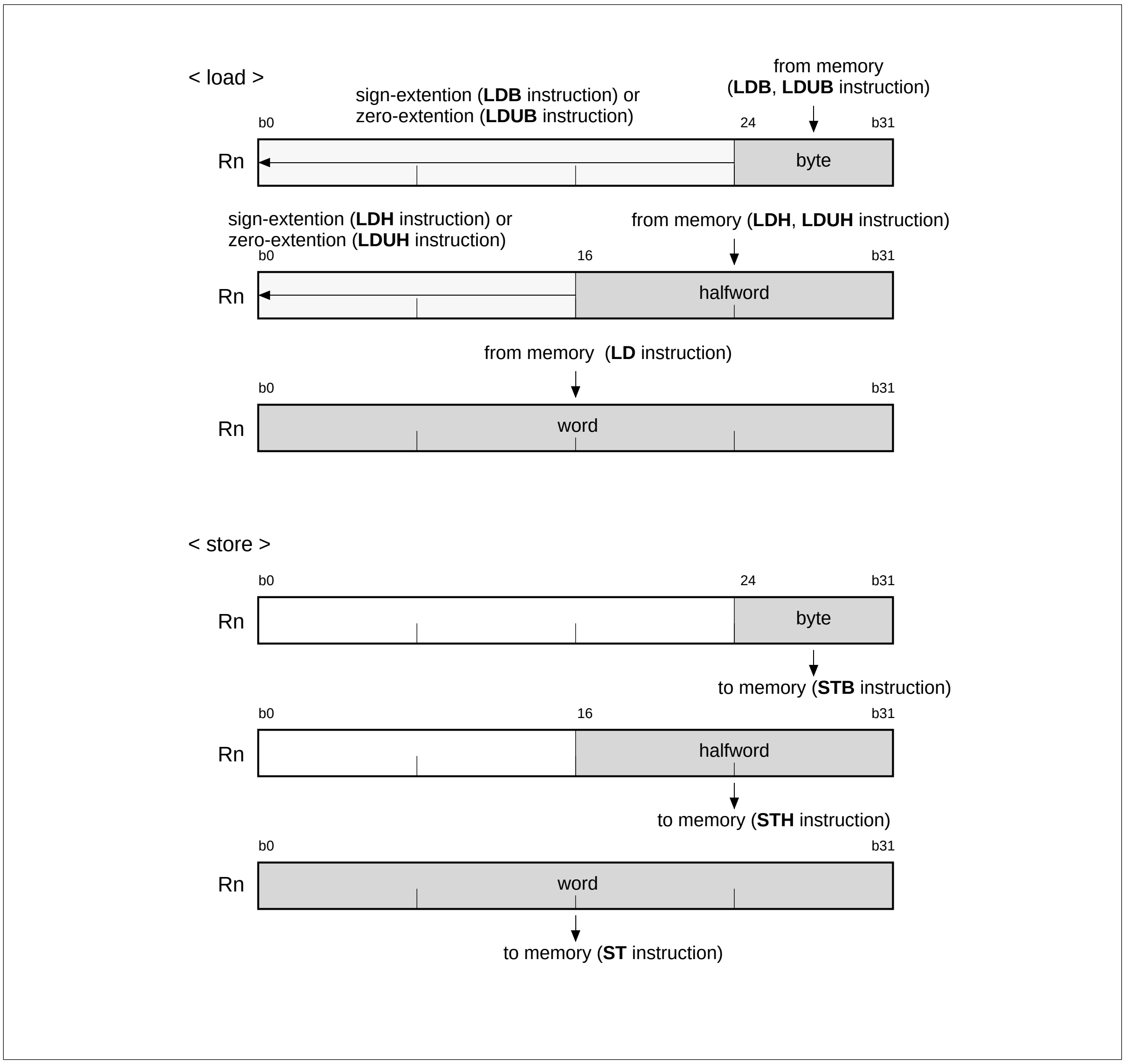


Figure 1.6.2 Data Format in a Register

(2) Data format in memory

The data sizes in memory can be byte (8 bits), halfword (16 bits) or word (32 bits). Although byte data can be located at any address, halfword and word data must be located at the addresses aligned with a halfword boundary (least significant address bit = "0") or a word boundary (two low-order address bits = "00"), respectively. If an attempt is made to access memory data that overlaps the halfword or word boundary, an address exception occurs.

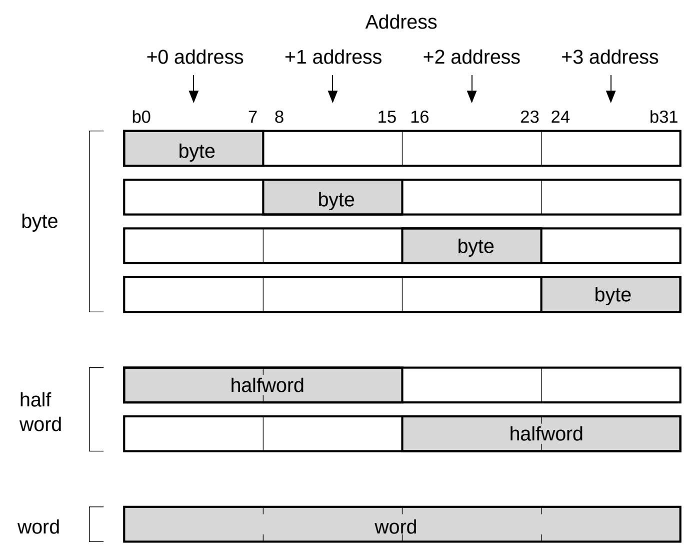


Figure 1.6.3 Data Format in Memory

1.7 Addressing Mode

M32R-FPU supports the following addressing modes.

(1) Register direct **[R or CR]**

The general-purpose register or the control register to be processed is specified.

(2) Register indirect **[@R]**

The contents of the register specify the address of the memory. This mode can be used by all load/store instructions.

(3) Register relative indirect **[@(disp, R)]**

(The contents of the register) + (16-bit immediate value which is sign-extended to 32 bits) specify the address of the memory.

(4) Register indirect and register update

- Adds 4 to register contents **[@R+]**
  The contents of the register specify the memory address, then 4 is added to the register contents.
  (Can only be specified with LD instruction).
- Add 2 to register contents **[@R+] [M32R-FPU extended addressing mode]**
  The contents of the register specify the memory address, then 2 is added to the register contents.
  (Can only be specified with STH instruction).
- Add 4 to register contents **[@+R]**
  The contents of the register is added by 4, the register contents specify the memory address.
  (Can only be specified with ST instruction).
- Subtract 4 to register contents **[@–R]**
  The content of the register is decreased by 4, then the register contents specify the memory address.
  (Can only be specified with ST instruction).

(5) immediate **[#imm]**

The 4-, 5-, 8-, 16- or 24-bit immediate value.

(6) PC relative **[pcdisp]**

(The contents of PC) + (8, 16, or 24-bit displacement which is sign-extended to 32 bits and 2 bits left-shifted) specify the address of memory.

This page left blank intentionally.

CHAPTER 2

INSTRUCTION SET

2.1 Instruction set overview

The M32R-FPU has a total of 100 instructions. The M32R-FPU has a RISC architecture. Memory is accessed by using the load/store instructions and other operations are executed by using register-to-register operation instructions.
M32R CPU supports compound instructions such as " load & address update" and "store & address update" which are useful for high-speed data transfer.

2.1.1 Load/store instructions

The load/store instructions carry out data transfers between a register and a memory.

LD	Load
LDB	Load byte
LDUB	Load unsigned byte
LDH	Load halfword
LDUH	Load unsigned halfword
LOCK	Load locked
ST	Store
STB	Store byte
STH	Store halfword
UNLOCK	Store unlocked

Three types of addressing modes can be specified for load/store instructions.

(1) Register indirect

The contents of the register specify the address. This mode can be used by all load/ store instructions.

(2) Register relative indirect

(The contents of the register) + (32-bit sign-extended 16-bit immediate value) specifies the address. This mode can be used by all except **LOCK** and **UNLOCK** instructions.

(3) Register indirect and register update

- Adds 4 to register contents **[@R+]**
  The contents of the register specify the memory address, then 4 is added to the register contents.
  (Can only be specified with LD instruction).
- Add 2 to register contents **[@R+] [M32R-FPU extended addressing mode]**
  The contents of the register specify the memory address, then 2 is added to the register contents.
  (Can only be specified with STH instruction).
- Add 4 to register contents **[@+R]**
  The contents of the register is added by 4, the register contents specity the memory address.
  (Can only be specified with ST instruction).
- Subtract 4 to register contents **[@–R]**
  The content of the register is decreased by 4, then the register contents specify the memory address.
  (Can only be specified with ST instruction).

When accessing halfword and word size data, it is necessary to specify the address on the halfword boundary or the word boundary (Halfword size should be such that the low-order 2 bits of the address are "00" or "10", and word size should be such that the low order 2 bits of the address are "00"). If an unaligned address is specified, an address exception occurs.
When accessing byte data or halfword data with load instructions, the high-order bits are sign-extended or zero-extended to 32 bits, and loaded to a register.

2.1.2 Transfer instructions

The transfer instructions carry out data transfers between registers or a register and an immediate value.

LD24	Load 24-bit immediate
LDI	Load immediate
MV	Move register
MVFC	Move from control register
MVTC	Move to control register
SETH	Set high-order 16-bit

2.1.3 Operation instructions

Compare, arithmetic/logic operation, multiply and divide, and shift are carried out between registers.

• compare instructions

CMP	Compare
CMPI	Compare immediate
CMPU	Compare unsigned
CMPUI	Compare unsigned immediate

• arithmetic operation instructions

ADD	Add
ADD3	Add 3-operand
ADDI	Add immediate
ADDV	Add with overflow checking
ADDV3	Add 3-operand with overflow checking
ADDX	Add with carry
NEG	Negate
SUB	Subtract
SUBV	Subtract with overflow checking
SUBX	Subtract with borrow

• logic operation instructions

AND	AND
AND3	AND 3-operand
NOT	Logical NOT
OR	OR
OR3	OR 3-operand
XOR	Exclusive OR
XOR3	Exclusive OR 3-operand

• multiply/divide instructions

DIV	Divide
DIVU	Divide unsigned
MUL	Multiply
REM	Remainder
REMU	Remainder unsigned

• shift instructions

SLL	Shift left logical
SLL3	Shift left logical 3-operand
SLLI	Shift left logical immediate
SRA	Shift right arithmetic
SRA3	Shift right arithmetic 3-operand
SRAI	Shift right arithmetic immediate
SRL	Shift right logical
SRL3	Shift right logical 3-operand
SRLI	Shift right logical immediate

2.1.4 Branch instructions

The branch instructions are used to change the program flow.

BC	Branch on C-bit
BEQ	Branch on equal to
BEQZ	Branch on equal to zero
BGEZ	Branch on greater than or equal to zero
BGTZ	Branch on greater than zero
BL	Branch and link
BLEZ	Branch on less than or equal to zero
BLTZ	Branch on less than zero
BNC	Branch on not C-bit
BNE	Branch on not equal to
BNEZ	Branch on not equal to zero
BRA	Branch
JL	Jump and link
JMP	Jump
NOP	No operation

Only a word-aligned (word boundary) address can be specified for the branch address.

The addressing mode of the **BRA**, **BL**, **BC** and **BNC** instructions can specify an 8-bit or 24-bit immediate value. The addressing mode of the **BEQ**, **BNE**, **BEQZ**, **BNEZ**, **BLTZ**, **BGEZ**, **BLEZ**, and **BGTZ** instructions can specify a 16-bit immediate value.
In the **JMP** and **JL** instructions, the register value becomes the branch address. However, the low-order 2-bit value of the register is ignored. In other branch instructions, (PC value of branch instruction) + (sign-extended and 2 bits left-shifted immediate value) becomes the branch address. However, the low order 2-bit value of the address becomes "00" when addition is carried out. For example, refer to **Figure 2.1.1**. When instruction A or B is a branch instruction, branching to instruction G, the immediate value of either instruction A or B becomes 4.
Simultaneous with execution of branching by the **JL** or **BL** instructions for subroutine calls, the PC value of the return address is stored in R14. The low-order 2-bit value of the address stored in R14 (PC value of the branch instruction + 4) is always cleared to "0". For example, refer to **Figure 2.1.1**. If an instruction A or B is a **JL** or **BL** instruction, the return address becomes that of the instruction C.

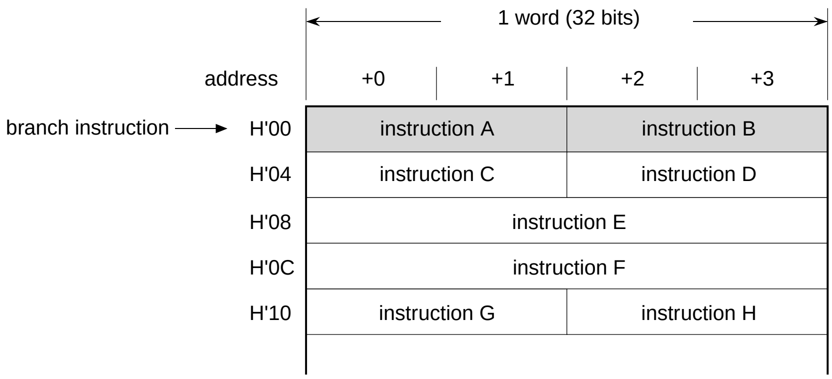


Fig. 2.1.1 Branch addresses of branch instruction

2.1.5 EIT-related instructions

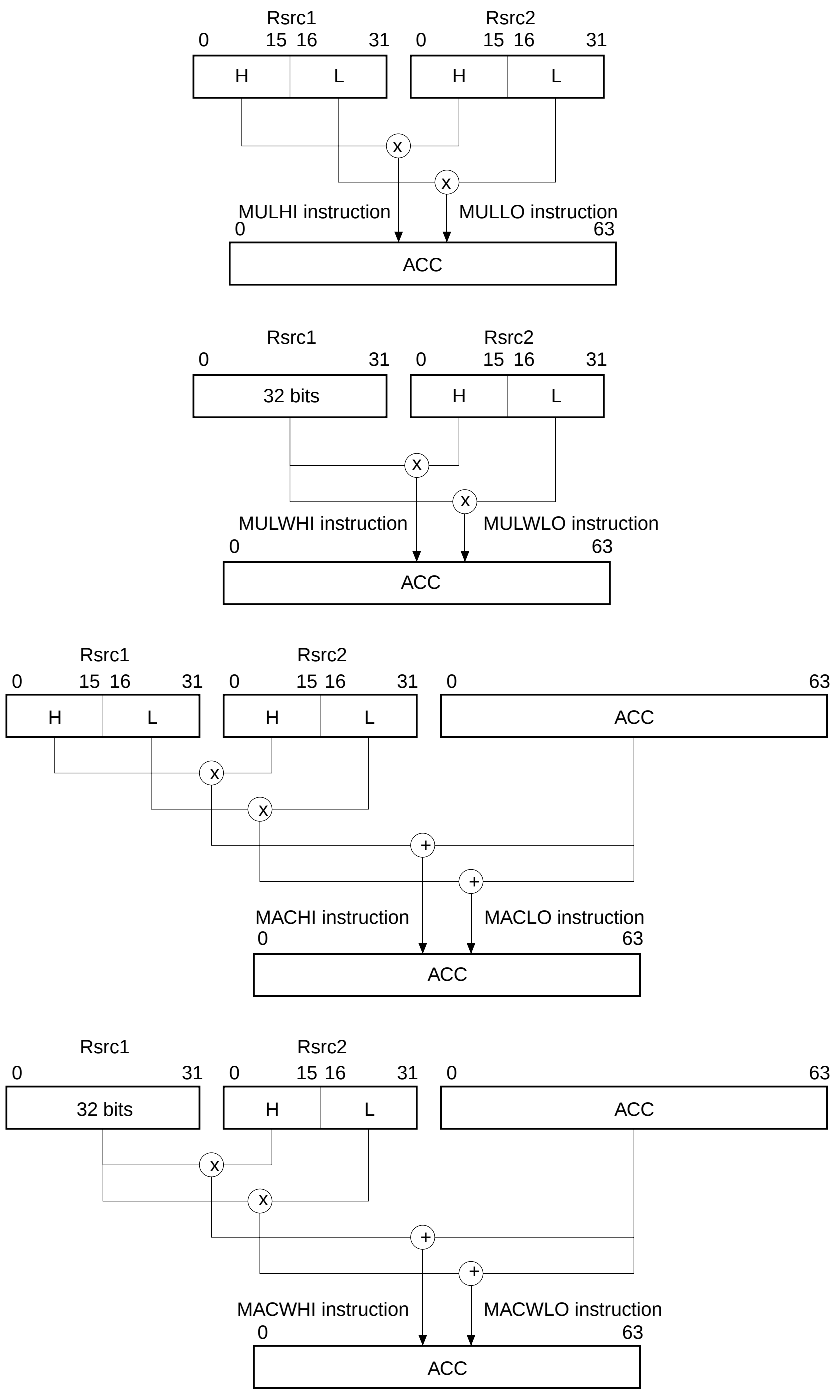
The EIT-related instructions carry out the EIT events (Exception, Interrupt and Trap). Trap initiation and return from EIT are EIT-related instructions.

TRAP	Trap
RTE	Return from EIT

2.1.6 DSP function instructions

The DSP function instructions carry out multiplication of 32 bits x 16 bits and 16 bits x 16 bits or multiply and add operation; there are also instructions to round off data in the accumulator and carry out transfer of data between the accumulator and a general-purpose register.

MACHI	Multiply-accumulate high-order halfwords
MACLO	Multiply-accumulate low-order halfwords
MACWHI	Multiply-accumulate word and high-order halfword
MACWLO	Multiply-accumulate word and low-order halfword
MULHI	Multiply high-order halfwords
MULLO	Multiply low-order halfwords
MULWHI	Multiply word and high-order halfword
MULWLO	Multiply word and low-order halfword
MVFACHI	Move high-order word from accumulator
MVFACLO	Move low-order word from accumulator
MVFACMI	Move middle-order word from accumulator
MVTACHI	Move high-order word to accumulator
MVTACLO	Move low-order word to accumulator
RAC	Round accumulator
RACH	Round accumulator halfword



Note: The location in the accumulator of the result and the appropriate sign extension are performed in the execution of the DSP function instruction. Refer to **Chapter 3** for details.

Fig. 2.1.2 DSP function instruction operation 1 (multiply, multiply and accumulate)

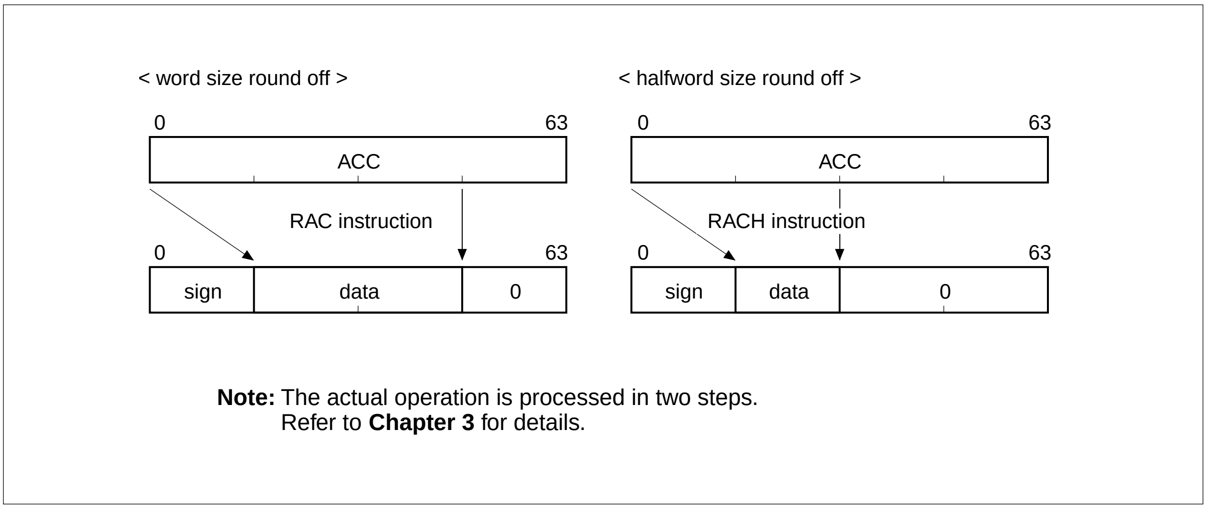


Fig. 2.1.3 DSP function instruction operation 2 (round off)

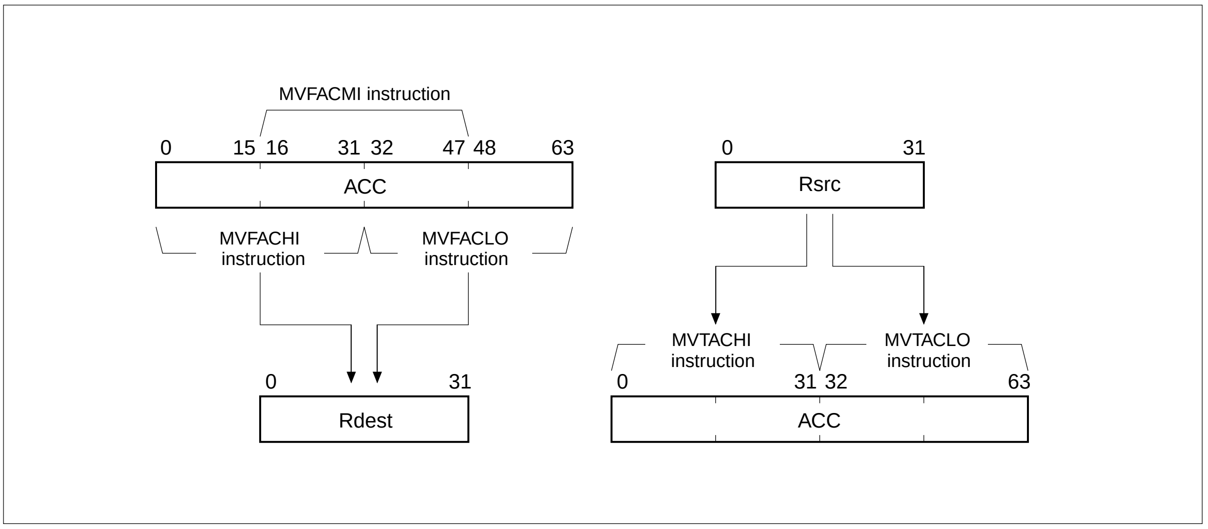


Fig. 2.1.4 DSP function instruction operation 3 (transfer between accumulator and register)

2.1.7 Floating-point Instructions

The following instructions execute floating-point operations.

FADD	Floating-point add
FSUB	Floating-point subtract
FMUL	Floating-point multiply
FDIV	Floating-point divede
FMADD	Floating-point multiply and add
FMSUB	Floating-point multiply and subtract
ITOF	Integer to float
UTOF	Unsigned integer to float
FTOI	Float to integer
FTOS	Float to short
FCMP	Floating-point compare
FCMPE	Floating-point compare with exeption if unordered

2.1.8 Bit Operation Instructions

These instructions determine the operation of the bit specified by the register or memory.

BSET	Bit set
BCLR	Bit clear
BTST	Bit test
SETPSW	Set PSW
CLRPSW	Clear PSW

2.2 Instruction format

There are two major instruction formats: two 16-bit instructions packed together within a word boundary, and a single 32-bit instruction (see **Figure 2.2.1**). Figure 2.2.2 shows the instruction format of M32R CPU.

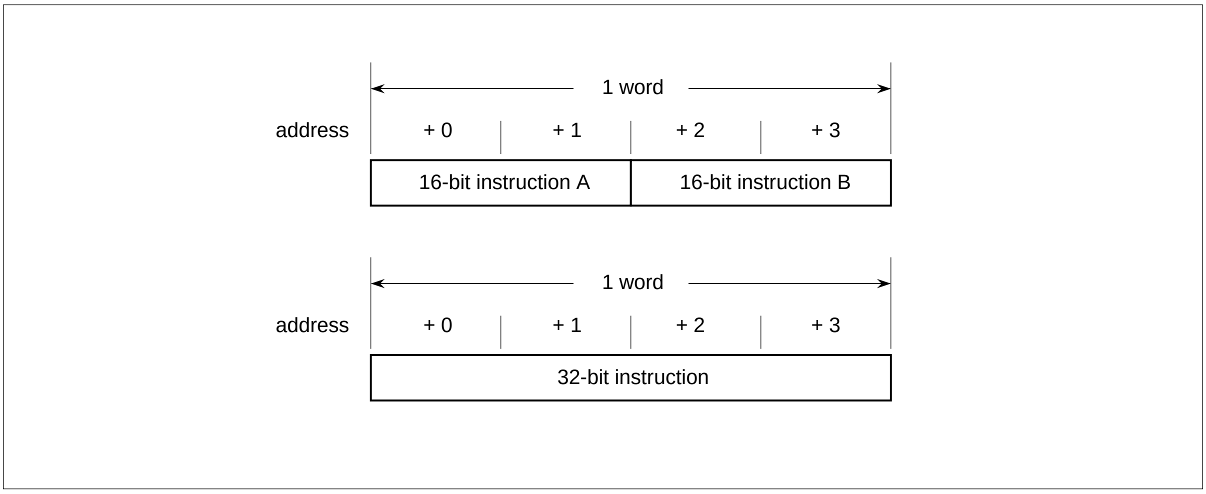


Fig. 2.2.1 16-bit instruction and 32-bit instruction

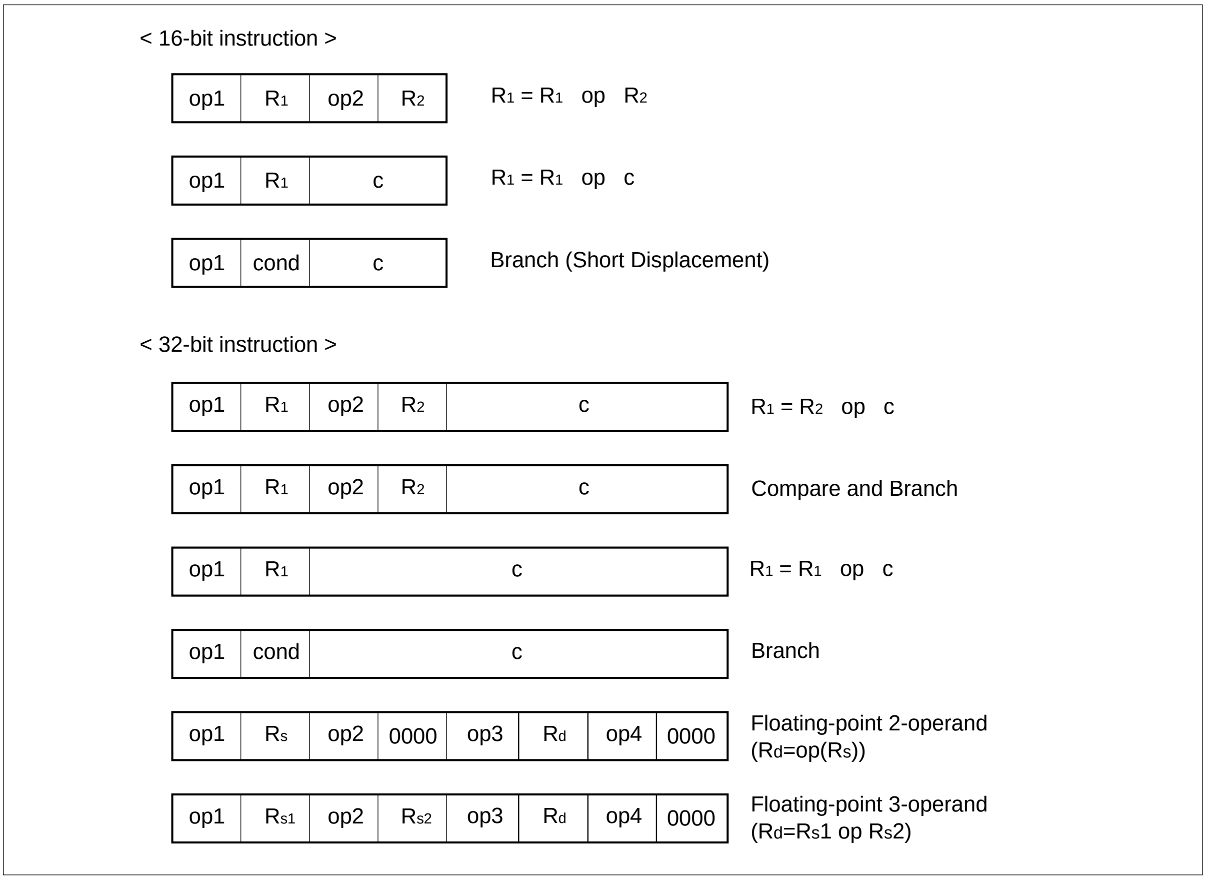


Fig. 2.2.2 Instruction format of M32R CPU

The MSB (Most Significant Bit) of a 32-bit instruction is always "1". The MSB of a 16-bit instruction in the high-order halfword is always "0" (instruction A in Figure 2.2.3), however the processing of the following 16-bit instruction depends on the MSB of the instruction.
In Figure 2.2.3, if the MSB of the instruction B is "0", instructions A and B are executed sequentially; B is executed after A. If the MSB of the instruction B is "1", instructions A and B are executed in parallel.
The current implementation allows only the NOP instruction as instruction B for parallel execution. The MSB of the NOP instruction used for word arraignment adjustment is changed to "1" automatically by a standard Mitsubishi assembler, then the M32R-FPU can execute this instruction without requiring any clock cycles.

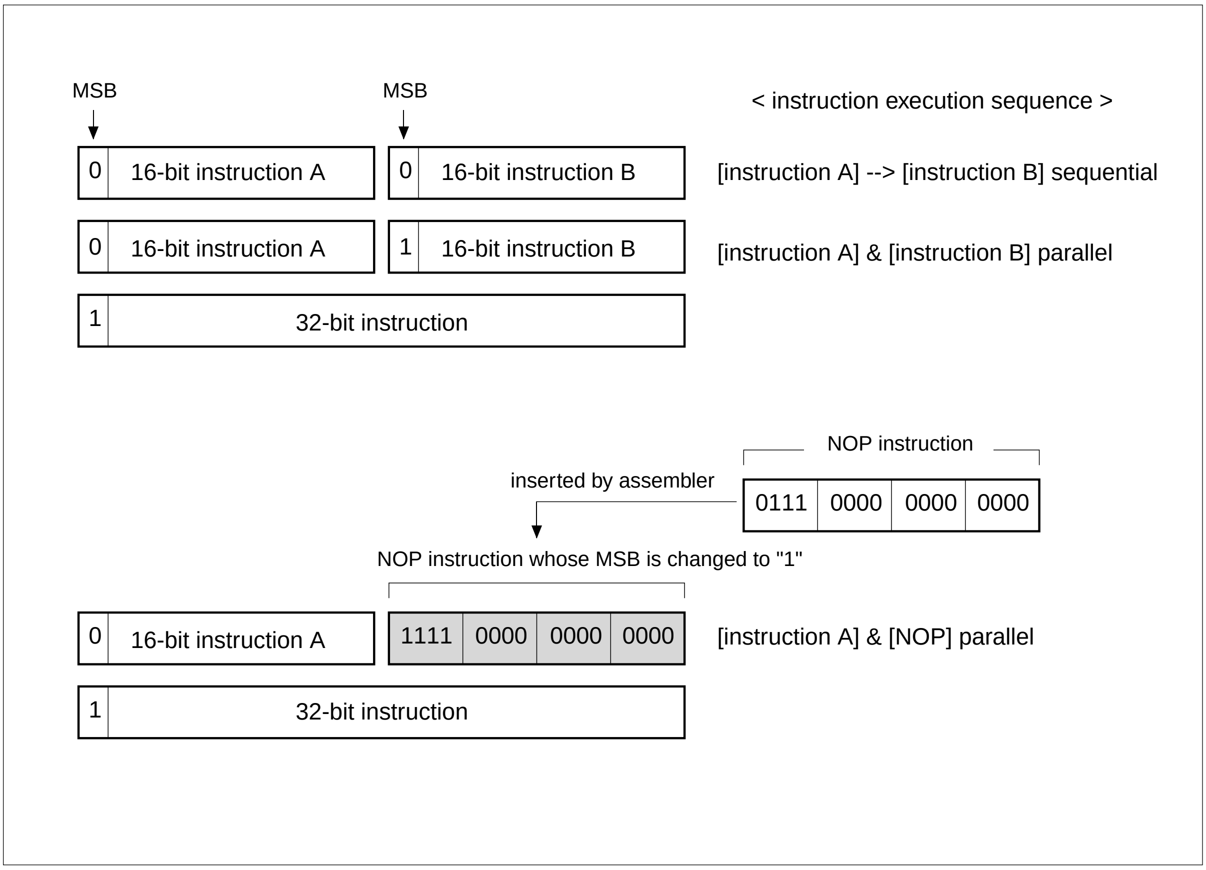


Fig. 2.2.3 Processing of 16-bit instructions

This page left blank intentionally.

CHAPTER 3

INSTRUCTIONS

3.1 Conventions for instruction description

Conventions for instruction description are summarized below.

[Mnemonic]

Shows the mnemonic and possible operands (operation target) using assembly language notation.

Table 3.1.1 Operand list

symbol(see note)	addressing mode	operation target
R	register direct	general-purpose registers (R0 - R15)
CR	control register	Mcontrol registers (CR0 = PSW, CR1 = CBR, CR2 = SPI, CR3 = SPU, CR6 = BPC, CR7 = FPSR)
@R	register indirect	memory specified by register contents as address
@(disp,R)	register relative indirect	memory specified by (register contents) + (sign-extended value of 16-bit displacement) as address
@R+	register indirect and register update	Add 4 to register contents. (Register contents specify the memory address, then 4 is added to the contents.)
@+R	register indirect and register update	Add 4 to register contents. (4 is added to the register contents, then the register contents specify the memory address.)
@-R	register indirect and register update	Subtract 4 to register contents. (4 is subtract to the register contents, hen the register contents specify the memory address.)
#imm	immediate	immediate value (refer to each instruction description)
#bitpos	Bit position	Contents of byte data bit position
pcdisp	PC relative	memory specified by (PC contents) + (8, 16, or 24-bit displacement which is sign-extended to 32 bits and 2 bits left-shifted) as address

Note: When expressing Rsrc or Rdest as an operand, a general-purpose register numbers (0 - 15) should be substituted for src or dest. When expressing CRsrc or CRdest, control register numbers (0 - 3, 6, 7) should be substituted for src or dest.

[Function]

Indicates the operation performed by one instruction. Notation is in accordance with C language notation.

Table 3.1.2 Operation expression (operator)

operator	meaning
+	addition (binomial operator)
-	subtraction (binomial operator)
*	multiplication (binomial operator)
/	division (binomial operator)
%	remainder operation (binomial operator)
++	increment (monomial operator)
--	decrement (monomial operator)

Table 3.1.3 Operation expression (operator) (cont.)

operator	meaning
-	sign invert (monomial operator)
=	substitute right side into left side (substitute operator)
+=	adds right and left variables and substitute into left side (substitute operator)
-=	subtract right variable from left variable and substitute into left side (substitute operator)
>	greater than (relational operator)
<	less than (relational operator)
>=	greater than or equal to (relational operator)
<=	less than or equal to (relational operator)
==	equal (relational operator)
!=	not equal (relational operator)
&&	AND (logical operator)
||	OR (logical operator)
!	NOT (logical operator)
?:	execute a conditional expression (conditional operator)

Table 3.1.4 Operation expression (bit operator)

operator	meaning
<<	bits are left-shifted
>>	bits are right-shifted
&	bit product (AND)
|	bit sum (OR)
^	bit exclusive or (EXOR)
~	bit invert

Table 3.1.5 Data type

expression	sign	bit length	range
signed char	yes	8	–128 to +127
signed short	yes	16	–32,768 to +32,767
signed int	yes	32	–2,147,483,648 to +2,147,483,647
unsigned char	no	8	0 to 255
unsigned short	no	16	0 to 655,535
unsigned int	no	32	0 to 4,294,967,295
signed64bit	yes	64	signed 64-bit integer (with accumulator)

Table 3.1.6 Data type (floating-point)

expression	floating-point format
float	single precision values format

[Description]

Describes the operation performed by the instruction and any condition bit change.

[EIT occurrence]

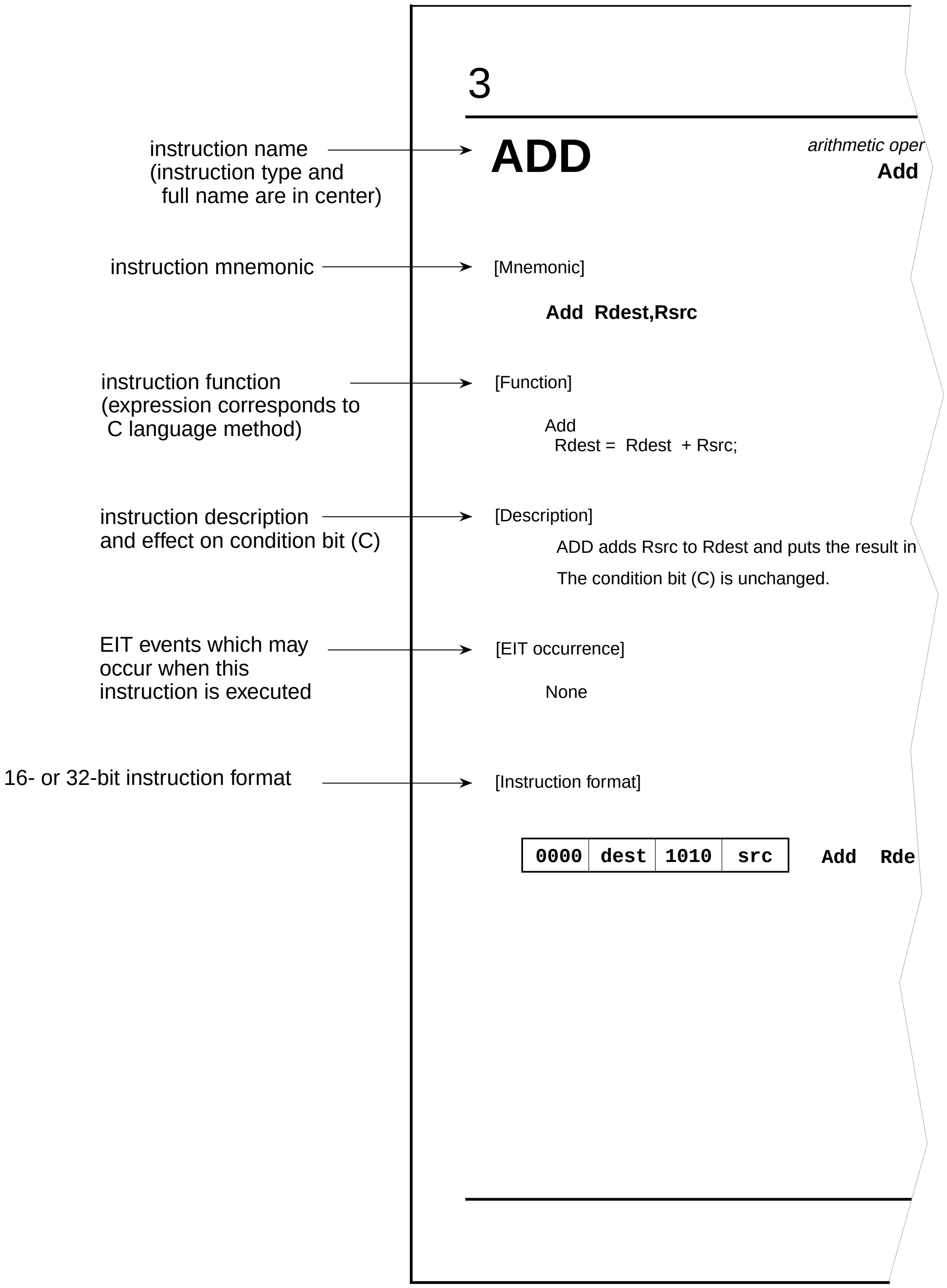
Shows possible EIT events (Exception, Interrupt, Trap) which may occur as the result of the instruction's execution. Only address exception (AE), floating-point exception (FPE) and trap (TRAP) may result from an instruction execution.

[Instruction format]

Shows the bit level instruction pattern (16 bits or 32 bits). Source and/or destination register numbers are put in the src and dest fields as appropriate. Any immediate or displacement value is put in the imm or disp field, its maximum size being determined by the width of the field provided for the particular instruction. Refer to **2.2 Instruction format** for detail.

3.2 Instruction description

This section lists M32R-FPU instructions in alphabetical order. Each page is laid out as shown below.



ADD

arithmetic/logic operation

Add

[Mnemonic]

```
ADD  Rdest,Rsrc
```

[Function]

Add
Rdest = Rdest + Rsrc;

[Description]

ADD adds Rsrc to Rdest and puts the result in Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	1010	src

ADD Rdest,Rsrc

ADD3

arithmetic operation instruction

Add 3-operand

ADD3

[Mnemonic]

`ADD3  Rdest,Rsrc,#imm16`

[Function]

Add

Rdest = Rsrc + (signed short) imm16;

[Description]

ADD3 adds the 16-bit immediate value to Rsrc and puts the result in Rdest. The immediate value is sign-extended to 32 bits before the operation.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1000	dest	1010	src	imm16

`ADD3 Rdest,Rsrc,#imm16`

ADDI

arithmetic operation instruction

Add immediate

[Mnemonic]

```
ADDI  Rdest,#imm8
```

[Function]

Add
Rdest = Rdest + (signed char) imm8;

[Description]

ADDI adds the 8-bit immediate value to Rdest and puts the result in Rdest.
The immediate value is sign-extended to 32 bits before the operation.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0100	dest	imm8

ADDI Rdest,#imm8

ADDV

arithmetic operation instruction

Add with overflow checking

ADDV

[Mnemonic]

```
ADDV   Rdest,Rsrc
```

[Function]

Add

Rdest = (signed) Rdest + (signed) Rsrc;
C = overflow ? 1 : 0;

[Description]

ADDV adds Rsrc to Rdest and puts the result in Rdest.
The condition bit (C) is set when the addition results in overflow; otherwise it is cleared.

[EIT occurrence]

None

[Encoding]

0000	dest	1000	src

`ADDV Rdest,Rsrc`

ADDV3 — *arithmetic operation instruction* — Add 3-operand with overflow checking — ADDV3

[Mnemonic]

`ADDV3 Rdest,Rsrc,#imm16`

[Function]

Add

Rdest = (signed) Rsrc + (signed) ((signed short) imm16);
C = overflow ? 1 : 0;

[Description]

ADDV3 adds the 16-bit immediate value to Rsrc and puts the result in Rdest. The immediate value is sign-extended to 32 bits before it is added to Rsrc.

The condition bit (C) is set when the addition results in overflow; otherwise it is cleared.

[EIT occurrence]

None

[Encoding]

1000	dest	1000	src	imm16

`ADDV3 Rdest,Rsrc,#imm16`

ADDX

arithmetic operation instruction

Add with carry

ADDX

[Mnemonic]

`ADDX  Rdest,Rsrc`

[Function]

Add

Rdest = (unsigned) Rdest + (unsigned) Rsrc + C;

C = carry_out ? 1 : 0;

[Description]

ADDX adds Rsrc and C to Rdest, and puts the result in Rdest.

The condition bit (C) is set when the addition result cannot be represented by a 32-bit unsigned integer; otherwise it is cleared.

[EIT occurrence]

None

[Encoding]

0000	dest	1001	src

`ADDX Rdest,Rsrc`

AND

logic operation instruction

AND

AND

[Mnemonic]

`AND Rdest,Rsrc`

[Function]

Logical AND

Rdest = Rdest & Rsrc;

[Description]

AND computes the logical AND of the corresponding bits of Rdest and Rsrc and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	1100	src

`AND Rdest,Rsrc`

AND3

logic operation instruction

AND 3-operand

AND3

[Mnemonic]

```
AND3   Rdest,Rsrc,#imm16
```

[Function]

Logical AND

Rdest = Rsrc & (unsigned short) imm16;

[Description]

AND3 computes the logical AND of the corresponding bits of Rsrc and the 16-bit immediate value, which is zero-extended to 32 bits, and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1000	dest	1100	src	imm16

```
AND3   Rdest,Rsrc,#imm16
```

BC *branch instruction* BC

Bit clear

M32R-FPU Extended Instruction

[Mnemonic]

(1) BC pcdisp8
(2) BC pcdisp24

[Function]

Branch

(1) if (C==1) PC = (PC & 0xfffffffc) + (((signed char) pcdisp8) << 2);
(2) if (C==1) PC = (PC & 0xfffffffc) + (sign_extend (pcdisp24) << 2);
where
#define sign_extend(x) (((signed) ((x)<< 8)) >>8)

[Description]

BC causes a branch to the specified label when the condition bit (C) is 1.

There are two instruction formats; which allows software, such as an assembler, to decide on the better format.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0111	1100	pcdisp8	**BC pcdisp8**

1111	1100		pcdisp24	**BC pcdisp24**

BCLR

bit operation

Bit clear

[M32R-FPU Extended Instruction]

[Mnemonic]

BCLR #bitpos,@(disp16,Rsrc)

[Function]

Bit operation for memory contents   Set 0 to specified bit.

* (signed char*) (Rsrc + (signed short) disp16) & = ~ (1<< (7-bitpos)) ;

[Description]

BCLR reads the byte data in the memory at the address specified by the Rsrc combined with the 16-bit displacement, and then stores the value of the bit that was specified by bitpos to be set to “0”. The displacement is sign-extended before the address calculation. bitpos becomes 0 to 7; MSB becomes 0 and LSB becomes 7. The memory is accessed in bytes. The LOCK bit is on while the BCLR instruction is executed, and is cleared when the execution is completed. The LOCK bit is internal to the CPU and cannot be directly read or written to by the user.

Condition bit C remains unchanged.

The LOCK bit is internal to the CPU and is the control bit for receiving all bus right requests from circuits other than the CPU.

Refer to the Users Manual for non-CPU bus right requests, as the handling differs according to the type of MCU.

[EIT occurrence]

None

[Encoding]

1010	0	bitpos	0111	src	disp16

BCLR #bitpos,@(disp16,Rsrc)

BEQ *branch instruction* Branch on equal to BEQ

[Mnemonic]

BEQ Rsrc1,Rsrc2,pcdisp16

[Function]

Branch

if (Rsrc1 == Rsrc2) PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BEQ causes a branch to the specified label when Rsrc1 is equal to Rsrc2.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	src1	0000	src2	pcdisp16

BEQ Rsrc1,Rsrc2,pcdisp16

BEQZ

branch instruction

Branch on equal to zero

BEQZ

[Mnemonic]

```
BEQZ  Rsrc,pcdisp16
```

[Function]

Branch

if (Rsrc == 0)   PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BEQZ causes a branch to the specified label when Rsrc is equal to zero.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	0000	1000	src	pcdisp16

```
BEQZ  Rsrc,pcdisp16
```

BGEZ

branch instruction

Branch on greater than or equal to zero

BGEZ

[Mnemonic]

```
BGEZ  Rsrc,pcdisp16
```

[Function]

Branch

if ((signed) Rsrc >= 0)   PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BGEZ causes a branch to the specified label when Rsrc treated as a signed 32-bit value is greater than or equal to zero.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	0000	1011	src	pcdisp16

```
BGEZ  Rsrc,pcdisp16
```

BGTZ

branch instruction

Branch on greater than zero

BGTZ

[Mnemonic]

`BGTZ  Rsrc,pcdisp16`

[Function]

Branch

if ((signed) Rsrc > 0) PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BGTZ causes a branch to the specified label when Rsrc treated as a signed 32-bit value is greater than zero.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	0000	1101	src	pcdisp16

`BGTZ  Rsrc,pcdisp16`

BL

branch instruction

Branch and link

[Mnemonic]

(1) BL pcdisp8
(2) BL pcdisp24

[Function]

Subroutine call (PC relative)

```
(1)  R14 = ( PC & 0xfffffffc ) + 4;
       PC = ( PC & 0xfffffffc ) + ( ( ( signed char ) pcdisp8 ) << 2 );
(2)  R14 = ( PC & 0xfffffffc ) + 4;
       PC = ( PC & 0xfffffffc ) + ( sign_extend ( pcdisp24 ) << 2 );
where
#define  sign_extend(x) ( ( ( signed ) ( (x)<< 8 ) ) >>8 )
```

[Description]

BL causes an unconditional branch to the address specified by the label and puts the return address in R14.

There are two instruction formats; this allows software, such as an assembler, to decide on the better format.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0111	1110	pcdisp8	BL pcdisp8

1111	1110		pcdisp24	BL pcdisp24

BLEZ

branch instruction

Branch on less than or equal to zero

BLEZ

[Mnemonic]

```
BLEZ  Rsrc,pcdisp16
```

[Function]

Branch

if ((signed) Rsrc <= 0)   PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BLEZ causes a branch to the specified label when the contents of Rsrc treated as a signed 32-bit value, is less than or equal to zero.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	0000	1100	src	pcdisp16

```
BLEZ  Rsrc,pcdisp16
```

BLTZ *branch instruction* Branch on less than zero BLTZ

[Mnemonic]

```
BLTZ  Rsrc,pcdisp16
```

[Function]

Branch

if ((signed) Rsrc < 0)   PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BLTZ causes a branch to the specified label when Rsrc treated as a signed 32-bit value is less than zero.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	0000	1010	src	pcdisp16

```
BLTZ  Rsrc,pcdisp16
```

BNC

branch instruction
Branch on not C-bit

[Mnemonic]

(1) BNC pcdisp8
(2) BNC pcdisp24

[Function]

Branch
(1) if (C==0) PC = (PC & 0xfffffffc) + (((signed char) pcdisp8) << 2);
(2) if (C==0) PC = (PC & 0xfffffffc) + (sign_extend (pcdisp24) << 2);
where
#define sign_extend(x) (((signed) ((x)<< 8)) >>8)

[Description]

BNC branches to the specified label when the condition bit (C) is 0.

There are two instruction formats; this allows software, such as an assembler, to decide on the better format.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0111	1101	pcdisp8	**BNC pcdisp8**

1111	1101		pcdisp24	**BNC pcdisp24**

BNE *branch instruction* Branch on not equal to BNE

[Mnemonic]

`BNE   Rsrc1,Rsrc2,pcdisp16`

[Function]

Branch

if (Rsrc1 != Rsrc2) PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BNE causes a branch to the specified label when Rsrc1 is not equal to Rsrc2.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	src1	0001	src2	pcdisp16

`BNE   Rsrc1,Rsrc2,pcdisp16`

BNEZ

branch instruction

Branch on not equal to zero

BNEZ

[Mnemonic]

```
BNEZ  Rsrc,pcdisp16
```

[Function]

Branch

if (Rsrc != 0)   PC = (PC & 0xfffffffc) + (((signed short) pcdisp16) << 2);

[Description]

BNEZ causes a branch to the specified label when Rsrc is not equal to zero.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1011	0000	1001	src	pcdisp16

```
BNEZ Rsrc,pcdisp16
```

BRA

branch instruction

Branch

BRA

[Mnemonic]

(1) BRA pcdisp8

(2) BRA pcdisp24

[Function]

Branch

(1) PC = (PC & 0xfffffffc) + (((signed char) pcdisp8) << 2);

(2) PC = (PC & 0xfffffffc) + (sign_extend (pcdisp24) << 2);

where

#define sign_extend(x) (((signed) ((x)<< 8)) >>8)

[Description]

BRA causes an unconditional branch to the address specified by the label.

There are two instruction formats; this allows software, such as an assembler, to decide on the better format.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0111	1111	pcdisp8	**BRA pcdisp8**

1111	1111		pcdisp24	**BRA pcdisp24**

BSET

bit operation Instructions

Bit set

[M32R-FPU Extended Instruction]

[Mnemonic]

BSET #bitpos,@(disp16,Rsrc)

[Function]

Bit operation for memory contents Set 0 to specified bit.

* (signed char*) (Rsrc + (signed short) disp16) : = (1<< (7-bitpos)) ;

[Description]

BSET reads the byte data in the memory at the address specified by the Rsrc combined with the 16-bit displacement, and then stores the value of the bit that was specified by bitpos to be set to “1”. The displacement is sign-extended before the address calculation. bitpos becomes 0 to 7; MSB becomes 0 and LSB becomes 7. The memory is accessed in bytes. The LOCK bit is on while the BSET instruction is executed, and is cleared when the execution is completed. The LOCK bit is internal to the CPU and cannot be directly read or written to by the user.

Condition bit C remains unchanged.

The LOCK bit is internal to the CPU and is the control bit for receiving all bus right requests from circuits other than the CPU.

Refer to the Users Manual for non-CPU bus right requests, as the handling differs according to the type of MCU.

[EIT occurrence]

None

[Encoding]

1010	0	bitpos	0110	src	disp16

BSET #bitpos,@(disp16,Rsrc)

BTST

bit operation Instructions

Bit test

[M32R-FPU Extended Instruction]

[Mnemonic]

BTST #bitpos,Rsrc

[Function]

Remove the bit specified by the register.

C = Rsrc >> (7-bitpos)) &1;

[Description]

Take out the bit specified as bitpos within the Rsrc lower eight bits and sets it in the condition bit (C). bitpos becomes 0 to 7, MSB becomes 0 and LSB becomes 7.

[EIT occurrence]

None

[Encoding]

0000	0	bitpos	1111	src

BTST #bitpos,Rsrc

CLRPSW

bit operation Instructions

Clear PSW

[M32R-FPU Extended Instruction]

[Mnemonic]

CLRPSW #imm8

[Function]

Set the undefined SM, IE, and C bits of PSW to 0.

PSW& = ~imm8 : 0xffffff00

[Description]

Set the AND result s of the reverse value of b0 (MSB), b1, and b7 (LSB) of the 8-bit immediate value and bits SM, IE, and C of PSW to the corresponding SM, IE, and C bits. When b7 (LSB) or #imm8 is 1, the condition bit (C) goes to 0. All other bits remain unchanged.

[EIT occurrence]

None

[Encoding]

0111	0010	imm8

CLRPSW #imm8

CMP *compare instruction* Compare CMP

[Mnemonic]

`CMP Rsrc1,Rsrc2`

[Function]

Compare

C = ((signed) Rsrc1 < (signed) Rsrc2) ? 1:0;

[Description]

The condition bit (C) is set to 1 when Rsrc1 is less than Rsrc2. The operands are treated as signed 32-bit values.

[EIT occurrence]

None

[Encoding]

0000	src1	0100	src2	**CMP Rsrc1,Rsrc2**

CMPI

compare instruction

Compare immediate

CMPI

[Mnemonic]

```
CMPI  Rsrc,#imm16
```

[Function]

Compare

C = ((signed) Rsrc < (signed short) imm16) ? 1:0;

[Description]

The condition bit (C) is set when Rsrc is less than 16-bit immediate value. The operands are treated as signed 32-bit values. The immediate value is sign-extended to 32-bit before the operation.

[EIT occurrence]

None

[Encoding]

1000	0000	0100	src	imm16

```
CMPI  Rsrc,#imm16
```

CMPU

compare instruction

Compare unsigned

CMPU

[Mnemonic]

```
CMPU  Rsrc1,Rsrc2
```

[Function]

Compare

C = ((unsigned) Rsrc1 < (unsigned) Rsrc2) ? 1:0;

[Description]

The condition bit (C) is set when Rsrc1 is less than Rsrc2. The operands are treated as unsigned 32-bit values.

[EIT occurrence]

None

[Encoding]

0000	src1	0101	src2

```
CMPU  Rsrc1,Rsrc2
```

CMPUI

compare instruction
Compare unsigned immediate

CMPUI

[Mnemonic]

```
CMPUI  Rsrc,#imm16
```

[Function]

Compare
C = ((unsigned) Rsrc < (unsigned) ((signed short) imm16)) ? 1:0;

[Description]

The condition bit (C) is set when Rsrc is less than the 16-bit immediate value. The operands are treated as unsigned 32-bit values. The immediate value is sign-extended to 32-bit before the operation.

[EIT occurrence]

None

[Encoding]

1000	0000	0101	src	imm16

```
CMPUI Rsrc,#imm16
```

DIV *multiply and divide instruction* Divide DIV

[Mnemonic]

`DIV   Rdest,Rsrc`

[Function]

Signed division
Rdest = (signed) Rdest / (signed) Rsrc;

[Description]

DIV divides Rdest by Rsrc and puts the quotient in Rdest.
The operands are treated as signed 32-bit values and the result is rounded toward zero.
The condition bit (C) is unchanged.
When Rsrc is zero, Rdest is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	0000	src	0000	0000	0000	0000

`DIV   Rdest,Rsrc`

DIVU

multiply and divide instruction

Divide unsigned

DIVU

[Mnemonic]

```
DIVU  Rdest,Rsrc
```

[Function]

Unsigned division
Rdest = (unsigned) Rdest / (unsigned) Rsrc;

[Description]

DIVU divides Rdest by Rsrc and puts the quotient in Rdest.
The operands are treated as unsigned 32-bit values and the result is rounded toward zero.
The condition bit (C) is unchanged.
When Rsrc is zero, Rdest is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	0001	src	0000	0000	0000	0000

```
DIVU  Rdest,Rsrc
```

FADD *floating-point Instructions* FADD

Floating-point add
[M32R-FPU Extended Instruction]

[Mnemonic]

```
FADD   Rdest,Rsrc1,Rsrc2
```

[Function]

Floating-point add

Rdest = Rsrc1 + Rsrc2 ;

[Description]

Add the floating-point single precision values stored in Rsrc1 and Rsrc2 and store the result in Rdest. The result is rounded according to the RM field of FPSR. The DN bit of FPSR handles the modification of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Overflow (OVF)
- Underflow (UDF)
- Inexact Exception (IXCT)

[Encoding]

1101	src1	0000	src2	0000	dest	0000	0000

```
FADD   Rdest,Rsrc1,Rsrc2
```

FADD

floating point Instructions

Floating-point addd

[M32R-FPU Extended Instruction]

FADD

[Supplemental Operation Description]

The following shows the values of Rsrc1 and Rsrc2 and the operation results when DN = 0 and DN = 1.

DN = 0

		Rsrc2							
		Normalized Number	+0	-0	+Infinity	-Infinity	Denormalized Number	QNaN	SNaN
	Normalized Number	add							
	+0		+0	(Note)		-Infinity			
	-0		(Note)	-0					
Rsrc1	+Infinity				+Infinity	IVLD			
	-Infinity		-Infinity		IVLD	-Infinity			
	Denormalized Number						UIPL		
	QNaN							QNaN	
	SNaN								IVLD

DN = 1

		Rsrc2						
		Normalized Number	+0, +Denormalized Number	-0, -Denormalized Number	+Infinity	-Infinity	QNaN	SNaN
	Normalized Number	add	Normalized Number					
	+0, +Denormalized Number	Normalized Number	+0	(Note)		-Infinity		
	-0, -Denormalized Number		(Note)	-0				
Rsrc1	+Infinity				+Infinity	IVLD		
	-Infinity		-Infinity		IVLD	-Infinity		
	QNaN						QNaN	
	SNaN							IVLD

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

Note: The rounding mode is “-0” when rounding toward “-Infinity”, and “+0” when rounding toward any other direction.

FCMP

floating point Instructions

Floating-point compare

[M32R-FPU Extended Instruction]

FCMP

[Mnemonic]

`FCMP   Rdest,Rsrc1,Rsrc2`

[Function]

Floating-point compare

Rdest = (comparison results of Rsrc1 and Rsrc2);

When at least one value, either Rsrc1 or Rsrc2, is SNaN, a floating-point exception (other than Invalid Operation Exception) occurs.

[Description]

Compare the floating-point single precision values stored in Rsrc1 and Rsrc2 and store the result in Rdest. The results of the comparison can be determined y the following methods.

Rdest		Comparison Results	Typical instructions used to determine comparison results
b0=0	All bits, b1 to b31, are 0.	Rsrc1=Rsrc2	beqz Rdest, LABEL
	b1 to b9=111 1111 11, Bits b10 to b31 are an undefined.	Comparison invalid	bgtz Rdest, LABEL
	All others	Rsrc1>Rsrc2	
b0=1	Bits b1 to b31 are an undefined.	Rsrc1<Rsrc2	bltz Rdest, LABEL

The DN bit of FPSR handles the conversion of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)

[Encoding]

1101	src1	0000	src2	0000	dest	1100	0000

`FCMP   Rdest,Rsrc1,Rsrc2`

FCMP

floating point Instructions

Floating-point compare

[M32R-FPU Extended Instruction]

FCMP

[Supplemental Operation Description]

The following shows the values of Rsrc1 and Rsrc2 and the operation results when DN = 0 and DN = 1.

DN = 0

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>Denormalized Number</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">Rsrc1</td><td>Normalized Number</td><td colspan="3">comparison</td><td rowspan="3">-Infinity</td><td rowspan="4">+Infinity</td><td rowspan="5"></td><td rowspan="6"></td><td rowspan="7"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td colspan="2" rowspan="2">00000000</td></tr>
<tr><td>-0</td></tr>
<tr><td>+Infinity</td><td colspan="3">+Infinity</td><td>00000000</td></tr>
<tr><td>-Infinity</td><td colspan="4">-Infinity</td><td>00000000</td></tr>
<tr><td>Denormalized Number</td><td colspan="6">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="7">comparison invalid</td></tr>
<tr><td>SNaN</td><td colspan="8">IVLD</td></tr>
</table>

DN = 1

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="7">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0, +Denormalized Number</td><td>-0, -Denormalized Number</td><td>+Infinity</td><td>-Infinity</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="7">Rsrc1</td><td>Normalized Number</td><td colspan="3">comparison</td><td rowspan="3">-Infinity</td><td rowspan="4">+Infinity</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0, +Denormalized Number</td><td rowspan="2"></td><td colspan="2" rowspan="2">00000000</td></tr>
<tr><td>-0, -Denormalized Number</td></tr>
<tr><td>+Infinity</td><td colspan="3">+Infinity</td><td>00000000</td></tr>
<tr><td>-Infinity</td><td colspan="4">-Infinity</td><td>00000000</td></tr>
<tr><td>QNaN</td><td colspan="6">comparison invalid</td></tr>
<tr><td>SNaN</td><td colspan="7">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception

UIPL: Unimplemented Exception

NaN: Not a Number

SNaN: Signaling NaN

QNaN: Quiet NaN

FCMPE *floating-point Instructions* FCMPE

Floating-point compare with exception if unordered

[M32R-FPU Extended Instruction]

[Mnemonic]

```
FCMPE   Rdest,Rsrc1,Rsrc2
```

[Function]

Floating-point compare

Rdest = (comparison results of Rsrc1 and Rsrc2);

When at least one value, either Rsrc1 or Rsrc2, is QNaN or SNaN, a floating-point exception (other than Invalid Operation Exception) occurs.

[Description]

Compare the floating-point single precision values stored in Rsrc1 and Rsrc2 and store the result in Rdest. The results of the comparison can be determined y the following methods.

Rdest		Comparison Results	Typical instructions used to determine comparison results
b0=0	All bits, b1 to b31, are 0.	Rsrc1=Rsrc2	beqz Rdest, LABEL
	b1 to b9=111 1111 11, Bits b10 to b31 are an undefined. (Note)	Comparison invalid	bgtz Rdest, LABEL
	All others	Rsrc1>Rsrc2	
b0=1	Bits b1 to b31 are an undefined.	Rsrc1<Rsrc2	bltz Rdest, LABEL

Note: Only when EV bit (b21 of FPSR Register) = “0”.

The DN bit of FPSR handles the conversion of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)

[Encoding]

1101	src1	0000	src2	0000	dest	1101	0000

```
FCMPE   Rdest,Rsrc1,Rsrc2
```

FCMPE

floating point Instructions

Floating-point compare with exception if unordered

[M32R-FPU Extended Instruction]

[Supplemental Operation Description]

The following shows the values of Rsrc1 and Rsrc2 and the operation results when DN = 0 and DN = 1.

DN = 0

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">Rsrc2</th></tr>
<tr><th>Normalized Number</th><th>+0</th><th>-0</th><th>+Infinity</th><th>-Infinity</th><th>Denormalized Number</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="8">Rsrc1</td><td>Normalized Number</td><td colspan="3">comparison</td><td rowspan="3">-Infinity</td><td rowspan="4">+Infinity</td><td rowspan="5"></td><td rowspan="6" colspan="2"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td rowspan="2" colspan="2">00000000</td></tr>
<tr><td>-0</td></tr>
<tr><td>+Infinity</td><td colspan="3">+Infinity</td><td>00000000</td></tr>
<tr><td>-Infinity</td><td colspan="4">-Infinity</td><td>00000000</td></tr>
<tr><td>Denormalized Number</td><td colspan="6">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="8" rowspan="2">IVLD</td></tr>
<tr><td>SNaN</td></tr>
</table>

DN = 1

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="7">Rsrc2</th></tr>
<tr><th>Normalized Number</th><th>+0, + Denormalized Number</th><th>-0, - Denormalized Number</th><th>+Infinity</th><th>-Infinity</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="7">Rsrc1</td><td>Normalized Number</td><td colspan="3">comparison</td><td rowspan="3">-Infinity</td><td rowspan="4">+Infinity</td><td rowspan="5" colspan="2"></td></tr>
<tr><td>+0, + Denormalized Number</td><td rowspan="2"></td><td rowspan="2" colspan="2">00000000</td></tr>
<tr><td>-0, - Denormalized Number</td></tr>
<tr><td>+Infinity</td><td colspan="3">+Infinity</td><td>00000000</td></tr>
<tr><td>-Infinity</td><td colspan="4">-Infinity</td><td>00000000</td></tr>
<tr><td>QNaN</td><td colspan="7" rowspan="2">IVLD</td></tr>
<tr><td>SNaN</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

FDIV

floating-point Instructions

Floating-point divide

[M32R-FPU Extended Instruction]

FDIV

[Mnemonic]

FDIV Rdest,Rsrc1,Rsrc2

[Function]

Floating-point divide

Rdest = Rsrc1 / Rsrc2 ;

[Description]

Divide the floating-point single precision value stored in Rsrc1 by the floating-point single precision value stored in Rsrc1 and store the result in Rdest. The result is rounded according to the RM field of FPSR. The DN bit of FPSR handles the modification of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Overflow (OVF)
- Underflow (UDF)
- Inexact Exception (IXCT)
- Zero Divide Exception (DIV0)

[Encoding]

1101	src1	0000	src2	0010	dest	0000	0000

FDIV Rdest,Rsrc1,Rsrc2

FDIV

floating point Instructions

Floating-point divide

[M32R-FPU Extended Instruction]

FDIV

[Supplemental Operation Description]

The following shows the values of Rsrc1 and Rsrc2 and the operation results when DN = 0 and DN = 1.

DN = 0

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">Rsrc2</th></tr>
<tr><th>Normalized Number</th><th>+0</th><th>-0</th><th>+Infinity</th><th>-Infinity</th><th>Denormalized Number</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="8">Rsrc1</td><td>Normalized Number</td><td>divide</td><td colspan="2">DIV0</td><td colspan="2">0</td><td rowspan="5"></td><td rowspan="6"></td><td rowspan="7"></td></tr>
<tr><td>+0</td><td rowspan="2">0</td><td colspan="2" rowspan="2">IVLD</td><td>+0</td><td>-0</td></tr>
<tr><td>-0</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td>+Infinity</td><td>-Infinity</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>Denormalized Number</td><td colspan="6">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="7">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="8">IVLD</td></tr>
</table>

DN = 1

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="7">Rsrc2</th></tr>
<tr><th>Normalized Number</th><th>+0, + Denormalized Number</th><th>-0, - Denormalized Number</th><th>+Infinity</th><th>-Infinity</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="7">Rsrc1</td><td>Normalized Number</td><td>divide</td><td colspan="2">DIV0</td><td colspan="2">0</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0, + Denormalized Number</td><td rowspan="2">0</td><td colspan="2" rowspan="2">IVLD</td><td>+0</td><td>-0</td></tr>
<tr><td>-0, - Denormalized Number</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td>+Infinity</td><td>-Infinity</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="7">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
DIV0: Zero Divide Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

FMADD

floating-point Instructions
Floating-point multiply and add
[M32R-FPU Extended Instruction]

FMADD

[Mnemonic]

```
FMADD  Rdest,Rsrc1,Rsrc2
```

[Function]

Floating-point multiply and add
Rdest = Rdest + Rsrc1 * Rsrc2 ;

[Description]

This instruction is executed in the following 2 steps.

● Step 1

Multiply the floating-point single precision value stored in Rsrc1 by the floating-point single precision value stored in Rsrc2.

The multiplication result is rounded toward 0 regardless of the value in the RM field of FPSR.

● Step 2

Add the result of Step 1 (the rounded value) and the floating-point single precision value stored in Rdest. The result is rounded according to the RM field of FPSR.

The result of this operation is stored in Rdest. Exceptions are determined in both Step 1 and Step 2. The DN bit of FPSR handles the conversion of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Overflow (OVF)
- Underflow (UDF)
- Inexact Exception (IXCT)

[Encoding]

1101	src1	0000	src2	0011	dest	0000	0000

```
FMADD  Rdest,Rsrc1,Rsrc2
```

FMADD

floating point Instructions

Floating-point multiply and add

[M32R-FPU Extended Instruction]

FMADD

[Supplemental Operation Description]

The following shows the values of Rsrc1, Rsrc2 and Rdest and the operation results when DN = 0 and DN = 1.

DN=0

Value after Multiplication Operation

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>Denormalized Number</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">Rsrc1</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td colspan="2">Infinity</td><td rowspan="5"></td><td rowspan="6"></td><td rowspan="7"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td>+0</td><td>-0</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-0</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td colspan="2" rowspan="2">IVLD</td><td>+Infinity</td><td>-Infinity</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>Denormalized Number</td><td colspan="6">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="7">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="8">IVLD</td></tr>
</table>

Value after Addition Operation

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="6">Value after Multiplication Operation</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>QNaN</td></tr>
<tr><td rowspan="8">Rdest</td><td>Normalized Number</td><td colspan="3">add</td><td rowspan="3"></td><td rowspan="3">-Infinity</td><td rowspan="6"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td>+0</td><td>(Note)</td></tr>
<tr><td>-0</td><td>(Note)</td><td>-0</td></tr>
<tr><td>+Infinity</td><td colspan="4">+Infinity</td><td>IVLD</td></tr>
<tr><td>-Infinity</td><td colspan="3">-Infinity</td><td>IVLD</td><td>-Infinity</td></tr>
<tr><td>Denormalized Number</td><td colspan="5">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="6">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

Note: The rounding mode is “-0” when rounding toward “-Infinity”, and “+0” when rounding toward any other direction.

FMADD

floating point Instructions

Floating-point multiply and add
[M32R-FPU Extended Instruction]

DN=1

Value after Multiplication Operation

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="7">Rsrc2</th></tr>
<tr><th>Normalized Number</th><th>+0, +Denormalized Number</th><th>-0, - Denormalized Number</th><th>+Infinity</th><th>-Infinity</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="7">Rsrc1</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td colspan="2">Infinity</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0, + Denormalized Number</td><td></td><td>+0</td><td>-0</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-0, - Denormalized Number</td><td></td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td colspan="2" rowspan="2">IVLD</td><td>+Infinity</td><td>-Infinity</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="7">IVLD</td></tr>
</table>

Value after Addition Operation

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="6">Value after Multiplication Operation</th></tr>
<tr><th>Normalized Number</th><th>+0</th><th>-0</th><th>+Infinity</th><th>-Infinity</th><th>QNaN</th></tr>
<tr><td rowspan="7">Rdest</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td rowspan="3"></td><td rowspan="3">-Infinity</td><td rowspan="5"></td></tr>
<tr><td>+0</td><td></td><td>+0</td><td>(Note)</td></tr>
<tr><td>-0</td><td></td><td>(Note)</td><td>-0</td></tr>
<tr><td>+Infinity</td><td colspan="4">+Infinity</td><td>IVLD</td></tr>
<tr><td>-Infinity</td><td colspan="3">-Infinity</td><td>IVLD</td><td>-Infinity</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="6">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

Note: The rounding mode is “-0” when rounding toward “-Infinity”, and “+0” when rounding toward any other direction.

FMSUB

floating-point Instructions

Floating-point multiply and subtract

[M32R-FPU Extended Instruction]

FMSUB

[Mnemonic]

`FMSUB  Rdest,Rsrc1,Rsrc2`

[Function]

Floating-point multiply and subtract

Rdest = Rdest - Rsrc1 * Rsrc2 ;

[Description]

This instruction is executed in the following 2 steps.

● Step 1

Multiply the floating-point single precision value stored in Rsrc1 by the floating-point single precision value stored in Rsrc2.

The multiplication result is rounded toward 0 regardless of the value in the RM field of FPSR.

● Step 2

Subtract the result (rounded value) of Step 1 from the floating-point single precision value stored in Rdest.

The subtraction result is rounded according to the RM field of FPSR.

The result of this operation is stored in Rdest. Exceptions are determined in both Step 1 and Step 2. The DN bit of FPSR handles the conversion of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Overflow (OVF)
- Underflow (UDF)
- Inexact Exception (IXCT)

[Encoding]

1101	src1	0000	src2	0011	dest	0100	0000

`FMSUB  Rdest,Rsrc1,Rsrc2`

FMSUB

floating point Instructions

Floating-point multiply and subtract
[M32R-FPU Extended Instruction]

FMSUB

[Supplemental Operation Description]

The following shows the values of Rsrc1, Rsrc2 and Rdest and the operation results when DN = 0 and DN = 1.

DN=0

Value after Multiplication Operation

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>Denormalized Number</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">Rsrc1</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td colspan="2">Infinity</td><td rowspan="6">UIPL</td><td rowspan="7">QNaN</td><td rowspan="8">IVLD</td></tr>
<tr><td>+0</td><td rowspan="2"></td><td>+0</td><td>-0</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-0</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td colspan="2" rowspan="2">IVLD</td><td>+Infinity</td><td>-Infinity</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>Denormalized Number</td><td colspan="5"></td></tr>
<tr><td>QNaN</td><td colspan="6"></td></tr>
<tr><td>SNaN</td><td colspan="7"></td></tr>
</table>

Value after Subtraction Operation

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="6">Value after Multiplication Operation</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>QNaN</td></tr>
<tr><td rowspan="8">Rdest</td><td>Normalized Number</td><td colspan="3">Subtraction</td><td rowspan="4">+Infinity</td><td rowspan="3">-Infinity</td><td rowspan="7">QNaN</td></tr>
<tr><td>+0</td><td rowspan="2"></td><td>+0</td><td>(Note)</td></tr>
<tr><td>-0</td><td>(Note)</td><td>-0</td></tr>
<tr><td>+Infinity</td><td colspan="3"></td><td>IVLD</td></tr>
<tr><td>-Infinity</td><td colspan="3">-Infinity</td><td>IVLD</td><td>-Infinity</td></tr>
<tr><td>Denormalized Number</td><td colspan="5">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="5"></td></tr>
<tr><td>SNaN</td><td colspan="6">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

Note: The rounding mode is “-0” when rounding toward “-Infinity”, and “+0” when rounding toward any other direction.

FMSUB

floating point Instructions

Floating-point multiply and subtract
[M32R-FPU Extended Instruction]

DN=1

Value after Multiplication Operation

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="7">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0, +Denormalized Number</td><td>-0, - Denormalized Number</td><td>+Infinity</td><td>-Infinity</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="7">Rsrc1</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td colspan="2">Infinity</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0, +Denormalized Number</td><td rowspan="2"></td><td>+0</td><td>-0</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-0, - Denormalized Number</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td colspan="2" rowspan="2">IVLD</td><td>+Infinity</td><td>-Infinity</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="7">IVLD</td></tr>
</table>

Value after Subtraction Operation

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="6">Value after Multiplication Operation</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>QNaN</td></tr>
<tr><td rowspan="7">Rdest</td><td>Normalized Number</td><td colspan="3">Subtraction</td><td rowspan="3">-Infinity</td><td rowspan="4">+Infinity</td><td rowspan="5"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td>(Note)</td><td>+0</td></tr>
<tr><td>-0</td><td>-0</td><td>(Note)</td></tr>
<tr><td>+Infinity</td><td colspan="3">+Infinity</td><td>IVLD</td></tr>
<tr><td>-Infinity</td><td colspan="4">-Infinity</td><td>IVLD</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="6">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

Note: The rounding mode is “-0” when rounding toward “-Infinity”, and “+0” when rounding toward any other direction.

FMUL

floating-point Instructions

Floating-point multiply

[M32R-FPU Extended Instruction]

FMUL

[Mnemonic]

```
FMUL  Rdest,Rsrc1,Rsrc2
```

[Function]

Floating-point multiply

Rdest = Rsrc1 * Rsrc2 ;

[Description]

Multiply the floating-point single precision value stored in Rsrc1 by the floating-point single precision value stored in Rsrc2 and store the results in Rdest. The result is rounded according to the RM field of FPSR. The DN bit of FPSR handles the modification of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Overflow (OVF)
- Underflow (UDF)
- Inexact Exception (IXCT)

[Encoding]

1101	src1	0000	src2	0001	dest	0000	0000

```
FMUL  Rdest,Rsrc1,Rsrc2
```

FMUL

floating point Instructions

Floating-point multiply

[M32R-FPU Extended Instruction]

FMUL

[Supplemental Operation Description]

The following shows the values of Rsrc1 and Rsrc2 and the operation results when DN = 0 and DN = 1.

DN=0

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0</td><td>-0</td><td>+Infinity</td><td>-Infinity</td><td>Denormalized Number</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">Rsrc1</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td colspan="2">Infinity</td><td rowspan="5"></td><td rowspan="6"></td><td rowspan="7"></td></tr>
<tr><td>+0</td><td rowspan="2"></td><td>+0</td><td>-0</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-0</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td colspan="2" rowspan="2">IVLD</td><td>+Infinity</td><td>-Infinity</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>Denormalized Number</td><td colspan="6">UIPL</td></tr>
<tr><td>QNaN</td><td colspan="7">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="8">IVLD</td></tr>
</table>

DN=1

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="7">Rsrc2</td></tr>
<tr><td>Normalized Number</td><td>+0, +Denormalized Number</td><td>-0, - Denormalized Number</td><td>+Infinity</td><td>-Infinity</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="7">Rsrc1</td><td>Normalized Number</td><td colspan="3">Multiplication</td><td colspan="2">Infinity</td><td rowspan="5"></td><td rowspan="6"></td></tr>
<tr><td>+0, +Denormalized Number</td><td rowspan="2"></td><td>+0</td><td>-0</td><td colspan="2" rowspan="2">IVLD</td></tr>
<tr><td>-0, - Denormalized Number</td><td>-0</td><td>+0</td></tr>
<tr><td>+Infinity</td><td rowspan="2">Infinity</td><td colspan="2" rowspan="2">IVLD</td><td>+Infinity</td><td>-Infinity</td></tr>
<tr><td>-Infinity</td><td>-Infinity</td><td>+Infinity</td></tr>
<tr><td>QNaN</td><td colspan="6">QNaN</td></tr>
<tr><td>SNaN</td><td colspan="7">IVLD</td></tr>
</table>

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

FSUB

floating-point Instructions

Floating-point subtract

[M32R-FPU Extended Instruction]

FSUB

[Mnemonic]

FSUB Rdest,Rsrc1,Rsrc2

[Function]

Floating-point subtract

Rdest = Rsrc1 - Rsrc2 ;

[Description]

Subtract the floating-point single precision value stored in Rsrc2 from the floating-point single precision value stored in Rsrc1 and store the results in Rdest. The result is rounded according to the RM field of FPSR. The DN bit of FPSR handles the modification of denormalized numbers. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Overflow (OVF)
- Underflow (UDF)
- Inexact Exception (IXCT)

[Encoding]

1101	src1	0000	src2	0000	dest	0100	0000

FSUB Rdest,Rsrc1,Rsrc2

FSUB *floating point Instructions* FSUB

Floating-point subtract
[M32R-FPU Extended Instruction]

[Supplemental Operation Description]

The following shows the values of Rsrc1 and Rsrc2 and the operation results when DN = 0 and DN = 1.

DN = 0

Rsrc1 \ Rsrc2	Normalized Number	+0	-0	+Infinity	-Infinity	Denormalized Number	QNaN	SNaN
Normalized Number	Subtraction	Subtraction	Subtraction	-Infinity	+Infinity	UIPL	QNaN	IVLD
+0	Subtraction	(Note)	+0	-Infinity	+Infinity	UIPL	QNaN	IVLD
-0	Subtraction	-0	(Note)	-Infinity	+Infinity	UIPL	QNaN	IVLD
+Infinity	+Infinity	+Infinity	+Infinity	IVLD	+Infinity	UIPL	QNaN	IVLD
-Infinity	-Infinity	-Infinity	-Infinity	-Infinity	IVLD	UIPL	QNaN	IVLD
Denormalized Number	UIPL	UIPL	UIPL	UIPL	UIPL	UIPL	QNaN	IVLD
QNaN	QNaN	QNaN	QNaN	QNaN	QNaN	QNaN	QNaN	IVLD
SNaN	IVLD	IVLD	IVLD	IVLD	IVLD	IVLD	IVLD	IVLD

DN = 1

Rsrc1 \ Rsrc2	Normalized Number	+0, +Denormalized Number	-0, -Denormalized Number	+Infinity	-Infinity	QNaN	SNaN
Normalized Number	Subtraction	Subtraction	Subtraction	-Infinity	+Infinity	QNaN	IVLD
+0, +Denormalized Number	Subtraction	(Note)	+0	-Infinity	+Infinity	QNaN	IVLD
-0, -Denormalized Number	Subtraction	-0	(Note)	-Infinity	+Infinity	QNaN	IVLD
+Infinity	+Infinity	+Infinity	+Infinity	IVLD	+Infinity	QNaN	IVLD
-Infinity	-Infinity	-Infinity	-Infinity	-Infinity	IVLD	QNaN	IVLD
QNaN	QNaN	QNaN	QNaN	QNaN	QNaN	QNaN	IVLD
SNaN	IVLD	IVLD	IVLD	IVLD	IVLD	IVLD	IVLD

IVLD: Invalid Operation Exception
UIPL: Unimplemented Exception
NaN: Not a Number
SNaN: Signaling NaN
QNaN: Quiet NaN

Note: The rounding mode is “-0” when rounding toward “-Infinity”, and “+0” when rounding toward any other direction.

FTOI

floating-point Instructions

Float to Integer

[M32R-FPU Extended Instruction]

FTOI

[Mnemonic]

```
FTOI  Rdest,Rsrc
```

[Function]

Convert the floating-point single precision value to 32-bit integer.

Rdest = (signed int) Rsrc ;

[Description]

Convert the floating-point single precision value stored in Rsrc to a 32-bit integer and store the result in Rdest.

The result is rounded toward 0 regardless of the value in the RM field of FPSR. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Inexact Exception (IXCT)

[Encoding]

1101	src	0000	0000	0100	dest	1000	0000

```
FTOI  Rdest,Rsrc
```

FTOI

floating point Instructions

Float to Integer

[M32R-FPU Extended Instruction]

FTOI

[Supplemental Operation Description]

The results of the FTOI instruction executed based on the Rsrc value, both when DN = 0 and DN = 1, are shown in below.

DN = 0

Rsrc Value (exponent with no bias)		Rdest	Exception
Rsrc ≥ 0	+Infinity	When EIT occurs: no change	Invalid Operation Exception
	127 ≥ exp ≥ 31	Other EIT: H'7FFF FFFF	
	30 ≥ exp ≥ -126	H'0000 0000 to H'7FFF FF80	No change (Note 1)
	+Denormalized value	No change	Unimplemented Exception
	+0	H'0000 0000	No change
Rsrc < 0	-0		
	-Denormalized value	No change	Unimplemented Exception
	30 ≥ exp ≥ -126	H'0000 0000 to H'8000 0080	No change (Note 1)
	127 ≥ exp ≥ 31	When EIT occurs: no change	Invalid Operation Exception
	-Infinity	Other EIT: H'8000 0080	(Note 2)
NaN	QNaN	When EIT occurs: no change Other EIT:	Invalid Operation Exception
	SNaN	Signed bit = 0:H'7FFF FFFF Signed bit = 1:H'8000 0000	

Note 1: Inexact Exception occurs when rounding is performed.
2: Inexact Exception does not occur when Rsrc = H'CF00 0000.

DN = 1

Rsrc Value (exponent with no bias)		Rdest	Exception
Rsrc ≥ 0	+Infinity	When EIT occurs: no change	Invalid Operation Exception
	127 ≥ exp ≥ 31	Other EIT: H'7FFF FFFF	
	30 ≥ exp ≥ -126	H'0000 0000 to H'7FFF FF80	No change (Note 1)
	+0, +Denormalized value	H'0000 0000	No change
Rsrc < 0	-0, -Denormalized value		
	30 ≥ exp ≥ -126	H'0000 0000 to H'8000 0080	No change (Note 1)
	127 ≥ exp ≥ 31	When EIT occurs: no change	Invalid Operation Exception
	-Infinity	Other EIT: H'8000 0000	(Note 2)
NaN	QNaN	When EIT occurs: no change Other EIT:	Invalid Operation Exception
	SNaN	Signed bit = 0:H'7FFF FFFF Signed bit = 1:H'8000 0000	

Note 1: Inexact Exception occurs when rounding is performed.
2: Inexact Exception does not occur when Rsrc = H'CF00 0000.

FTOS

floating-point Instructions

Float to short

[M32R-FPU Extended Instruction]

FTOS

[Mnemonic]

```
FTOS  Rdest,Rsrc
```

[Function]

Convert the floating-point single precision value to 16-bit integer.

Rdest = (signed int) Rsrc ;

[Description]

Convert the floating-point single precision value stored in Rsrc to a 16-bit integer and store the result in Rdest.

The result is rounded toward 0 regardless of the value in the RM field of FPSR. The condition bit (C) remains unchanged.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Unimplemented Operation Exception (UIPL)
- Invalid Operation Exception (IVLD)
- Inexact Exception (IXCT)

[Encoding]

1101	src	0000	0000	0100	dest	1100	0000

```
FTOS  Rdest,Rsrc
```

FTOS

floating point Instructions

Float to short

[M32R-FPU Extended Instruction]

[Supplemental Operation Description]

The results of the FTOS instruction executed based on the Rsrc value, both when DN = 0 and DN = 1, are shown in below.

DN = 0

Rsrc Value (exponent with no bias)		Rdest	Exception
Rsrc ≥ 0	+Infinity	When EIT occurs: no change	Invalid Operation Exception
	127 ≥ exp ≥ 15	Other EIT: H'0000 7FFFF	
	14 ≥ exp ≥ -126	H'0000 0000 to H'0000 7FFF	No change (Note 1)
	+Denormalized value	No change	Unimplemented Exception
	+0	H'0000 0000	No change
Rsrc < 0	-0		
	-Denormalized value	No change	Unimplemented Exception
	14 ≥ exp ≥ -126	H'0000 0000 to H'FFFF 8001	No change (Note 1)
	127 ≥ exp ≥ 15	When EIT occurs: no change	Invalid Operation Exception
	-Infinity	Other EIT: H'FFFF 8000	(Note 2)
NaN	QNaN	When EIT occurs: no change Other EIT:	Invalid Operation Exception
	SNaN	Signed bit = 0:H'0000 7FFF Signed bit = 1:H'FFFF 8000	

Note 1: Inexact Exception occurs when rounding is performed.
2: Inexact Exception does not occur when Rsrc = H'CF00 0000.

DN = 1

Rsrc Value (exponent with no bias)		Rdest	Exception
Rsrc ≥ 0	+Infinity	When EIT occurs: no change	Invalid Operation Exception
	127 ≥ exp ≥ 15	Other EIT: H'0000 7FFF	
	14 ≥ exp ≥ -126	H'0000 0000 to H'0000 7FFF	No change (Note 1)
	+0, +Denormalized value	H'0000 0000	No change
Rsrc < 0	-0, -Denormalized value		
	14 ≥ exp ≥ -126	H'0000 0000 to H'FFFF 8001	No change (Note 1)
	127 ≥ exp ≥ 15	When EIT occurs: no change	Invalid Operation Exception
	-Infinity	Other EIT: H'FFFF 8000	(Note 2)
NaN	QNaN	When EIT occurs: no change Other EIT:	Invalid Operation Exception
	SNaN	Signed bit = 0:H'0000 7FFF Signed bit = 1:H'FFFF 8000	

Note 1: Inexact Exception occurs when rounding is performed.
2: No Exceptions occur when Rsrc = H'C700 0000. When Rsrc = H'C700 0001 to H'C700 00FF, the Inexact Exception occurs and the Invalid Operation Exception does not occur.

ITOF

floating-point Instructions

Integer to float

[M32R-FPU Extended Instruction]

ITOF

[Mnemonic]

ITOF Rdest,Rsrc

[Function]

Convert the integer to a floating-point single precision value.

Rdes = (float) Rsrc ;

[Description]

Converts the 32-bit integer stored in Rsrc to a floating-point single precision value and stores the result in Rdest. The result is rounded according to the RM field of FPSR. The condition bit (C) remains unchanged. H'0000 0000 is handled as "+0" regardless of the Rounding Mode.

[EIT occurrence]

Floating-Point Exceptions (FPE)

- Inexact Exception (IXCT)

[Encoding]

1101	src	0000	0000	0100	dest	0000	0000

ITOF Rdest,Rsrc

JL

branch instruction

Jump and link

[Mnemonic]

```
JL   Rsrc
```

[Function]

Subroutine call (register direct)
R14 = (PC & 0xfffffffc) + 4;
PC = Rsrc & 0xfffffffc;

[Description]

JL causes an unconditional jump to the address specified by Rsrc and puts the return address in R14.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	1110	1100	src	JL Rsrc

JMP *branch instruction* Jump JMP

[Mnemonic]

`JMP   Rsrc`

[Function]

Jump
PC = Rsrc & 0xfffffffc;

[Description]

JMP causes an unconditional jump to the address specified by Rsrc.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	1111	1100	src	JMP Rsrc

LD

load/store instruction

Load

[Mnemonic]

```
(1)  LD  Rdest,@Rsrc
(2)  LD  Rdest,@Rsrc+
(3)  LD  Rdest,@(disp16,Rsrc)
```

[Function]

Load to register from the contents of the memory.

(1) Rdest = *(int *) Rsrc;
(2) Rdest = *(int *) Rsrc, Rsrc += 4;
(3) Rdest = *(int *) (Rsrc + (signed short) disp16);

[Description]

(1) The contents of the memory at the address specified by Rsrc are loaded into Rdest.
(2) The contents of the memory at the address specified by Rsrc are loaded into Rdest. Rsrc is post incremented by 4.
(3) The contents of the memory at the address specified by Rsrc combined with the 16-bit displacement are loaded into Rdest.
The displacement value is sign-extended to 32 bits before the address calculation.
The condition bit (C) is unchanged.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	dest	1100	src		LD Rdest,@Rsrc
0010	dest	1110	src		LD Rdest,@Rsrc+
1010	dest	1100	src	disp16	

LD Rdest,@(disp16,Rsrc)

LD24 — *load/store instruction* — Load 24-bit immediate — LD24

[Mnemonic]

`LD24 Rdest,#imm24`

[Function]

Load the 24-bit immediate value into register.
Rdest = imm24 & 0x00ffffff;

[Description]

LD24 loads the 24-bit immediate value into Rdest. The immediate value is zero-extended to 32 bits.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1110	dest		imm24

`LD24 Rdest,#imm24`

LDB

load/store instruction

Load byte

LDB

[Mnemonic]

```
(1)  LDB  Rdest,@Rsrc
(2)  LDB  Rdest,@(disp16,Rsrc)
```

[Function]

Load to register from the contents of the memory.

(1) Rdest = *(signed char *) Rsrc;

(2) Rdest = *(signed char *) (Rsrc + (signed short) disp16);

[Description]

(1) LDB sign-extends the byte data of the memory at the address specified by Rsrc and loads it into Rdest.

(2) LDB sign-extends the byte data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest.
The displacement value is sign-extended to 32 bits before the address calculation.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0010	dest	1000	src	LDB Rdest,@Rsrc

1010	dest	1000	src	disp16

LDB Rdest,@(disp16,Rsrc)

LDH

load/store instruction

Load halfword

LDH

[Mnemonic]

```
(1)  LDH  Rdest,@Rsrc
(2)  LDH  Rdest,@(disp16,Rsrc)
```

[Function]

Load to register from the contents of the memory.

(1) Rdest = *(signed short *) Rsrc;
(2) Rdest = *(signed short *) (Rsrc + (signed short) disp16);

[Description]

(1) LDH sign-extends the halfword data of the memory at the address specified by Rsrc and loads it into Rdest.

(2) LDH sign-extends the halfword data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest.

The displacement value is sign-extended to 32 bits before the address calculation.

The condition bit (C) is unchanged.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	dest	1010	src

LDH Rdest,@Rsrc

1010	dest	1010	src	disp16

LDH Rdest,@(disp16,Rsrc)

LDI *transfer instruction* Load immediate LDI

[Mnemonic]

```
(1)  LDI  Rdest,#imm8
(2)  LDI  Rdest,#imm16
```

[Function]

Load the immediate value into register.

(1) Rdest = (signed char) imm8;
(2) Rdest = (signed short) imm16;

[Description]

(1) LDI loads the 8-bit immediate value into Rdest.
The immediate value is sign-extended to 32 bits.
(2) LDI loads the 16-bit immediate value into Rdest.
The immediate value is sign-extended to 32 bits.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0110	dest	imm8	

LDI Rdest,#imm8

1001	dest	1111	0000	imm16

LDI Rdest,#imm16

LDUB

load/store instruction

Load unsigned byte

LDUB

[Mnemonic]

```
(1)  LDUB  Rdest,@Rsrc
(2)  LDUB  Rdest,@(disp16,Rsrc)
```

[Function]

Load to register from the contents of the memory.

(1) Rdest = *(unsigned char *) Rsrc;

(2) Rdest = *(unsigned char *) (Rsrc + (signed short) disp16);

[Description]

(1) LDUB zero-extends the byte data from the memory at the address specified by Rsrc and loads it into Rdest.

(2) LDUB zero-extends the byte data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest.

The displacement value is sign-extended to 32 bits before address calculation.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0010	dest	1001	src	LDUB Rdest,@Rsrc

1010	dest	1001	src	disp16

LDUB Rdest,@(disp16,Rsrc)

LDUH

load/store instruction

Load unsigned halfword

LDUH

[Mnemonic]

```
(1)  LDUH  Rdest,@Rsrc
(2)  LDUH  Rdest,@(disp16,Rsrc)
```

[Function]

Load to register from the contents of the memory.

(1) Rdest = *(unsigned short *) Rsrc;

(2) Rdest = *(unsigned short *) (Rsrc + (signed short) disp16);

[Description]

(1) LDUH zero-extends the halfword data from the memory at the address specified by Rsrc and loads it into Rdest.

(2) LDUH zero-extends the halfword data in memory at the address specified by Rsrc combined

with the 16-bit displacement, and loads it into Rdest.

The displacement value is sign-extended to 32 bits before the address calculation.

The condition bit (C) is unchanged.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	dest	1011	src	**LDUH Rdest,@Rsrc**

1010	dest	1011	src	disp16

LDUH Rdest,@(disp16,Rsrc)

LOCK

load/store instruction

Load locked

[Mnemonic]

LOCK Rdest,@Rsrc

[Function]

Load locked
LOCK = 1, Rdest = *(int *) Rsrc;

[Description]

The contents of the word at the memory location specified by Rsrc are loaded into Rdest. The condition bit (C) is unchanged.
This instruction sets the LOCK bit in addition to simple loading.
When the LOCK bit is 1, external bus master access is not accepted.
The LOCK bit is cleared by executing the UNLOCK instruction.
The LOCK bit is located in the CPU and operates based on the LOCK and UNLOCK instructions. The user cannot directly read or write to this bit.

The LOCK bit is internal to the CPU and is the control bit for receiving all bus right requests from circuits other than the CPU.
Refer to the Users Manual for non-CPU bus right requests, as the handling differs according to the type of MCU.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	dest	1101	src

LOCK Rdest,@Rsrc

MACHI *DSP function instruction* Multiply-accumulate high-order halfwords MACHI

[Mnemonic]

`MACHI  Rsrc1,Rsrc2`

[Function]

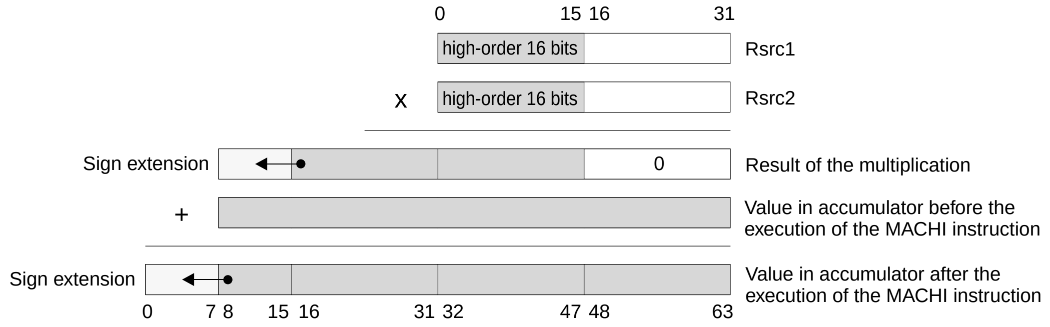
Multiply and add
accumulator += ((signed) (Rsrc1 & 0xffff0000) * (signed short) (Rsrc2 >> 16));

[Description]

MACHI multiplies the high-order 16 bits of Rsrc1 and the high-order 16 bits of Rsrc2, then adds the result to the low-order 56 bits in the accumulator.

The LSB of the multiplication result is aligned with bit 47 in the accumulator, and the portion corresponding to bits 8 through 15 of the accumulator is sign-extended before addition. The result of the addition is stored in the accumulator. The high-order 16 bits of Rsrc1 and Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0100	src2

`MACHI  Rsrc1,Rsrc2`

MACLO *DSP function instruction* Multiply-accumulate low-order halfwords MACLO

[Mnemonic]

```
MACLO  Rsrc1,Rsrc2
```

[Function]

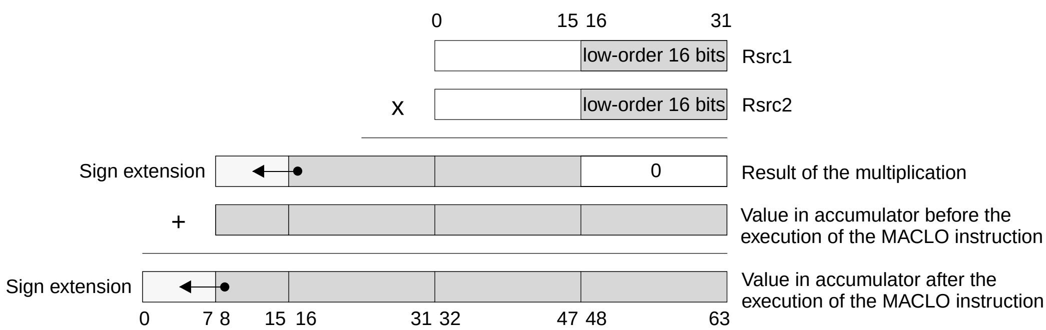
Multiply and add
accumulator += ((signed) (Rsrc1 << 16) * (signed short) Rsrc2) ;

[Description]

MACLO multiplies the low-order 16 bits of Rsrc1 and the low-order 16 bits of Rsrc2, then adds the result to the low order 56 bits in the accumulator.

The LSB of the multiplication result is aligned with bit 47 in the accumulator, and the portion corresponding to bits 8 through 15 of the accumulator is sign-extended before addition. The result of the addition is stored in the accumulator. The low-order 16 bits of Rsrc1 and Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0101	src2

```
MACLO  Rsrc1,Rsrc2
```

MACWHI

DSP function instruction
Multiply-accumulate word and high-order halfword

MACWHI

[Mnemonic]

```
MACWHI  Rsrc1,Rsrc2
```

[Function]

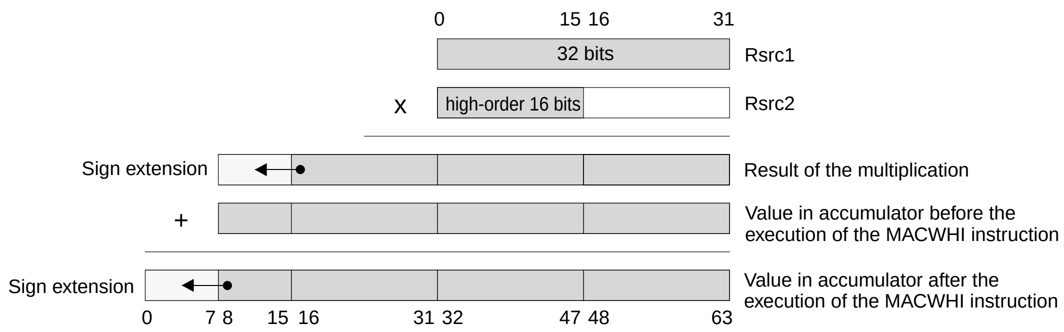
Multiply and add
accumulator += ((signed) Rsrc1 * (signed short) (Rsrc2 >> 16));

[Description]

MACWHI multiplies the 32 bits of Rsrc1 and the high-order 16 bits of Rsrc2, then adds the result to the low-order 56 bits in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 8 through 15 of the accumulator is sign extended before addition. The result of addition is stored in the accumulator. The 32 bits of Rsrc1 and the high-order 16 bits of Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0110	src2

`MACWHI  Rsrc1,Rsrc2`

MACWLO

DSP function instruction

Multiply-accumulate word and low-order halfword

MACWLO

[Mnemonic]

`MACWLO Rsrc1,Rsrc2`

[Function]

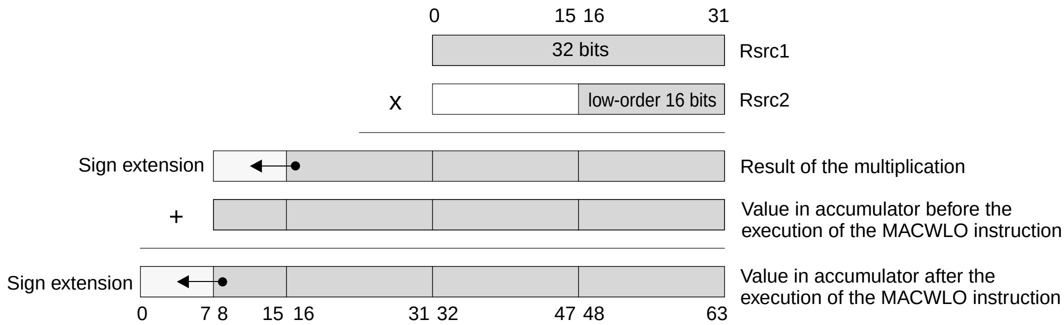
Multiply and add
accumulator += ((signed) Rsrc1 * (signed short) Rsrc2) ;

[Description]

MACWLO multiplies the 32 bits of Rsrc1 and the low-order 16 bits of Rsrc2, then adds the result to the low-order 56 bits in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 8 through 15 of the accumulator is sign-extended before the addition. The result of the addition is stored in the accumulator. The 32 bits Rsrc1 and the low-order 16 bits of Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0111	src2

`MACWLO Rsrc1,Rsrc2`

MUL *multiply and divide instruction* **Multiply** MUL

[Mnemonic]

```
MUL  Rdest,Rsrc
```

[Function]

Multiply
{ signed64bit tmp;
tmp = (signed64bit) Rdest * (signed64bit) Rsrc;
Rdest = (int) tmp;}

[Description]

MUL multiplies Rdest by Rsrc and puts the result in Rdest.
The operands are treated as signed values.
The contents of the accumulator are destroyed by this instruction. The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	0110	src

MUL Rdest,Rsrc

MULHI

DSP function instruction

Multiply high-order halfwords

MULHI

[Mnemonic]

`MULHI  Rsrc1,Rsrc2`

[Function]

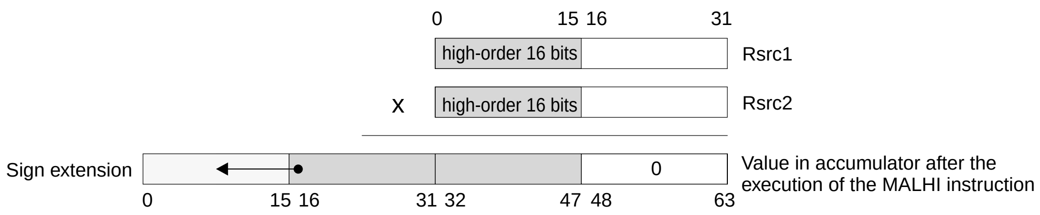
Multiply
accumulator = ((signed) (Rsrc1 & 0xffff000) * (signed short) (Rsrc2 >> 16));

[Description]

MULHI multiplies the high-order 16 bits of Rsrc1 and the high-order 16 bits of Rsrc2, and stores the result in the accumulator.

However, the LSB of the multiplication result is aligned with bit 47 in the accumulator, and the portion corresponding to bits 0 through 15 of the accumulator is sign-extended. Bits 48 through 63 of the accumulator are cleared to 0. The high-order 16 bits of Rsrc1 and Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0000	src2

`MULHI  Rsrc1,Rsrc2`

MULLO

DSP function instruction

Multiply low-order halfwords

MULLO

[Mnemonic]

`MULLO   Rsrc1,Rsrc2`

[Function]

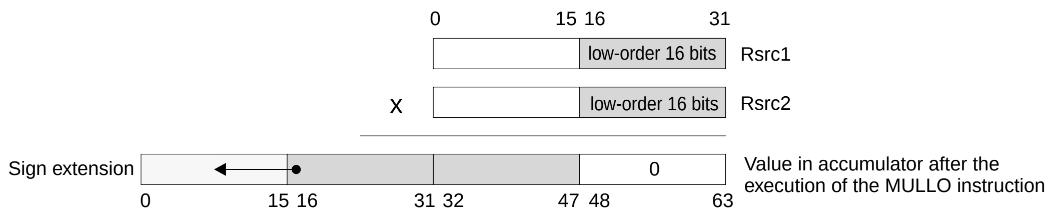
Multiply
accumulator = ((signed) (Rsrc1 << 16) * (signed short) Rsrc2);

[Description]

MULLO multiplies the low-order 16 bits of Rsrc1 and the low-order 16 bits of Rsrc2, and stores the result in the accumulator.

The LSB of the multiplication result is aligned with bit 47 in the accumulator, and the portion corresponding to bits 0 through 15 of the accumulator is sign extended. Bits 48 through 63 of the accumulator are cleared to 0. The low-order 16 bits of Rsrc1 and Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0001	src2

`MULLO   Rsrc1,Rsrc2`

MULWHI

DSP function instruction

Multiply word and high-order halfword

MULWHI

[Mnemonic]

`MULWHI Rsrc1,Rsrc2`

[Function]

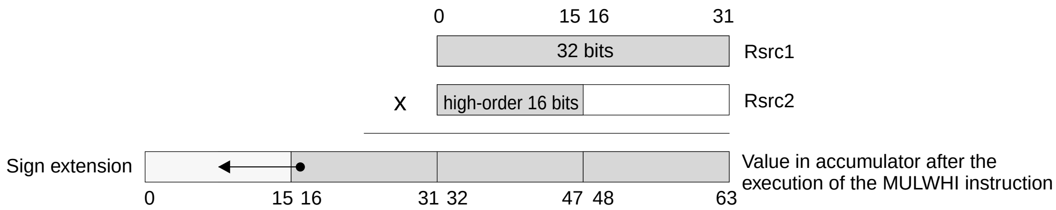
Multiply
accumulator = ((signed) Rsrc1 * (signed short) (Rsrc2 >> 16));

[Description]

MULWHI multiplies the 32 bits of Rsrc1 and the high-order 16 bits of Rsrc2, and stores the result in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 0 through 15 of the accumulator is sign-extended. The 32 bits of Rsrc1 and high-order 16 bits of Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0010	src2

`MULWHI Rsrc1,Rsrc2`

MULWLO

DSP fucntion instruction
Multiply word and low-order halfword

MULWLO

[Mnemonic]

```
MULWLO  Rsrc1,Rsrc2
```

[Function]

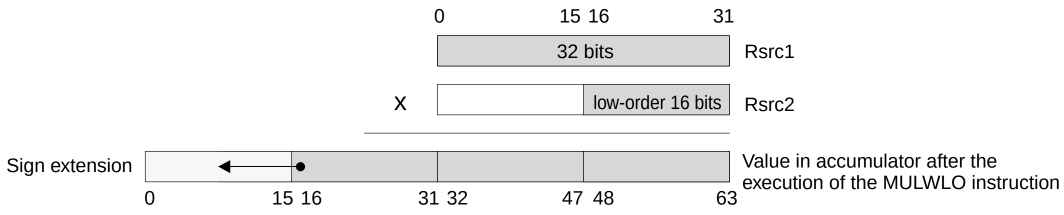
Multiply
accumulator = ((signed) Rsrc1 * (signed short) Rsrc2);

[Description]

MULWLO multiplies the 32 bits of Rsrc1 and the low-order 16 bits of Rsrc2, and stores the result in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 0 through 15 of the accumulator is sign extended. The 32 bits of Rsrc1 and low-order 16 bits of Rsrc2 are treated as signed values.

The condition bit (C) is unchanged.



[EIT occurrence]

None

[Encoding]

0011	src1	0011	src2

```
MULWLO  Rsrc1,Rsrc2
```

MV

transfer instruction

Move register

MV

[Mnemonic]

```
MV  Rdest,Rsrc
```

[Function]

Transfer

Rdest = Rsrc;

[Description]

MV moves Rsrc to Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	1000	src	MV Rdest,Rsrc

MVFACHI

DSP function instruction

Move high-order word from accumulator

MVFACHI

[Mnemonic]

`MVFACHI  Rdest`

[Function]

Transfer from accumulator to register
Rdest = (int) (accumulator >> 32) ;

[Description]

MVFACHI moves the high-order 32 bits of the accumulator to Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	dest	1111	0000

`MVFACHI  Rdest`

MVFACLO

DSP function instruction

Move low-order word from accumulator

MVFACLO

[Mnemonic]

```
MVFACLO  Rdest
```

[Function]

Transfer from accumulator to register
Rdest = (int) accumulator

[Description]

MVFACLO moves the low-order 32 bits of the accumulator to Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	dest	1111	0001	MVFACLO Rdest

MVFACMI

DSP function instruction

Move middle-order word from accumulator

MVFACMI

[Mnemonic]

```
MVFACMI  Rdest
```

[Function]

Transfer from accumulator to register
Rdest = (int) (accumulator >> 16) ;

[Description]

MVFACMI moves bits16 through 47 of the accumulator to Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	dest	1111	0010

MVFACMI Rdest

MVFC

transfer instruction

Move from control register

MVFC

[Mnemonic]

```
MVFC  Rdest,CRsrc
```

[Function]

Transfer from control register to register
Rdest = CRsrc ;

[Description]

MVFC moves CRsrc to Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	1001	src	MVFC Rdest,CRsrc

MVTACHI *DSP function instruction* Move high-order word to accumulator MVTACHI

[Mnemonic]

`MVTACHI  Rsrc`

[Function]

Transfer from register to accumulator
accumulator [0 : 31] = Rsrc ;

[Description]

MVTACHI moves Rsrc to the high-order 32 bits of the accumulator.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	src	0111	0000

`MVTACHI  Rsrc`

MVTACLO

DSP function instruction

Move low-order word to accumulator

MVTACLO

[Mnemonic]

```
MVTACLO  Rsrc
```

[Function]

Transfer from register to accumulator
accumulator [32 : 63] = Rsrc ;

[Description]

MVTACLO moves Rsrc to the low-order 32 bits of the accumulator.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	src	0111	0001	MVTACLO Rsrc

MVTC

transfer instruction

Move to control register

[Mnemonic]

```
MVTC  Rsrc,CRdest
```

[Function]

Transfer from register to control register
CRdest = Rsrc ;

[Description]

MVTC moves Rsrc to CRdest.
If PSW(CR0) is specified as CRdest, the condition bit (C) is changed; otherwise it is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	1010	src

`MVTC  Rsrc,CRdest`

NEG

arithmetic operation instruction

Negate

NEG

[Mnemonic]

```
NEG   Rdest,Rsrc
```

[Function]

Negate

Rdest = 0 – Rsrc ;

[Description]

NEG negates (changes the sign of) Rsrc treated as a signed 32-bit value, and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	0011	src

```
NEG   Rdest,Rsrc
```

NOP

branch instruction

No operation

NOP

[Mnemonic]

NOP

[Function]

No operation
/* */

[Description]

NOP performs no operation. The subsequent instruction then processed.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0111	0000	0000	0000	NOP

NOT *logic operation instruction* Logical NOT NOT

[Mnemonic]

```
NOT  Rdest,Rsrc
```

[Function]

Logical NOT

Rdest = ~ Rsrc ;

[Description]

NOT inverts each of the bits of Rsrc and puts the result in Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	1011	src	NOT Rdest,Rsrc

OR *logic operation instruction* OR **OR**

[Mnemonic]

```
OR  Rdest,Rsrc
```

[Function]

Logical OR

Rdest = Rdest | Rsrc ;

[Description]

OR computes the logical OR of the corresponding bits of Rdest and Rsrc, and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	1110	src

`OR  Rdest,Rsrc`

OR3

logic operation instruction

OR 3-operand

OR3

[Mnemonic]

`OR3 Rdest,Rsrc,#imm16`

[Function]

Logical OR

Rdest = Rsrc | (unsigned short) imm16 ;

[Description]

OR3 computes the logical OR of the corresponding bits of Rsrc and the 16-bit immediate value, which is zero-extended to 32 bits, and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1000	dest	1110	src	imm16

`OR3 Rdest,Rsrc,#imm16`

RAC

DSP function instruction

Round accumulator

RAC

[Mnemonic]

RAC

[Function]

Saturation Process
{ signed64bit tmp;
tmp = (signed64bit) accumulator << 1;
tmp = tmp + 0x0000 0000 0000 8000;
if(0x0000 7fff ffff 0000 < tmp)
    accumulator = 0x0000 7fff ffff 0000;
else if(tmp < 0xffff 8000 0000 0000)
    accumulator = 0xffff 8000 0000 0000;
else
    accumulator = tmp & 0xffff ffff ffff 0000; }

[Description]

RAC rounds the contents in the accumulator to word size and stores the result in the accumulator.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

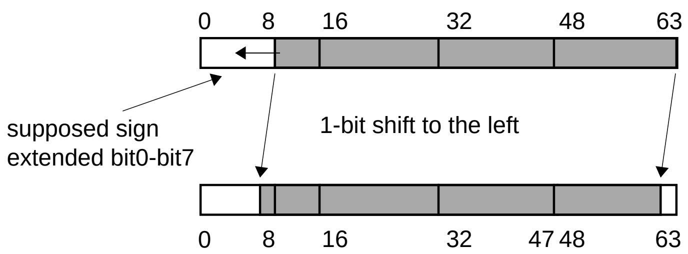
0101	0000	1001	0000	RAC

RAC *DSP function instruction* RAC
Round accumulator

[Supplement]

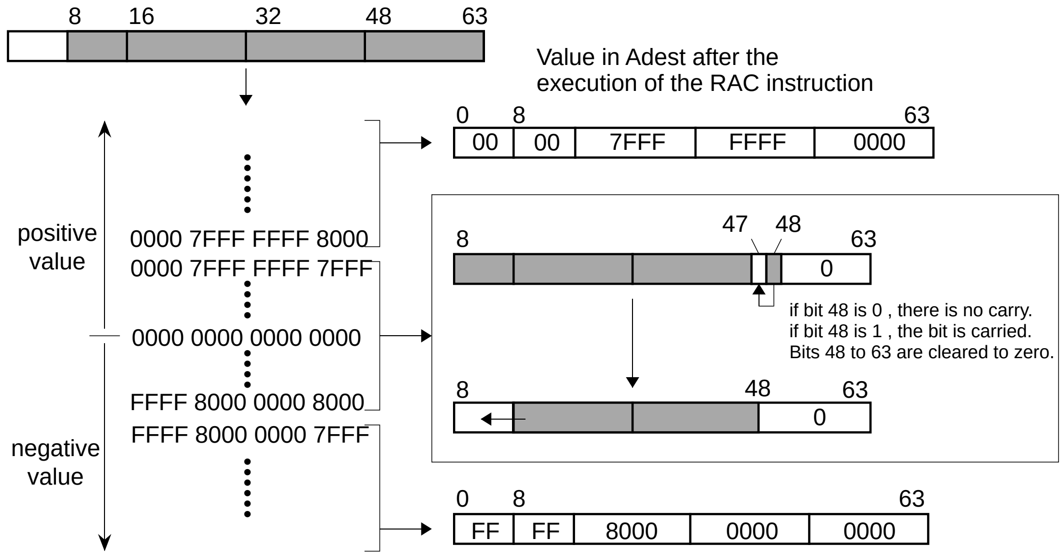
This instruction is executed in two steps as shown below:

<step 1>



<step 2>

The value in the accumulator is altered depending on the supposed bit 80 through 7 after left-shift operation and bit 8 through bit 63 after shift operation.



RACH

DSP function instruction

Round accumulator halfword

RACH

[Mnemonic]

RACH

[Function]

Saturation Process
{ signed64bit tmp;
tmp = (signed64bit) accumulator << 1;
tmp = tmp + 0x0000 0000 8000 0000;
if(0x0000 7fff 0000 0000 < tmp)
    accumulator = 0x0000 7fff 0000 0000;
else if(tmp < 0xffff 8000 0000 0000)
    accumulator = 0xffff 8000 0000 0000;
else
    accumulator = tmp & 0xffff ffff 0000 0000; }

[Description]

RACH rounds the contents in the accumulator to halfword size and stores the result in the accumulator.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	**0000**	**1000**	**0000**	**RACH**

RACH

DSP function instruction

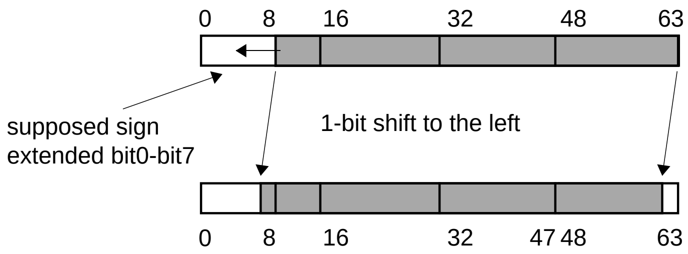
Round accumulator halfword

RACH

[Supplement]

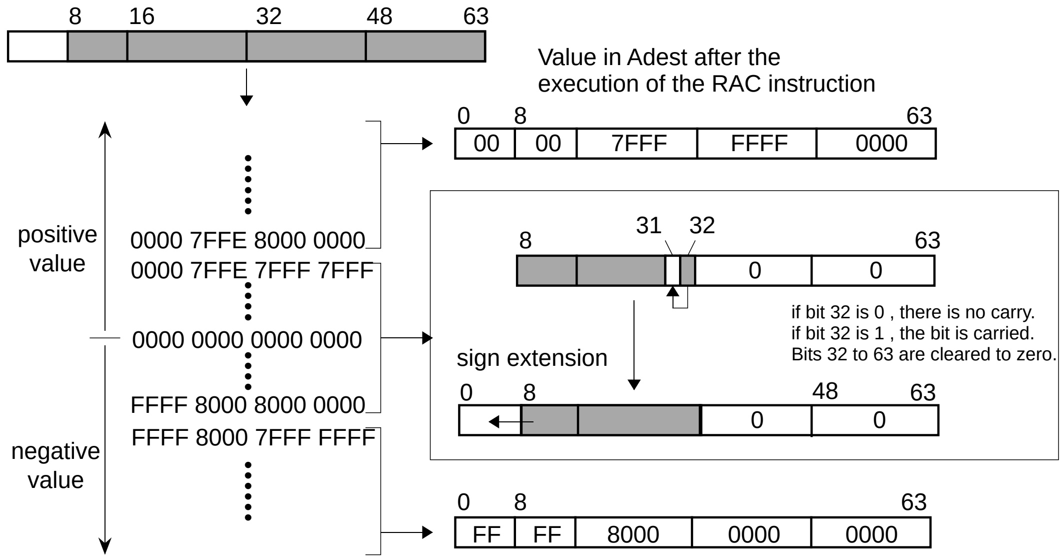
This instruction is executed in two steps, as shown below.

<proccess 1>



<proccess 2>

The value in the accumulator is altered depending on the supposed bit 80 through 7 after left-shift operation and bit 8 through bit 63 after shift operation.



REM

multiply and divide instruction

Remainder

REM

[Mnemonic]

```
REM   Rdest,Rsrc
```

[Function]

Signed remainder
Rdest = (signed) Rdest % (signed) Rsrc ;

[Description]

REM divides Rdest by Rsrc and puts the quotient in Rdest. The operands are treated as signed 32-bit values.

The quotient is rounded toward zero and the quotient takes the same sign as the dividend.

The condition bit (C) is unchanged.

When Rsrc is zero, Rdest is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	0010	src	0000	0000	0000	0000

```
REM   Rdest,Rsrc
```

REMU

multiply and divide instruction

Remainder unsigned

REMU

[Mnemonic]

`REMU Rdest,Rsrc`

[Function]

Unsigned remainder
Rdest = (unsigned) Rdest % (unsigned) Rsrc ;

[Description]

REMU divides Rdest by Rsrc and puts the quotient in Rdest.
The operands are treated as unsigned 32-bit values.
The condition bit (C) is unchanged.
When Rsrc is zero, Rdest is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	0011	src	0000	0000	0000	0000

`REMU Rdest,Rsrc`

RTE

EIT-related instruction

Return from EIT

RTE

[Mnemonic]

RTE

[Function]

Return from EIT

SM = BSM ;
IE = BIE ;
C = BC ;
PC = BPC & 0xfffffffc ;

[Description]

RTE restores the SM, IE and C bits of the PSW from the BSM, BIE and BC bits, and jumps to the address specified by BPC.

At this time, because the BSM, BIE, and BC bits in the PSW register are undefined, the BPC is also undefined.

[EIT occurrence]

None

[Encoding]

0001	**0000**	**1101**	**0110**	**RTE**

SETH

Transfer instructions

Set high-order 16-bit

SETH

[Mnemonic]

```
SETH  Rdest,#imm16
```

[Function]

Transfer instructions

Rdest = (signed short) imm16 << 16 ;

[Description]

SETH load the immediate value into the 16 most significant bits of Rdest.
The 16 least significant bits become zero.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1101	dest	1100	0000	imm16

```
SETH  Rdest,#imm16
```

SETPSW

Bit Operation Instructions

Set PSW

[M32R-FPU Extended Instruction]

[Mnemonic]

SETPSW #imm8

[Function]

Set the undefined SM, IE, anc C bits of PSW to 1.
PSW : = imm8&0x000000ff

[Description]

Set the AND result of the value of b0 (MSB), b1, and b7 (LSB) of the 8-bit immediate value and bits SM, IE, and C of PSW to the corresponding SM, IE, and C bits. When b7 (LSB) or #imm8 is 1, the condition bit (C) goes to 0. All other bits remain unchanged.

[EIT occurrence]

None

[Encoding]

0111	0001	imm8

SETPSW #imm8

[Note]

Set the 8-bit immediate values of b2 to b6 to “0”.

SLL

shift instruction

Shift left logical

SLL

[Mnemonic]

`SLL   Rdest,Rsrc`

[Function]

Logical left shift
Rdest = Rdest << (Rsrc & 31) ;

[Description]

SLL left logical-shifts the contents of Rdest by the number specified by Rsrc, shifting zeroes into the least significant bits.

Only the five least significant bits of Rsrc are used.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	0100	src

`SLL   Rdest,Rsrc`

SLL3 — *shift instruction* — Shift left logical 3-operand — SLL3

[Mnemonic]

```
SLL3  Rdest,Rsrc,#imm16
```

[Function]

Logical left shift
Rdest = Rsrc << (imm16 & 31) ;

[Description]

SLL3 left logical-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, shifting zeroes into the least significant bits.

Only the five least significant bits of the 16-bit immediate value are used.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	1100	src	imm16

```
SLL3  Rdest,Rsrc,#imm16
```

SLLI

shift instruction

Shift left logical immediate

SLLI

[Mnemonic]

```
SLLI  Rdest,#imm5
```

[Function]

Logical left shift
Rdest = Rdest << imm5 ;

[Description]

SLLI left logical-shifts the contents of Rdest by the number specified by the 5-bit immediate value, shifting zeroes into the least significant bits.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	dest	010	imm5

`SLLI  Rdest,#imm5`

SRA

shift instruction

Shift right arithmetic

SRA

[Mnemonic]

```
SRA   Rdest,Rsrc
```

[Function]

Arithmetic right shift
Rdest = (signed) Rdest >> (Rsrc & 31) ;

[Description]

SRA right arithmetic-shifts the contents of Rdest by the number specified by Rsrc, replicates the sign bit in the MSB of Rdest and puts the result in Rdest.

Only the five least significant bits are used.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	0010	src

`SRA   Rdest,Rsrc`

SRA3

shift instruction

Shift right arithmetic 3-operand

SRA3

[Mnemonic]

`SRA3 Rdest,Rsrc,#imm16`

[Function]

Arithmetic right shift
Rdest = (signed) Rsrc >> (imm16 & 31) ;

[Description]

SRA3 right arithmetic-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, replicates the sign bit in Rsrc and puts the result in Rdest.
Only the five least significant bits are used.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	1010	src	imm16

`SRA3 Rdest,Rsrc,#imm16`

SRAI

shift instruction

Shift right arithmetic immediate

SRAI

[Mnemonic]

SRAI Rdest,#imm5

[Function]

Arithmetic right shift
Rdest = (signed) Rdest >> imm5 ;

[Description]

SRAI right arithmetic-shifts the contents of Rdest by the number specified by the 5-bit immediate value, replicates the sign bit in MSB of Rdest and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	dest	001	imm5

SRAI Rdest,#imm5

SRL *shift instruction* Shift right logical SRL

[Mnemonic]

`SRL   Rdest,Rsrc`

[Function]

Logical right shift
Rdest = (unsigned) Rdest >> (Rsrc & 31) ;

[Description]

SRL right logical-shifts the contents of Rdest by the number specified by Rsrc, shifts zeroes into the most significant bits and puts the result in Rdest.
Only the five least significant bits of Rsrc are used.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0001	dest	0000	src

`SRL   Rdest,Rsrc`

SRL3 *shift instruction* Shift right logical 3-operand SRL3

[Mnemonic]

`SRL3 Rdest,Rsrc,#imm16`

[Function]

Logical right shift
Rdest = (unsigned) Rsrc >> (imm16 & 31) ;

[Description]

SRL3 right logical-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, shifts zeroes into the most significant bits. Only the five least significant bits of the immediate value are valid.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1001	dest	1000	src	imm16

`SRL3 Rdest,Rsrc,#imm16`

SRLI

shift instruction

Shift right logical immediate

SRLI

[Mnemonic]

SRLI Rdest,#imm5

[Function]

Logical right shift
Rdest = (unsigned) Rdest >> (imm5 & 31) ;

[Description]

SRLI right arithmetic-shifts Rdest by the number specified by the 5-bit immediate value, shifting zeroes into the most significant bits.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0101	dest	000	imm5

SRLI Rdest,#imm5

ST *load/store instruction* Store ST

[Mnemonic]

```
(1)  ST  Rsrc1,@Rsrc2
(2)  ST  Rsrc1,@+Rsrc2
(3)  ST  Rsrc1,@-Rsrc2
(4)  ST  Rsrc1,@(disp16,Rsrc2)
```

[Function]

Store

(1) * (int *) Rsrc2 = Rsrc1;
(2) Rsrc2 += 4, * (int *) Rsrc2 = Rsrc1;
(3) Rsrc2 -= 4, * (int *) Rsrc2 = Rsrc1;
(4) * (int *) (Rsrc2 + (signed short) disp16) = Rsrc1;

[Description]

(1) ST stores Rsrc1 in the memory at the address specified by Rsrc2.
(2) ST increments Rsrc2 by 4 and stores Rsrc1 in the memory at the address specified by the resultant Rsrc2.
(3) ST decrements Rsrc2 by 4 and stores the contents of Rsrc1 in the memory at the address specified by the resultant Rsrc2.
(4) ST stores Rsrc1 in the memory at the address specified by Rsrc combined with the 16-bit displacement. The displacement value is sign-extended before the address calculation.
The condition bit (C) is unchanged.

[EIT occurrence]

Address exception (AE)

ST

load/store instruction

Store

ST

[Encoding]

0010	src1	0100	src2		ST Rsrc1,@Rsrc2
0010	src1	0110	src2		ST Rsrc1,@+Rsrc2
0010	src1	0111	src2		ST Rsrc1,@-Rsrc2
1010	src1	0100	src2	disp16	

ST Rsrc1,@(disp16,Rsrc2)

STB *load/store instruction* Store byte STB

[Mnemonic]

(1) STB Rsrc1,@Rsrc2
(2) STB Rsrc1,@(disp16,Rsrc2)

[Function]

Store
(1) * (char *) Rsrc2 = Rsrc1;
(2) * (char *) (Rsrc2 + (signed short) disp16) = Rsrc1;

[Description]

(1) STB stores the least significant byte of Rsrc1 in the memory at the address specified by Rsrc2.
(2) STB stores the least significant byte of Rsrc1 in the memory at the address specified by Rsrc
combined with the 16-bit displacement.
The displacement value is sign-extended to 32 bits before the address calculation.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0010	src1	0000	src2	**STB Rsrc1,@Rsrc2**

1010	src1	0000	src2	disp16

STB Rsrc1,@(disp16,Rsrc2)

STH

load/store instruction

Store halfword

[M32R-FPU Extended Mnemonic]

STH

[Mnemonic]

```
(1)  STH  Rsrc1,@Rsrc2
(2)  STH  Rsrc1,@Rsrc2+   [M32R-FPU Extended Mnemonic]
(3)  STH  Rsrc1,@(disp16,Rsrc2)
```

[Function]

Store

(1) * (signed short *) Rsrc2 = Rsrc1;

(2) * (signed short *) Rsrc2 = Rsrc1, Rsrc2 + = 2 ;

(3) * (signed short *) (Rsrc2 + (signed short) disp16) = Rsrc1;

[Description]

(1) STH stores the least significant halfword of Rsrc1 in the memory at the address specified by Rsrc2.

(2) STH stores the LSB halfword of Rsrc1 to the memory of the address specified by Rsrc2, and then increments Rsrc2 by 2.

(3) STH stores the least significant halfword of Rsrc1 in the memory at the address specified by Rsrc combined with the 16-bit displacement. The displacement value is sign-extended to 32 bits before the address calculation.

The condition bit (C) is unchanged.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	src1	0010	src2	STH Rsrc1,@Rsrc2
0010	src1	0011	src2	STH Rsrc1,@Rsrc2+
1010	src1	0010	src2	disp16

STH Rsrc1,@(disp16,Rsrc2)

SUB *arithmetic operation instruction* Subtract SUB

[Mnemonic]

`SUB   Rdest,Rsrc`

[Function]

Subtract

Rdest = Rdest - Rsrc;

[Description]

SUB subtracts Rsrc from Rdest and puts the result in Rdest.
The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	0010	src

`SUB   Rdest,Rsrc`

SUBV

arithmetic operation instruction

Subtract with overflow checking

SUBV

[Mnemonic]

`SUBV  Rdest,Rsrc`

[Function]

Subtract

Rdest = Rdest - Rsrc;

C = overflow ? 1 : 0;

[Description]

SUBV subtracts Rsrc from Rdest and puts the result in Rdest.

The condition bit (C) is set when the subtraction results in overflow; otherwise, it is cleared.

[EIT occurrence]

None

[Encoding]

0000	dest	0000	src

`SUBV  Rdest,Rsrc`

SUBX

arithmetic operation instruction

Subtract with borrow

SUBX

[Mnemonic]

```
SUBX   Rdest,Rsrc
```

[Function]

Subtract

Rdest = (unsigned) Rdest - (unsigned) Rsrc - C;

C = borrow ? 1 : 0;

[Description]

SUBX subtracts Rsrc and C from Rdest and puts the result in Rdest.

The condition bit (C) is set when the subtraction result cannot be represented by a 32-bit unsigned integer; otherwise it is cleared.

[EIT occurrence]

None

[Encoding]

0000	dest	0001	src	SUBX Rdest,Rsrc

TRAP

EIT-related instruction

Trap

[Mnemonic]

TRAP #imm4

[Function]

Trap occurrence
BPC = PC + 4;
BSM = SM;
BIE = IE;
BC = C ;
IE = 0;
C = 0;
call_trap_handler(imm4);

[Description]

TRAP generates a trap with the trap number specified by the 4-bit immediate value.
IE and C bits are cleared to "0".

[EIT occurrence]

Trap (TRAP)

[Encoding]

0001	0000	1111	imm4

TRAP #imm4;

UNLOCK

load/store instruction

Store unlocked

UNLOCK

[Mnemonic]

`UNLOCK  Rsrc1,@Rsrc2`

[Function]

Store unlocked

```
if ( LOCK == 1 ) { * ( int *) Rsrc2 = Rsrc1; }
LOCK = 0;
```

[Description]

When the LOCK bit is 1, the contents of Rsrc1 are stored at the memory location specified by Rsrc2. When the LOCK bit is 0, store operation is not executed. The condition bit (C) is unchanged.

This instruction clears the LOCK bit to 0 in addition to the simple storage operation.

The LOCK bit is internal to the CPU and cannot be accessed except by using the LOCK and UNLOCK instructions.

The user cannot directly read or write to this bit.

The LOCK bit is internal to the CPU and is the control bit for receiving all bus right requests from circuits other than the CPU.

Refer to the Users Manual for non-CPU bus right requests, as the handling differs according to the type of M

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	src1	0101	src2

`UNLOCK  Rsrc1,@Rsrc2`

UTOF

Floating Point Instructions

Unsigned integer to float

[M32R-FPU Extended Instruction]

UTOF

[Mnemonic]

```
UTOF   Rdest,Rsrc
```

[Function]

Convert from unsigned integer to floating-point single precision value.

Rdest = (float) (unsigned int) Rsrc ;

[Description]

UTOF converts the 32-bit unsigned integer stored in Rsrc to a floating-point single precision value, and the result is stored in Rdest. The result is rounded according to the RM field in FPSR. The condition bit (C) remains unchanged.

H’0000 0000 is treated as “+0” regardless of the Rounding Mode.

[EIT occurrence]

Floating-Point Exceptions (FPE)

• Inexact Exception (IXCT)

[Encoding]

1101	src	0000	0000	0100	dest	0100	0000

```
UTOF   Rdest,Rsrc
```

XOR

logic operation instruction

Exclusive OR

XOR

[Mnemonic]

`XOR   Rdest,Rsrc`

[Function]

Exclusive OR

Rdest = (unsigned) Rdest ^ (unsigned) Rsrc;

[Description]

XOR computes the logical XOR of the corresponding bits of Rdest and Rsrc, and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

0000	dest	1101	src

`XOR   Rdest,Rsrc`

XOR3

logic operation instruction

Exclusive OR 3-operand

XOR3

[Mnemonic]

`XOR3 Rdest,Rsrc,#imm16`

[Function]

Exclusive OR

Rdest = (unsigned) Rsrc ^ (unsigned short) imm16;

[Description]

XOR3 computes the logical XOR of the corresponding bits of Rsrc and the 16-bit immediate value, which is zero-extended to 32 bits, and puts the result in Rdest.

The condition bit (C) is unchanged.

[EIT occurrence]

None

[Encoding]

1000	dest	1101	src	imm16

`XOR3 Rdest,Rsrc,#imm16`

APPENDICES

Appendix1 Hexadecimal Instraction Code

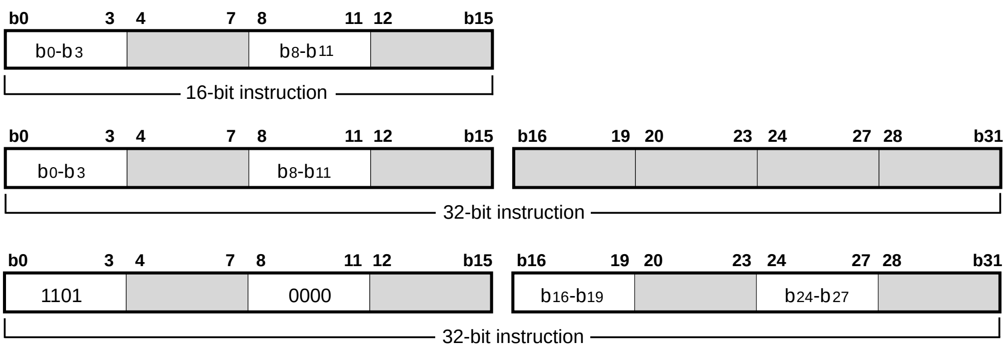
The bit pattern of each instruction and correspondence of mnemonic are shown below.
The instructions enclosed in the bold lines are M32R-FPU extended instructions.

Appendix Table 1.1.1 Instruction Code Table

	b0-b3 \ b8-b11	0000	0001	0010	0011	0100	0101	0110	0111
	hexadecimal numeral	0	1	2	3	4	5	6	7
16-bit instruction	0000 / 0	**SUBV** Rdest,Rsrc	**SUBX** Rdest,Rsrc	**SUB** Rdest,Rsrc	**NEG** Rdest,Rsrc	**CMP** Rsrc1,Rsrc2	**CMPU** Rsrc1,Rsrc2		
16-bit instruction	0001 / 1	**SRL** Rdest,Rsrc		**SRA** Rdest,Rsrc		**SLL** Rdest,Rsrc		**MUL** Rdest,Rsrc	
16-bit instruction	0010 / 2	**STB** Rsrc1,@Rsrc2		**STH** Rsrc1,@Rsrc2	**STH** Rsrc1,@Rsrc2+	**ST** Rsrc1,@Rsrc2	**UNLOCK** Rsrc1,@Rsrc2	**ST** Rsrc1,@+Rsrc2	**ST** Rsrc1,@-Rsrc2
16-bit instruction	0011 / 3	**MULHI** Rsrc1,Rsrc2	**MULLO** Rsrc1,Rsrc2	**MULWHI** Rsrc1,Rsrc2	**MULWLO** Rsrc1,Rsrc2	**MACHI** Rsrc1,Rsrc2	**MACLO** Rsrc1,Rsrc2	**MACWHI** Rsrc1,Rsrc2	**MACWLO** Rsrc1,Rsrc2
16-bit instruction	0100 / 4	**ADDI** Rdest,#imm8							
16-bit instruction	0101 / 5	**SRLI** Rdest,#imm5		**SRAI** Rdest,#imm5		**SLLI** Rdest,#imm5			**MVTACHI, MVTACLO** (*2)
16-bit instruction	0110 / 6	**LDI** Rdest,#imm8							
16-bit instruction	0111 / 7	**NOP** (*1)	**BC, BNC, BL, BRA, SETPSW, CLRPSW** (*1)						
32-bit instruction	1000 / 8					**CMPI** Rsrc,#imm16	**CMPUI** Rsrc,#imm16		
32-bit instruction	1001 / 9	**DIV** Rdest,Rsrc	**DIVU** Rdest,Rsrc	**REM** Rdest,Rsrc	**REMU** Rdest,Rsrc				
32-bit instruction	1010 / A	**STB** Rsrc1,@(disp16,Rsrc2)		**STH** Rsrc1,@(disp16,Rsrc2)		**ST** Rsrc1,@(disp16,Rsrc2)		**BSET** #bitpos,@(disp16,Rsrc)	**BCLR** #bitpos,@(disp16,Rsrc)
32-bit instruction	1011 / B	**BEQ** Rsrc1,Rsrc2,pcdisp16	**BNE** Rsrc1,Rsrc2,pcdisp16						
32-bit instruction	1100 / C								
32-bit instruction	1101 / D	**FPU** externded instruction							
32-bit instruction	1110 / E	**LD24** Rdest,#imm24							
32-bit instruction	1111 / F	**BC, BNC, BL, BRA** (*1)							

FPU extended instruction (b0-b3 = 1101, b8-b11 = 0000)

	b16-b19 \ b24-b27	0000	0001	0010	0011	0100	0101	0110	0111
	hexadecimal numeral	0	1	2	3	4	5	6	7
32-bit instruction	0000 / 0	**FADD**				**FSUB**			
32-bit instruction	0001 / 1	**FMUL**							
32-bit instruction	0010 / 2	**FDIV**							
32-bit instruction	0011 / 3	**FMADO**				**FMSUB**			
32-bit instruction	0100 / 4	**ITOF**				**UTOF**			
32-bit instruction	0101 / 5								
32-bit instruction	0110 / 6								
32-bit instruction	0111 / 7								



1000	1001	1010	1011	1100	1101	1110	1111	b8-b11		
8	9	A	B	C	D	E	F	hexadecimal numeral	b0-b3	
ADDV Rdest,Rsrc	**ADDX** Rdest,Rsrc	**ADD** Rdest,Rsrc	**NOT** Rdest,Rsrc	**AND** Rdest,Rsrc	**XOR** Rdest,Rsrc	**OR** Rdest,Rsrc		0	0000	16-bit instruction
MV Rdest,Rsrc	**MVFC** Rdest,CRsrc	**MVTC** Rsrc,CRdest		**JL, JMP** (✻1)	**RTE**		**TRAP** #imm4	1	0001	16-bit instruction
LDB Rdest,@Rsrc	**LDUB** Rdest,@Rsrc	**LDH** Rdest,@Rsrc	**LDUH** Rdest,@Rsrc	**LD** Rdest,@Rsrc	**LOCK** Rdest,@Rsrc	**LD** Rdest,@Rsrc+		2	0010	16-bit instruction
								3	0011	16-bit instruction
ADDI Rdest,#imm8								4	0100	16-bit instruction
RACH	**RAC**						**MVFACHI, MVFACLO, MVFACMI** (✻2)	5	0101	16-bit instruction
LDI Rdest,#imm8								6	0110	16-bit instruction
BC, BNC, BL, BRA (✻1)								7	0111	16-bit instruction
ADDV3 Rdest,Rsrc,#imm16		**ADD3** Rdest,Rsrc,#imm16		**AND3** Rdest,Rsrc,#imm16	**XOR3** Rdest,Rsrc,#imm16	**OR3** Rdest,Rsrc,#imm16		8	1000	32-bit instruction
SRL3 Rdest,Rsrc,#imm16		**SRA3** Rdest,Rsrc,#imm16		**SLL3** Rdest,Rsrc,#imm16			**LDI** Rdest,#imm16	9	1001	32-bit instruction
LDB Rdest,@(disp16,Rsrc)	**LDUB** Rdest,@(disp16,Rsrc)	**LDH** Rdest,@(disp16,Rsrc)	**LDUH** Rdest,@(disp16,Rsrc)	**LD** Rdest,@(disp16,Rsrc)				A	1010	32-bit instruction
BEQZ Rsrc,pcdisp16	**BNEZ** Rsrc,pcdisp16	**BLTZ** Rsrc,pcdisp16	**BGEZ** Rsrc,pcdisp16	**BLEZ** Rsrc,pcdisp16	**BGTZ** Rsrc,pcdisp16			B	1011	32-bit instruction
								C	1100	32-bit instruction
				SETH Rdest,#imm16				D	1101	32-bit instruction
LD24 Rdest,#imm24								E	1110	32-bit instruction
BC, BNC, BL, BRA (✻1)								F	1111	32-bit instruction

1000	1001	1010	1011	1100	1101	1110	1111	b24-b27		
0	1	2	3	4	5	6	7	hexadecimal numeral	b16-b19	
				FCMP	**FCMPE**			0	0000	32-bit instruction
								1	0001	32-bit instruction
								2	0010	32-bit instruction
								3	0011	32-bit instruction
FTOI				**FTOS**				4	0100	32-bit instruction
								5	0101	32-bit instruction
								6	0110	32-bit instruction
								7	0111	32-bit instruction

Note. In addition to b0-b3, b8-b11, instructions shown the above ✻1, ✻2 in the table are decided by the following bit patterns.
As for details of bit patterns of each instruction, refer to "3.2 Instruction description."
✻1: b4-b7, ✻2: b12-b15

Appendix 2 Instruction List

The M32R-FPU instruction list is shown below (in alphabetical order).

mnemonic		function	condition bit (C)
ADD	Rdest,Rsrc	Rdest = Rdest + Rsrc	–
ADD3	Rdest,Rsrc,#imm16	Rdest = Rsrc + (sh)imm16	–
ADDI	Rdest,#imm8	Rdest = Rdest + (sb)imm8	–
ADDV	Rdest,Rsrc	Rdest = Rdest + Rsrc	change
ADDV3	Rdest,Rsrc,#imm16	Rdest = Rsrc + (sh)imm16	change
ADDX	Rdest,Rsrc	Rdest = Rdest + Rsrc + C	change
AND	Rdest,Rsrc	Rdest = Rdest & Rsrc	–
AND3	Rdest,Rsrc,#imm16	Rdest = Rsrc & (uh)imm16	–
BC	pcdisp8	if(C) PC=PC+((sb)pcdisp8<<2)	–
BC	pcdisp24	if(C) PC=PC+((s24)pcdisp24<<2)	–
BCLR	#bitpos,@(disp16,Rsrc)	*(sb *)(Rsrc + (sh)disp16) & = ~(1<<(7-bitpos))	–
BEQ	Rsrc1,Rsrc2,pcdisp16	if(Rsrc1 == Rsrc2) PC=PC+((sh)pcdisp16<<2)	–
BEQZ	Rsrc,pcdisp16	if(Rsrc == 0) PC=PC+((sh)pcdisp16<<2)	–
BGEZ	Rsrc,pcdisp16	if(Rsrc >= 0) PC=PC+((sh)pcdisp16<<2)	–
BGTZ	Rsrc,pcdisp16	if(Rsrc > 0) PC=PC+((sh)pcdisp16<<2)	–
BL	pcdisp8	R14=PC+4,PC=PC+((sb)pcdisp8<<2)	–
BL	pcdisp24	R14=PC+4,PC=PC+((s24)pcdisp24<<2)	–
BLEZ	Rsrc,pcdisp16	if(Rsrc <= 0) PC=PC+((sh)pcdisp16<<2)	–
BLTZ	Rsrc,pcdisp16	if(Rsrc < 0) PC=PC+((sh)pcdisp16<<2)	–
BNC	pcdisp8	if(!C) PC=PC+((sb)pcdisp8<<2)	–
BNC	pcdisp24	if(!C) PC=PC+((s24)pcdisp24<<2)	–
BNE	Rsrc1,Rsrc2,pcdisp16	if(Rsrc1 != Rsrc2) PC=PC+((sh)pcdisp16<<2)	–
BNEZ	Rsrc,pcdisp16	if(Rsrc != 0) PC=PC+((sh)pcdisp16<<2)	–
BRA	pcdisp8	PC=PC+((sb)pcdisp8<<2)	–
BRA	pcdisp24	PC=PC+((s24)pcdisp24<<2)	–
BSET	#bitpos,@(disp16,Rsrc)	*(sb *)(Rsrc + (sh)disp16) | = (1<<(7-bitpos))	–
BTST	#bitpos,Rsrc	(Rsrc>>(7-bitpos))&1	change
CLRPSW	#imm8	PSW & = ~imm8 | 0xffffff00	change
CMP	Rsrc1,Rsrc2	(s)Rsrc1 < (s)Rsrc2	change
CMPI	Rsrc,#imm16	(s)Rsrc < (sh)imm16	change
CMPU	Rsrc1,Rsrc2	(u)Rsrc1 < (u)Rsrc2	change
CMPUI	Rsrc,#imm16	(u)Rsrc < (u)((sh)imm16)	change
DIV	Rdest,Rsrc	Rdest = (s)Rdest / (s)Rsrc	–
DIVU	Rdest,Rsrc	Rdest = (u)Rdest / (u)Rsrc	–
FADD	Rdest,Rsrc1,Rsrc2	Rdest = Rsrc1 + Rsrc2	–
FCMP	Rdest,Rsrc1,Rsrc2	Rdest = (Rsrc1 == Rsrc2)?32'h00000000:((Rsrc1< Rsrc2)?{1.31'bx}:{0.31'bx}	–
FCMPE	Rdest,Rsrc1,Rsrc2	FCMP with Exception when unordered	–
FDIV	Rdest,Rsrc1,Rsrc2	Rdest = Rsrc1 / Rsrc2	–

mnemonic		function	condition bit (C)
FMADD	Rdest,Rsrc1,Rsrc2	Rdest = Rdest + Rsrc1 * Rsrc2	–
FMSUB	Rdest,Rsrc1,Rsrc2	Rdest = Rdest - Rsrc1 * Rsrc2	–
FMUL	Rdest,Rsrc1,Rsrc2	Rdest = Rdest * Rsrc2	–
FSUB	Rdest,Rsrc1,Rsrc2	Rdest = Rsrc1 - Rsrc2	–
FTOI	Rdest,Rsrc	Rdest = (s)Rsrc2	–
FTOS	Rdest,Rsrc	Rdest = (sh)Rsrc	–
ITOF	Rdest,Rsrc	Rdest = (float)Rsrc	–
JL	Rsrc	R14 = PC+4, PC = Rsrc	–
JMP	Rsrc	PC = Rsrc	–
LD	Rdest,@(disp16,Rsrc)	Rdest = *(s *)(Rsrc+(sh)disp16)	–
LD	Rdest,@Rsrc	Rdest = *(s *)Rsrc	–
LD	Rdest,@Rsrc+	Rdest = *(s *)Rsrc, Rsrc += 4	–
LD24	Rdest,#imm24	Rdest = imm24 & 0x00ffffff	–
LDB	Rdest,@(disp16,Rsrc)	Rdest = *(sb *)(Rsrc+(sh)disp16)	–
LDB	Rdest,@Rsrc	Rdest = *(sb *)Rsrc	–
LDH	Rdest,@(disp16,Rsrc)	Rdest = *(sh *)(Rsrc+(sh)disp16)	–
LDH	Rdest,@Rsrc	Rdest = *(sh *)Rsrc	–
LDI	Rdest,#imm16	Rdest = (sh)imm16	–
LDI	Rdest,#imm8	Rdest = (sb)imm8	–
LDUB	Rdest,@(disp16,Rsrc)	Rdest = *(ub *)(Rsrc+(sh)disp16)	–
LDUB	Rdest,@Rsrc	Rdest = *(ub *)Rsrc	–
LDUH	Rdest,@(disp16,Rsrc)	Rdest = *(uh *)(Rsrc+(sh)disp16)	–
LDUH	Rdest,@Rsrc	Rdest = *(ub *)Rsrc	–
LOCK	Rdest,@Rsrc	LOCK = 1, Rdest = *(s *)Rsrc	–
MACHI	Rsrc1,Rsrc2	accumulator += (s)(Rsrc1 & 0xffff0000) * (s)((s)Rsrc2>>16)	–
MACLO	Rsrc1,Rsrc2	accumulator += (s)(Rsrc1<<16) * (sh)Rsrc2	–
MACWHI	Rsrc1,Rsrc2	accumulator += (s)Rsrc1 * (s)((s)Rsrc2>>16)	–
MACWLO	Rsrc1,Rsrc2	accumulator += (s)Rsrc1 * (sh)Rsrc2	–
MUL	Rdest,Rsrc	Rdest = (s)Rdest * (s)Rsrc	–
MULHI	Rsrc1,Rsrc2	accumulator = (s)(Rsrc1 & 0xffff0000) * (s)((s)Rsrc2>>16)	–
MULLO	Rsrc1,Rsrc2	accumulator = (s)(Rsrc1<<16) * (sh)Rsrc2	–
MULWHI	Rsrc1,Rsrc2	accumulator = (s)Rsrc1 * (s)((s)Rsrc2>>16)	–
MULWLO	Rsrc1,Rsrc2	accumulator = (s)Rsrc1 * (sh)Rsrc2	–
MV	Rdest,Rsrc	Rdest = Rsrc	–
MVFACHI	Rdest	Rdest = accumulater >> 32	–
MVFACLO	Rdest	Rdest = accumulator	–
MVFACMI	Rdest	Rdest = accumulator >> 16	–
MVFC	Rdest,CRsrc	Rdest = CRsrc	–
MVTACHI	Rsrc	accumulator[0:31] = Rsrc	–
MVTACLO	Rsrc	accumulator[32:63] = Rsrc	–
MVTC	Rsrc,CRdest	CRdest = Rsrc	change

mnemonic		function	condition bit (C)
NEG	**Rdest,Rsrc**	**Rdest = 0 - Rsrc**	–
NOP		**/*no-operation*/**	–
NOT	**Rdest,Rsrc**	**Rdest = ~Rsrc**	–
OR	**Rdest,Rsrc**	**Rdest = Rdest | Rsrc**	–
OR3	**Rdest,Rsrc,#imm16**	**Rdest = Rsrc | (uh)imm16**	–
RAC		**Round the 32-bit value in the accumulator**	–
RACH		**Round the 16-bit value in the accumulator**	–
REM	**Rdest,Rsrc**	**Rdest = (s)Rdest % (s)Rsrc**	–
REMU	**Rdest,Rsrc**	**Rdest = (u)Rdest % (u)Rsrc**	–
RTE		**PC = BPC & 0xfffffffc,** **PSW[SM,IE,C] = PSW[BSM,BIE,BC]**	change
SETH	**Rdest,#imm16**	**Rdest = imm16 << 16**	–
SETPSW	**#imm8**	**PSW | = imm8&0x000000ff**	change
SLL	**Rdest,Rsrc**	**Rdest = Rdest << (Rsrc & 31)**	–
SLL3	**Rdest,Rsrc,#imm16**	**Rdest = Rsrc << (imm16 & 31)**	–
SLLI	**Rdest,#imm5**	**Rdest = Rdest << imm5**	–
SRA	**Rdest,Rsrc**	**Rdest = (s)Rdest >> (Rsrc & 31)**	–
SRA3	**Rdest,Rsrc,#imm16**	**Rdest = (s)Rsrc >> (imm16 & 31)**	–
SRAI	**Rdest,#imm5**	**Rdest = (s)Rdest >> imm5**	–
SRL	**Rdest,Rsrc**	**Rdest = (u)Rdest >> (Rsrc & 31)**	–
SRL3	**Rdest,Rsrc,#imm16**	**Rdest = (u)Rsrc >> (imm16 & 31)**	–
SRLI	**Rdest,#imm5**	**Rdest = (u)Rdest >> imm5**	–
ST	**Rsrc1,@(disp16,Rsrc2)**	***(s *)(Rsrc2+(sh)disp16) = Rsrc1**	–
ST	**Rsrc1,@+Rsrc2**	**Rsrc2 += 4, *(s *)Rsrc2 = Rsrc1**	–
ST	**Rsrc1,@-Rsrc2**	**Rsrc2 -= 4, *(s *)Rsrc2 = Rsrc1**	–
ST	**Rsrc1,@Rsrc2**	***(s *)Rsrc2 = Rsrc1**	–
STB	**Rsrc1,@(disp16,Rsrc2)**	***(sb *)(Rsrc2+(sh)disp16) = Rsrc1**	–
STB	**Rsrc1,@Rsrc2**	***(sb *)Rsrc2 = Rsrc1**	–
STH	**Rsrc1,@(disp16,Rsrc2)**	***(sh *)(Rsrc2+(sh)disp16) = Rsrc1**	–
STH	**Rsrc1,@Rsrc2**	***(sh *)Rsrc2 = Rsrc1**	–
STH	**Rsrc1,@Rsrc2+**	***(sh *)Rsrc2 = Rsrc1, Rsrc2 += 2**	–
SUB	**Rdest,Rsrc**	**Rdest = Rdest - Rsrc**	–
SUBV	**Rdest,Rsrc**	**Rdest = Rdest - Rsrc**	change
SUBX	**Rdest,Rsrc**	**Rdest = Rdest - Rsrc - C**	change
TRAP	**#n**	**PSW[BSM,BIE,BC] = PSW[SM,IE,C]** **PSW[SM,IE,C] = PSW[SM,0,0]** **Call trap-handler number-n**	change
UNLOCK	**Rsrc1,@Rsrc2**	**if(LOCK) { *(s *)Rsrc2 = Rsrc1; } LOCK=0**	–
UTOF	**Rdest,Rsrc**	**Rdest = (float)(unsigned int) Rsrc;**	–
XOR	**Rdest,Rsrc**	**Rdest = Rdest ^ Rsrc**	–
XOR3	**Rdest,Rsrc,#imm16**	**Rdest = Rsrc ^ (uh)imm16**	–

```
where:
typedef singed int      s;  /* 32 bit signed integer (word)*/
typedef unsigned int    u;  /* 32 bit unsigned integer (word)*/
typedef signed short    sh; /* 16 bit signed integer (halfword)*/
typedef unsigned short  uh; /* 16 bit unsigned integer (halfword)*/
typedef signed char     sb; /*  8 bit signed integer (byte)*/
typedef unsigned char   ub; /*  8 bit unsigned integer (byte)*/
```

Appendix 3 Pipeline Processing

Appendix 3.1 Instructions and Pipeline Processing

Appendix Figure 3.1.1 shows each instruction type and the pipeline process.

■ Load/Store instruction

6 stages

Pipeline Stage	IF	D	E	MEM1	MEM2	WB

*The number of cycles required by the MEM1 stage varies according to the access, but the MEM2 stage is normally executed in 1 cycle.

■ All other integer instructions

4 stages

Pipeline Stage	IF	D	E	WB

*Multi-cycle instructions such as the multiply instruction are executed in multiple cycles in the E stage.

Pipeline Stage	IF	D	E	••••••	E	WB

FPU instruction (excluding FMADD, FMSUB)

5 stages

Pipeline Stage	IF	D	E1	E2	WB

*The E1 and E2 stages cannot be executed at the same time as the E stage.
*The E1 stage of the FDIV instruction requires 13 cycles.

■ FPU instruction (FMADD, FMSUB)

6 stages

Pipeline Stage	IF	D	EM	EA	E2	WB

* The EM and EA stages cannot be executed at the same time as the E or E1 stage.

*Operation stages with the same name cannot be executed at the same time. In general, stages with different names can be executed in parallel, but the following combinations are not acceptable.

- E stage executed with E1, E2, EM or EA stage.
- E1 stage executed with EM or EA stage.

*Bypass process: When using the result of one instruction in a subsequent instruction, the first result may bypass the register file and be sent on to the execution stage of the subsequent instruction. The following is an example of a bypass process:

- E stage continuing to WB stage → E, E1, EM stages
- MEM2 stage continuing to WB stage → E, E1, EM, EA stages

Appendix Figure 3.1.1 Instructions and Pipeline Process

The overview of each pipeline stage is shown below.

- IF stage (instruction fetch stage)
  The instruction fetch (IF) is processed in this stage. There is an instruction queue and instructions are fetched until the queue is full regardless of the completion of decoding in the D stage.
  If there is an instruction already in the instruction queue, the instruction read out of the instruction queue is passed to the instruction decoder.

- D stage (decode stage)
  Instruction decoding is processed in the first half of the D stage (DEC1).
  The subsequent instruction decoding (DEC2) and a register fetch (RF) is processed in the second half of the stage.

- E stage (execution stage)
  Operations and address calculations (OP) are processed in the E stage.
  If an operation result from the previous instruction is required, bypass process (BYP) is performed in the first half of the E stage.

- E1, EM, EA stage (execution stage)
  These are the initial stages for execution of the FPU instructions. The EM and EA stages only use instructions FMADD and FMSUB. All other instructions are used in the E1stage

- E2 stage (execution stage)
  This is the secondary stage for the execution of FPU instructions and mainly rounding is performed.

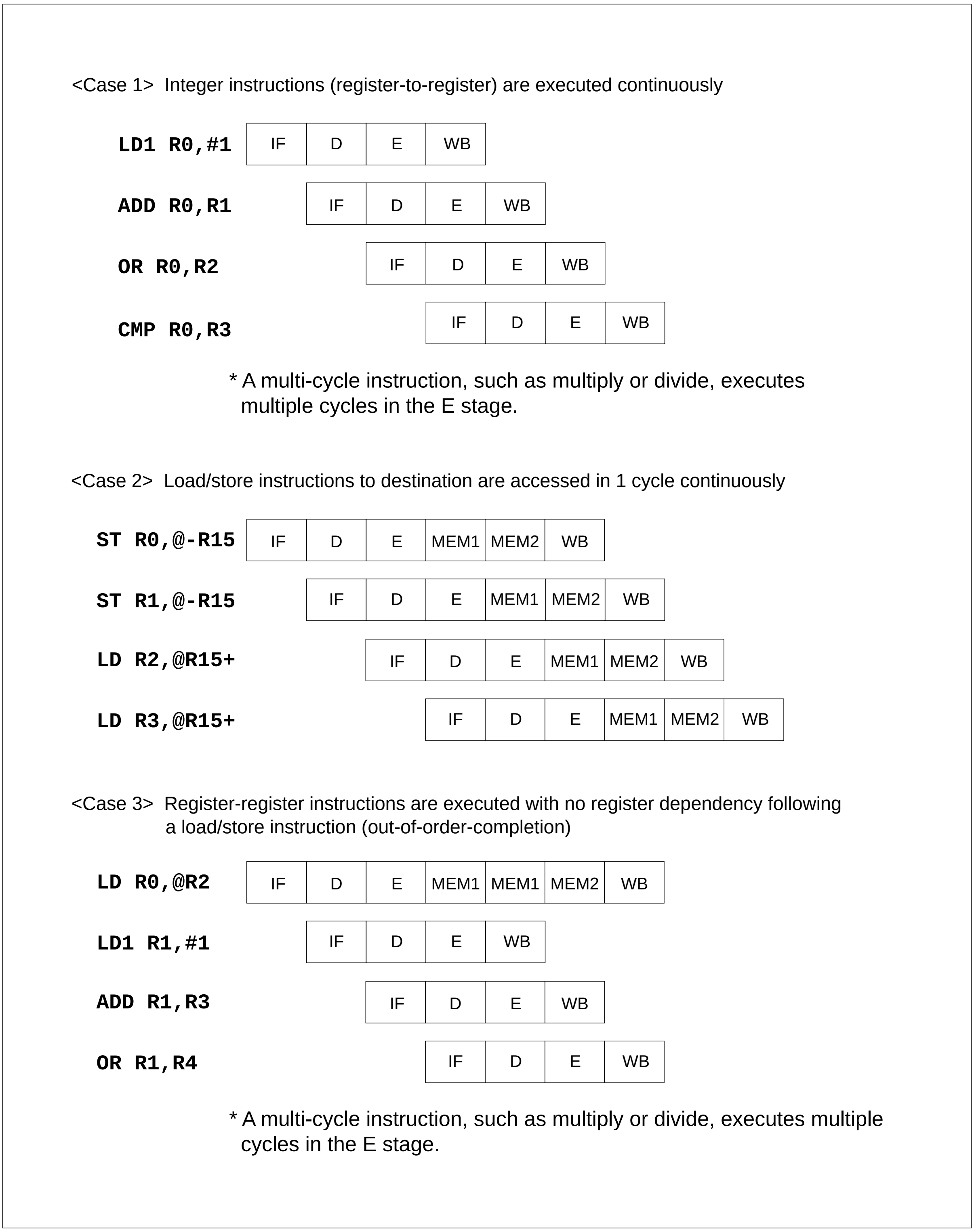
- MEM stage (memory access stage)
  Operand accesses (OA) are processed in the MEM stage. This stage is used only when the load/store instruction is executed.

- WB stage (write back stage)
  The operation results and fetched data are written to the registers in the WB stage.

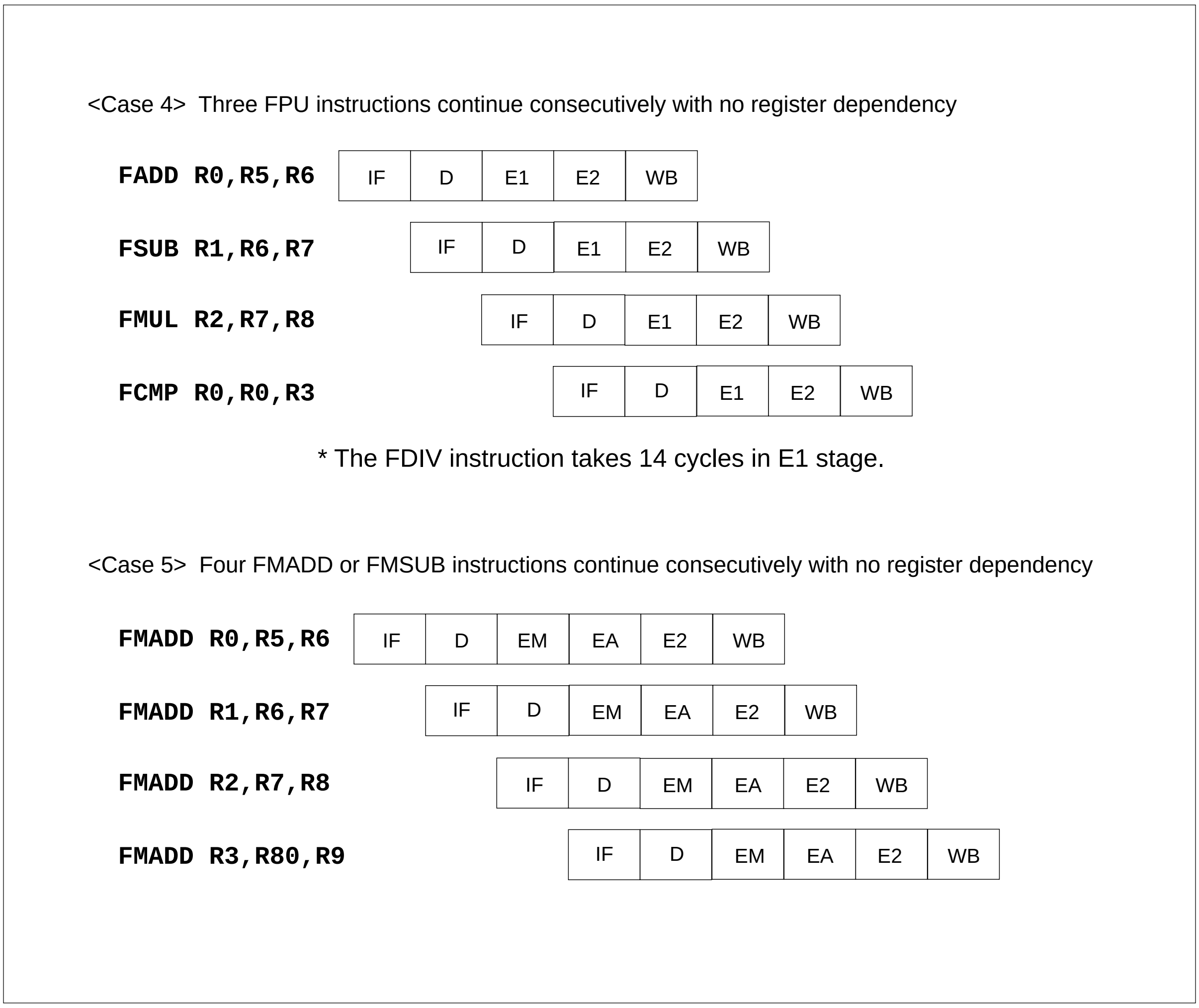
Appendix 3.2 Pipeline Basic Operation

(1) Pipeline Flow with no Stall

The following diagram shows an ideal pipeline flow that has no stall and executes each instruction in 1 clock cycle. (Since this is just an ideal case, all instructions may not be piplined in.)



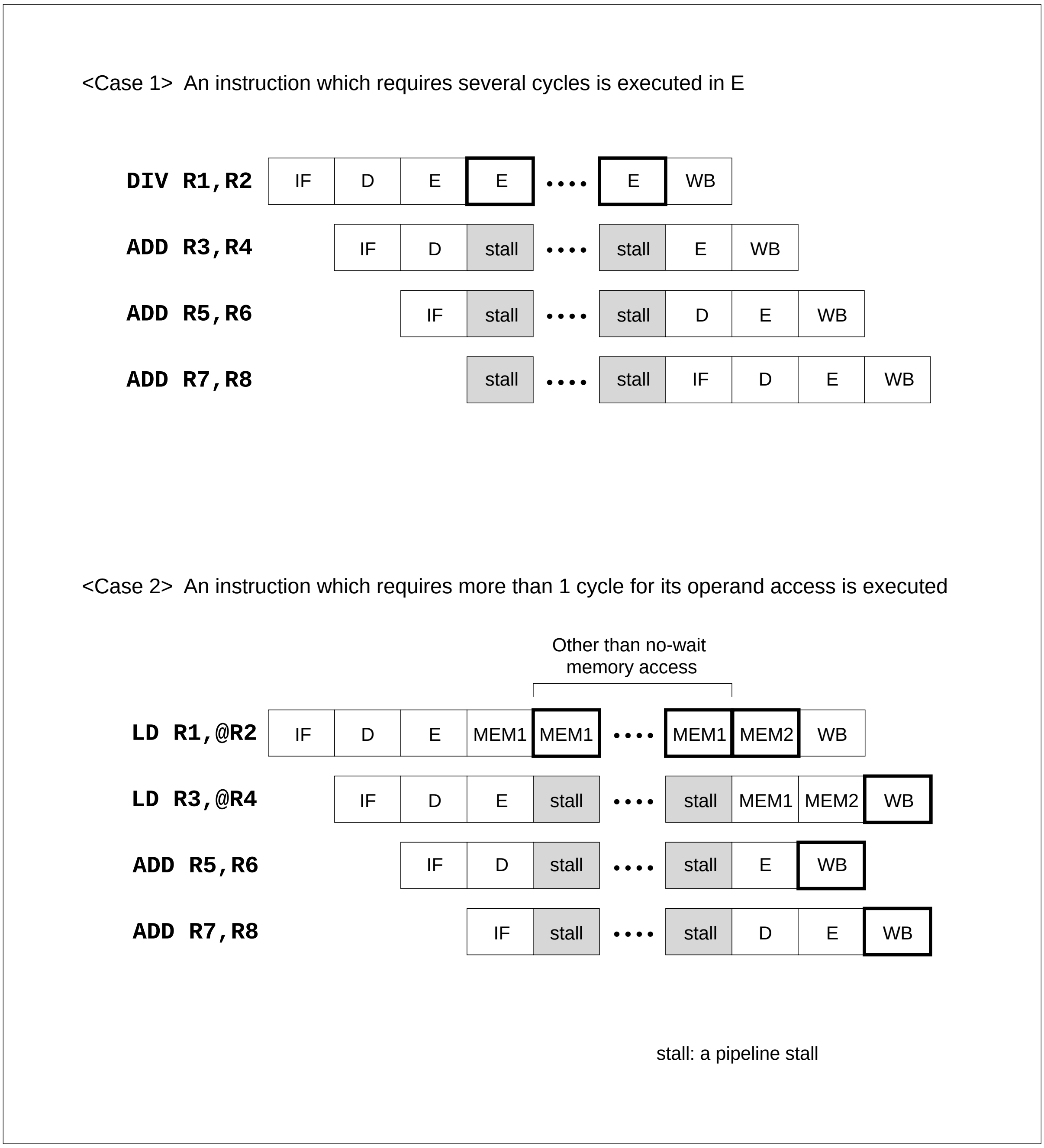
Appendix Figure 3.2.1 Pipeline Flow with no Stall (1)



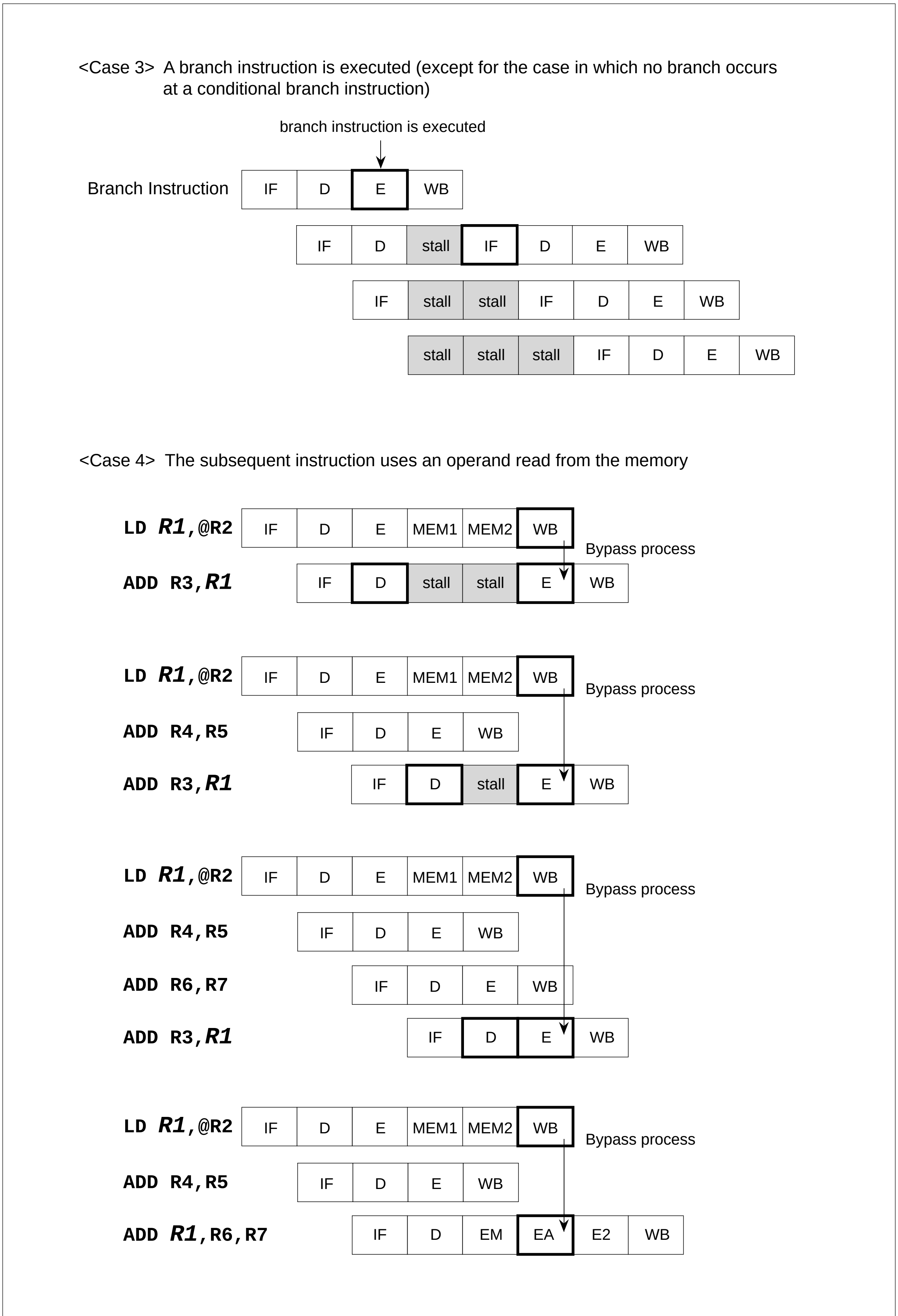
Appendix Figure 3.2.2 Pipeline Flow with no Stall (2)

(2) Pipeline Flow with Stalls

A pipeline stage may stall due to execution of a process or branch instruction. The following diagrams show typical stall cases.



Appendix Figure 3.2.3 Pipeline Flow with Stalls (1)



Appendix Figure 3.2.4 Pipeline Flow with Stalls (2)

<Case 5> The PSW is written by an MVTC, SETPSW, or CLRPSW instruction and the subsequent instruction reads R15

MVTC R1,*PSW*	IF	D	E	WB		
SUB R3,*R15*		IF	D	stall	E	WB

<Case 6> FPSR is accessed by an MVFC instruction after the FPU instruction is executed

FADD R0,R1,R2	IF	D	E1	E2	WB		
MVFC R3,*FPSR*		IF	D	stall	stall	E	WB

<Case 7> The operation result of the FPU instruction is used by the subsequent instruction

FADD *R0*,R1,R2	IF	D	E1	E2	WB			
FADD R3,*R0*,R4		IF	D	stall	stall	E1	E2	WB

FMADD *R0*,R1,R2	IF	D	EM	EA	E2	WB			
FMADD *R0*,R3,R4		IF	D	stall	stall	EM	EA	E2	WB

FMADD *R0*,R1,R2	IF	D	EM	EA	E2	WB				
FMADD R3,*R0*,R4		IF	D	stall	stall	stall	EM	EA	E2	WB

Appendix Figure 3.2.5 Pipeline Flow with Stalls (3)

<Case 8> The FPU and integer instructions run consecutively (with no register dependency)

ADD R0,R1	IF	D	E	WB					
FADD R2,R3,R4		IF	D	E1	E2	WB			
ADD R5,R6			IF	D	stall	E	WB		
FADD R7,R8,R9				IF	stall	D	E1	E2	WB

<Case 9> The FPU and integer instructions run consecutively (with register dependency)

ADD *R0*,R1	IF	D	E	**WB**						
FADD *R0*,*R0*,R4		IF	D	**E1**	E2	WB				
ADD *R0*,R6			IF	D	stall	stall	E	**WB**		
FADD *R0*,*R0*,R9				IF	stall	stall	D	**E1**	E2	WB

Bypass process

Bypass process

<Case 10> The FMADD/FMSUB instructions run consecutively with the integer instruction
(with no register dependency)

ADD R0,R1	IF	D	E	WB							
FMADD R2,R3,R4		IF	D	EM	EA	E2	WB				
ADD R5,R6			IF	D	stall	stall	E	WB			
FMADD R7,R8,R9				IF	stall	stall	D	EM	EA	E2	WB

<Case 11> The FMADD/FMSUB instructions run consecutively with the integer instruction
(with register dependency)

ADD *R0*,R1	IF	D	E	**WB**								
FMADD *R0*,*R0*,R4		IF	D	**EM**	EA	E2	WB					
ADD *R0*,R6			IF	D	stall	stall	stall	E	WB			
FMADD *R0*,R8,R9				IF	stall	stall	stall	D	EM	EA	E2	WB

Bypass process

Appendix Figure 3.2.6 Pipeline Flow with Stalls (4)

<Case 12> The FPU and FMADD/FMSUB instructions run consecutively (with no register dependency)

FADD R0,R1,R10	IF	D	E1	E2	WB					
FMADD R2,R3,R4		IF	D	EM	EA	E2	WB			
FADD R5,R6,R11			IF	D	stall	E1	E2	WB		
FMADD R7,R8,R9				IF	stall	D	EM	EA	E2	WB

<Case 13> The FPU and FMADD/FMSUB instructions run consecutively (with register dependency)

FADD *R0*,R1,R10	IF	D	E1	E2	WB										
FMADD *R0*,*R0*,R4		IF	D	stall	stall	EM	EA	E2	WB						
FADD *R0*,*R0*,R11			IF	stall	stall	D	stall	stall	stall	E1	E2	WB			
FMADD *R0*,R8,R9				stall	stall	IF	stall	stall	stall	D	stall	EM	EA	E2	WB

Appendix Figure 3.2.7 Pipeline Flow with Stalls (5)

Appendix 4 Instruction Execution Time

Normally, the E stage is considered as representing as the instruction execution time, however, because of the pipeline processing the execution time for other stages may effect the total instruction execution time. In particular, the IF, D, and E stages of the subsequent instruction must be considered after a branch has occurred.
The following shows the number of the instruction execution cycles for each pipeline stage.
The execution time of the IF and MEM stages depends on the implementation of each product of the M32R family.
Refer to the user's manual of each product for the execution time of these stages.

Note 1: FPU instruction uses E1 and EM stages.

Appendix Table 4.1.1 Instruction Execution Cycles per Pipeline Stage [excluding FPU instructions]

	the number of execution cycles in each stage					
instruction	IF	D	E	MEM1	MEM2	WB
load instruction (LD, LDB, LDUB, LDH, LDUH, LOCK)	R (note 1)	1	1	R (note 1)	1	1
store instruction (ST, STB, STH, UNLOCK)	R (note 1)	1	1	W (note 1)	1	(1) (note 2)
BSET, BCLR instructions	R (note 1)	1	R (note 1) +3	W (note 1)	1	–
multiply instruction (MUL)	R (note 1)	1	3	–	–	1
divide/reminder instruction (DIV, DIVU, REM, REMU)	R (note 1)	1	37	–	–	1
other instructions (DSP function instructions, including BTST, SETPSW, CLRPSW)	R (note 1)	1	1	–	–	1

Note 1: R, W: Refer to the user's manual prepared for each product.
Note 2: Within the store instruction, only instructions which include the register indirect and register update addressing mode require 1 cycle in the WB stage. All other instructions do not require extra cycles.

Appendix Table 4.1.2 Instruction Execution Cycles per Pipeline Stage [FPU instructions]

	the number of execution cycles in each stage						
instruction	IF	D	E1	EM	EA	E2	WB
FMADD, FMSUB instructions	R (note 1)	1	–	1	1	1	1
FDIV instruction	R (note 1)	1	14	–	–	1	1
other FPU instructions	R (note 1)	1	1	–	–	1	1

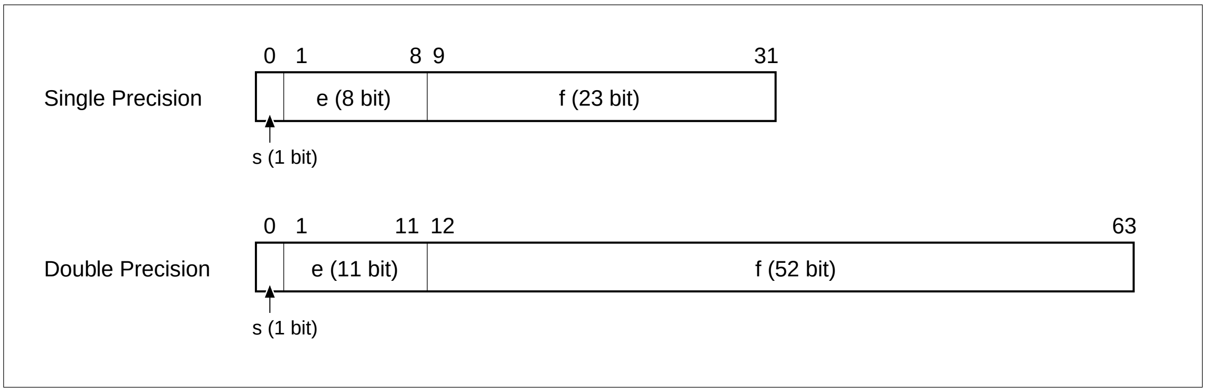
Note 1: R, W: Refer to the user's manual prepared for each product.

Appendix 5 IEEE754 Specification Overview

The following is a basic overview of the IEEE754 specification. M32R-FPU fulfills the IEEE754 requirements through a combination of software and hardware features.

Appendix 5.1 Floating Point Formats

The following describes the floating-point formats.



Appendix Figure 5.1.1 Floating-Point Formats

s: Sign bit. 0 = positive number, 1 = negative numbers
e: Exponent. This represents a value that was made positive by adding 127 to a single precision value or 1023 to a double precision value (biased exponent).
f : Fraction. Represents the fraction field of the value.

Using these symbols, the floating-point values (normalized numbers) can be described by the following expressions:

Single-Precision Format: $(-1)^{s} \times 1.f \times 2^{(e-127)}$
Double-Precision Format: $(-1)^{s} \times 1.f \times 2^{(e-1023)}$

- Certain values do not fit into the above expressions, such as ± , ±0, NaN (Not a Number), denormalized numbers, etc.
- Other formats, such as expanded double precision, can also be used.

★ M32R-FPU only supports the single-precision format. The double precision format is supported in the software library.

Appendix Table 5.1.1 Single Precision Floating-Point Bit Values

Exponent		Expressed value
Before adding bias	After adding bias (=0111 1111)	
0111 1111 (+127)	1111 1110	Normalized number
• • •	• • •	(The absolute value can be described for the range
1000 0010 (-126)	0000 0001	of 1. 0…0 x 2 ^ -126 to 1. 1…1 x 2 ^ 127)
(1000 0001 (-127))	0000 0000	Fraction field = all 0: ±0 Fraction field all 0: denormalized number
(1000 0000 (-128))	1111 1111	Fraction field = all 0: ± Fraction field all 0: NaN (the value is split into SNaN and QNaN according to the value of high-order bit of the fraction field)

(1) Denormalized Numbers

Denormalized numbers represent numbers (values??) that have an absolute value less than 1. 0...0 x 2 ^ -126. Single-precision denormalized numbers are expressed as follows:

(-1) ^ s x 0.f x 2 ^ -126

(2) NaN (Not a Number)

SNaN (Signaling NaN): a NaN in which the MSB of the decimal fraction field is "0". When SNaN is used as the source operand in an operation, an IVLD occurs. SNaNs are useful in identifying program bugs when used as the initial value in a variable. However, SNaNs cannot be generated by hardware.

QNaN (Quiet NaN): a NaN in which the MSB of the decimal fraction field is "1". Even when QNaN is used as the source operand in an operation, an IVLD will not occur (excluding comparison and format conversion). Because a result can be checked by the arithmetic operations, QNaN allows the user to debug without executing an EIT processing. QNaNs are created by hardware.

Appendix 5.2 Rounding

The following 4 rounding modes are specified by IEEE754.

Appendix Table 5.2.1 Four Rounding Modes

Rounding Mode	Operation
Round to Nearest (default)	Assuming an infinite range of precision, round to the best approximation of the result. Round an interval arithmetic result to an even number.
Round toward –Infinity	Round to the smaller magnitude of the result.
Round toward +Infinity	Round to the larger magnitude of the result.
Round toward 0	Round to the smaller in magnitude of the absolute value of the result.

- “Round to Nearest” is the default mode and produces the most accurate value.
- “Round toward –Infinity,” “Round toward +Infinity” and “Round toward Zero” are used for interval arithmetic to insure precision

Appendix 5.3 Exceptions

IEEE754 allows the following 5 exceptions. The floating-point status register is used to determine whether the EIT process will be executed when an Exception occurs.

(1) Overflow Exception (OVF)

The exception occurs when the absolute value of the operation result exceeds the largest describable precision in the floating-point format. Appendix Table 5.3.1 shows the operation results when an OVF occurs.

Appendix Table 5.3.1 Operation Result due to OVF Exception

Rounding Mode	Sign of Result	Result	
		when the OVF EIT processing is masked	when the OVF EIT processing is executed
–Infinity	+	+MAX	round (x2 ^ -a) a = 192 (single-precision) a = 1536 (double-precision)
	–	–Infinity	
+Infinity	+	+Infinity	
	–	–MAX	
0	+	+MAX	
	–	–MAX	
Nearest	+	+Infinity	
	–	–Infinity	

Note : • When the Underflow Exception Enable (EU) bit (FPSR register bit 18) = "0"
• When the Underflow Exception Enable (EU) bit (FPSR register bit 18) = "1"

(2) Underflow Exception (UDF)

The exception occurs when the absolute value of the operation result is less then the largest describable precision in the floating-point format. Appendix Table 5.3.2 shows the operation results when a UDF occurs.

Appendix Table 5.3.2 Operation Results due to UDF Exception

Result	
when the UDF EIT processing is masked	when the UDF EIT processing is executed
Denormalized Numbers (The denomalize flag is set only when rounding occurs.)	round (x2 ^ a) a = 192 (single-precision), a = 1536 (double-precision)

Note: • When the operation result is rounded, an Inexact Exception is generated simultaneously.

(3) Inexact Exception (IXCT)

The exception occurs when the operation result differs from a result led out with an infinite range of precision. Appendix Table 5.3.3 shows operation results and the respective conditions in which each IXCT occurs.

Appendix Table 5.3.3 Operation Results and Respective Conditions for IXCT Exception

Occurrence Condition	Result	
	when the IXCT EIT processing is masked	when the IXCT EIT processing is executed
Overflow occurs in OVF Exception masked condition	Reference OVF Exception table	Same as left
Rounding occurs	Rounded value	Same as left

(4) Zero Division Exception (DIV0)

The exception occurs when a finite, nonzero value is divided by zero. Appendix Table 5.3.4 shows the operation result when a DIV0 occurs.

Appendix Table 5.3.4 Operation Results for DIV0 Exception

Dividend	Result	
	when the DIV0 EIT processing is masked	when the IXCT EIT processing is executed
Nonzero finite value	± Infinity (Sign of result is exclusive-OR (EXOR) of signs of divider and dividend.)	Destination unchanged

Please note that the DIV0 EIT operation does not occur in the following factors.

Dividend	Operation
0	Invalid Operation Exception occurs
Infinity	No Exception occurs (result is “Infinity”)

(5) Invalid Operation Exception (IVLD)

The exception occurs when an invalid operation is executed. Appendix Table 5.3.5 shows operation results and the respective conditions in which each IVLD occurs.

Appendix Table 5.3.5 Operation Results due to IVLD Exception

Occurrence Condition	Result	
	when the IVLD EIT processing is masked	when the IVLD EIT processing is executed
Operation for SNaN operand	QNaN	(Destination unchanged)
+Infinity– (+Infinity), –Infinity– (–Infinity)		
0 ✕ Infinity		
0 ÷ 0, Infinity ÷ Infinity		
oute operation for values less then 0		
Integer conversion overflow: NaN and are converted to integers	Undefined	
When < or > comparison was performed on NaN	(No change)	

Important: The following operations never generate an Exception.

(-0): returns –0

/ 0:   returns   (Sign of result is exclusive-OR (EXOR) of signs of divider and dividend.)

■ Definition of Terms

• Exception

Special conditions generated by execution of floating-point instructions. The corresponding enable bits of the floating-point status register are used to determine whether the EIT processing will be executed when an Exception occurs. However, the actual generation of an exception cannot be masked.

• EIT Processing

An operation triggered by the generation of an Exception, in which the flow jumps to a floating-point Exception vector address, or a string of related Exception operation sequences is triggered. The corresponding enable bits of the floating-point status register are used to determine whether the EIT processing will be executed when an Exception occurs.

• Intermediate Result of Operation

The value resulting from calculations of infinite and unbounded exponent and mantissa bits. In actual implementation, the number of exponent and mantissa bits is finite and the intermediate result is rounded so that the final operation result can be determined.

Appendix 6 M32R-FPU Specification Supplemental Explanation

Appendix 6.1 Operation Comparision: Using 1 instruction (FMADD or FMSBU) vs. two instructions (FMUL and FADD)

The following is an explanation of the differences between an operation using just one instruction (FMADD or FMSUB) and an operation using 2 instructions (FMUL and FADD).

Appendix 6.1.1 Rounding Mode

The rounding mode for an operation using both FMUL and FADD rounds both FMUL and FADD according to the setting of the FPSR RM field. However, the result of the FMADD or FMSUB instruction in Step 1 (multiply stage) is not rounded according to the setting of FPSR RM field, rather it is rounded toward zero.

Appendix 6.1.2 Exception occurring in Step 1

Two instructions are compared below as examples of Exception occurring in Step 1.

● FMUL + FADD:

```
FMUL    R3, R1, R2    (R3 = R1 * R2)
FADD    R0, R3, R0    (R0 = R3 + R0)
```

● FMADD or FMSUB:

```
FMADD   R0, R1, R2    (R0 = R0 +R1 * R2)
```

Note: If the register supports different operations than those described above, the operations may differ in some ways to those shown below.

(1) Overflow occurs in Step 1

<When EO = 0, EX = 0: OVF and IXCT occur>

Type of R0	Condition		FMUL + FADD Operation	FMADD Operation
Normalized number, 0	–		R0 = OVF immediate value (Note 1) + R0	R0 = OVF immediate value (Note 2)
Infinity	when OVF immediate value is R0 and the opposite sign of the infinity sign	EV=0	IVLD occurs R0=H'7FFF FFFF	same as left
		EV=1	IVLD occurs, EIT occurs R0 = maintained	same as left
	factors other than above	–	R0 = (same as original value)	same as left
Denormalized number	DN=0		UIPL occurs, EIT occurs R0 = maintained	same as left
	DN=1		R0 = OVF immediate value (Note 1)	same as left
QNaN	–		R0 = maintained (QNaN)	same as left
SNaN	EV=0		IVLD occurs R0 = R0 converted to QNaN	same as left
	EV=0		IVLD occurs, EIT occurs R0 = maintained (SNaN)	same as left

Note 1: Refer to [Appendix Table 5.3.1 Operation Result due to OVF Exception] for immediate values if an overflow occurs due to Overflow Exclusion when the EIT processing is masked.

Note 2: In Step 1, the rounding mode is set to [Round toward 0]. However, when an overflow occurs, the immediate value is rounded according to the rounding mode. Refer to [Appendix Table 5.3.1 Operation Result due to OVF Exception] for these values. However, when the rounding mode is [round toward nearest], the OVF immediate value = infinity and the R0 value becomes the same as that of FMUL + FADD.

<When EO = 1: OVF occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	EIT occurs when FMUL is completed R0 = maintained	EIT occurs, R0 = maintained
Denormalized number	DN=0	Same as above	UIPL occurs, EIT occurs R0 = maintained
	DN=1	Same as above	EIT occurs R0 = maintained
QNaN	–	Same as above	Same as above
SNaN	EV=0	Same as above	IVLD occurs, EIT occurs R0 = maintained
	EV=1	Same as above	Same as above

(2) When underflow occurs in Step 1

<When EU = 0, DN = 1: UDF occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	R0 = R0 + 0	Same as left
Denormalized number	–	R0 = 0	Same as left
QNaN	–	R0 = maintained (QNaN)	Same as left
SNaN	EV=0	R0 = R0 converted to QNaN IVLD occurs	Same as left
	EV=1	R0 = maintained (SNaN) IVLD occurs, EIT occurs	Same as left

<When EU = 0, DN = 0: UDF and UIPL occur>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	EIT occurs when FMUL is completed R0 = maintained	EIT occurs, R0 = maintained
Denormalized number	–	Same as above	Same as above
QNaN	–	Same as above	Same as above
SNaN	EV=0	Same as above	IVLD occurs, EIT occurs R0 = maintained
	EV=1	Same as above	Same as above

<When EU = 1: UDF occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	EIT occurs when FMUL is completed R0 = maintained	EIT occurs, R0 = maintained
Denormalized number	DN=0	Same as above	UIPL occurs, EIT occurs R0 = maintained
	DN=1	Same as above	EIT occurs R0 = maintained
QNaN	–	Same as above	Same as above
SNaN	EV=0	Same as above	IVLD occurs, EIT occurs R0 = maintained
	EV=1	Same as above	Same as above

(3) When Invalid Operation Exception occurs in Step 1

■ If at least one of [R1, R2] is an SNaN

<When EV = 0: IVLD occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized	–	R0 = R3 (SNaN converted to QNaN)	Same as left
Denormalized number	DN=0	R0 = R3 (SNaN converted to QNaN)	Same as left
	DN=1	R0 = R3 (SNaN converted to QNaN)	Same as left
QNaN	–	R0 = maintained (QNaN)	Same as left
SNaN	–	R0 = R0 converted to QNaN	Same as left

<When EV = 1: IVLD occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	EIT occurs when FMUL is completed R0 = maintained	EIT occurs, R0 = maintained
Denormalized number	DN=0	Same as above	UIPL occurs, EIT occurs R0 = maintained
	DN=1	Same as above	EIT occurs, R0 = maintained
QNaN	–	Same as above	Same as above
SNaN	–	Same as above	Same as above

■ If "✕ " occurs in [R1, R2]

<When EV = 0: IVLD occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized	–	R0 = H'7FFF FFFF	Same as left
Denormalized number	DN=0	R0 = H'7FFF FFFF	Same as left
	DN=1	R0 = H'7FFF FFFF	Same as left
QNaN	–	R0 = maintained (QNaN)	Same as left
SNaN	–	R0 = R0 converted to QNaN	Same as left

<When EV = 1: IVLD occurs>

Same results as when "If at least one of [R1, R2] is an SNaN."

(4) When Inexact Operation Exception occurs in Step 1

■ If an Inexact Operation occurs due to rounding:
<When EX = 0: IXCT occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	R0 = rounded value of R1*R2 + R0	Same as left
Denormalized number	DN=0	UIPL occurs, EIT occurs R0 = maintained	Same as left
	DN=1	R0 = rounded value of R1*R2	Same as left
QNaN	–	R0 = maintained (QNaN)	Same as left
SNaN	EV=0	IVLD occurs R0 = R0 converted to QNaN	Same as left
	EV=1	IVLD occurs, EIT occurs R0 = maintained (SNaN)	Same as left

<When EX = 1: IXCT occurs>

Type of R0	Condition	FMUL + FADD Operation	FMADD Operation
Normalized number, 0, Infinity	–	EIT occurs when FMUL is completed R0 = maintained	EIT occurs, R0 = maintained
Denormalized number	DN=0	Same as above	UIPL occurs, EIT occurs R0 = maintained
	DN=0	Same as above	EIT occurs R0 = maintained
QNaN	–	Same as above	Same as above
SNaN	EV=0	Same as above	IVLD occurs, EIT occurs R0 = maintained
	EV=1	Same as above	Same as above

■ When an Inexact Operation occurs due to an OVF at EO = 0:

<When EV = 0: IXCT occurs>
Refer to “(1) Overflow occurs in Step 1 <When EO = 0, EX = 0: OVF and IXCT occur>”.

<When EV = 1: IXCT occurs>
Same results as “■ If an Inexact Operation occurs due to rounding <when EX = 1: IXCT occurs>”.

Appendix 6.2 Rules concerning Generation of QNaN in M32R-FPU

The following are rules concerning generating a QNaN as an operation result. Instructions that generate NaNs as operation results are FADD, FSUB, FMUL, FDIV, FMADD, and FMSUB.

[Important Note]

This rule does not apply when the data that is sent to Rdest, the results of the FCMP or FCMPE comparison, comprise a NaN bit pattern.

<FADD, FSUB, FMUL, FDIV>

Source Operand (Rsrc1, Rsrc2)	Rdest
SNaN and QNaN	SNaN converted to QNaN (Note 1)
Both SNaN	Rsrc2 converted to QNaN (Note 1)
Both QNaN	Rscr2
SNaN and actual number	SNaN converted to QNaN (Note 1)
QNaN and actual number	QNaN
Neither operand is NaN; IVLD occurs	H'7FFF FFFF

Note 1: SNaN b9 is set to “1” and the operand is converted to QNaN.

<FMADD, FMSUB>

Source Operand		Rdest
Rdest	Rsrc1, Rsrc2	
Actual number	SNaN and QNaN	SNaN converted to QNaN (Note 1)
	Both SNaN	Rsrc2 converted to QNaN (Note 1)
	Both QNaN	Rscr2
	SNaN and actual number	SNaN converted to QNaN (Note 1)
	QNaN and actual number	QNaN
	Neither operand is NaN; IVLD occurs	H'7FFF FFFF
QNaN	Don't care	Rdest (maintained)
SNaN	Don't care	Rdest converted to QNaN (Note 1)

Note 1: SNaN b9 is set to “1” and the operand is converted to QNaN.

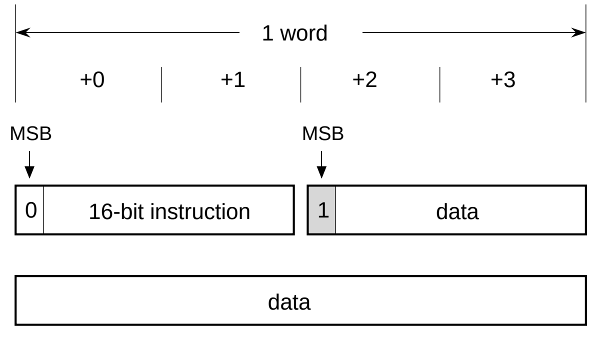
Appendix 7 Precautions

Appendix 7.1 Precautions to be taken when aligning data

When aligning or allocating the data area following the code area in a program, the alignment must be done from an address that has an adjusted word alignment.

If the data area is aligned or allocated without adjusting the word alignment, a 16-bit instruction may exist in the high-order halfword of the word, and data with MSB of "1" may be aligned to the following halfword. In this case, the M32R family upward-compatible CPU recognizes the 16-bit instruction and the data as a pair of parallel executable instructions and executes the instructions as such.

In consideration of the upward compatibility of software when programming, if the high-order halfword has a 16-bit instruction, make sure that the following data area is aligned or allocated from an address that has an adjusted word alignment.



This page left blank intentionally.

INDEX

INDEX

Symbol

A

B

C

D

INDEX

INDEX

R

S

T

U

MITSUBISHI SEMICONDUCTORS
M32R-FPU SOFTWARE MANUAL

Jan. First Edition 2003

Editioned by
Committee of editing of Mitsubishi Semiconductor User's Manual

Published by
Mitsubishi Electric Corp., Semiconductor Marketing Division

Software Manual
M32R-FPU

Renesas Technology Corp.
Nippon Bldg.,6-2,Otemachi 2-chome,Chiyoda-ku,Tokyo,100-0004 Japan

New publication, effective Jan. 2003.
Specifications subject to change without notice.