

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

MITSUBISHI 32-BIT SINGLE-CHIP MICROCOMPUTER

M32R family

Software Manual

1998-07-01

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 materials 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 or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams and charts, represent 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.
- 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.

Table of contents

CHAPTER 1 CPU PROGRAMMING MODEL

1.1 CPU register	1-2
1.2 General-purpose registers	1-2
1.3 Control registers	1-3
1.3.1 Processor status word register: PSW (CR0)	1-4
1.3.2 Condition bit register: CBR (CR1)	1-5
1.3.3 Interrupt stack pointer: SPI (CR2) User stack pointer: SPU (CR3)	1-5
1.3.4 Backup PC: BPC (CR6)	1-5
1.4 Accumulator	1-6
1.5 Program counter	1-6
1.6 Data format	1-7
1.6.1 Data types	1-7
1.6.2 Data formats	1-8
1.7 Addressing mode	1-10

CHAPTER 2 INSTRUCTION SET

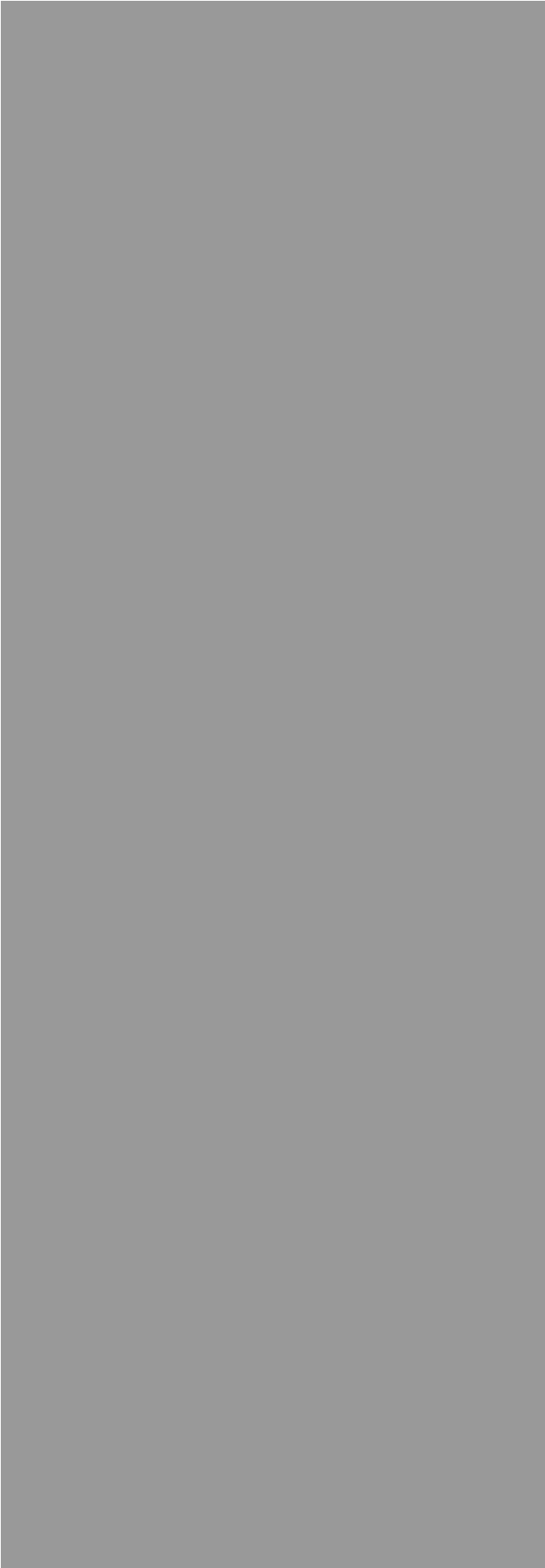
2.1 Instruction set overview	2-2
2.1.1 Load/store instructions	2-2
2.1.2 Transfer instructions	2-4
2.1.3 Operation instructions	2-4
2.1.4 Branch instructions	2-6
2.1.5 EIT-related instructions	2-8
2.1.6 DSP function instructions	2-8
2.2 Instruction format	2-11

CHAPTER 3 INSTRUCTIONS

3.1 Conventions for instruction description	3-2
3.2 Instruction description	3-5

APPENDICES

Appendix A Instruction list	A-2
Appendix B Pipeline stages	A-5
B.1 Overview of pipeline processing	A-5
B.2 Instructions and pipeline processing	A-6
B.3 Pipeline processing	A-7
Appendix C Instruction execution time	A-10



CHAPTER 1

CPU PROGRAMMING MODEL

- 1.1 CPU register
- 1.2 General-purpose registers
- 1.3 Control registers
- 1.4 Accumulator
- 1.5 Program counter
- 1.6 Data format
- 1.7 Addressing mode

CPU PROGRAMMING MODEL

1.1 CPU register

1.1 CPU register

The M32R CPU has 16 general-purpose registers, 5 control registers, an accumulator and a program counter. The accumulator is of 64-bit width. The registers and program counter are of 32-bit width.

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. R14 is used as the link register and R15 as the stack pointer (SPI or SPU). 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 bit (SM) in the processor status word register (PSW).

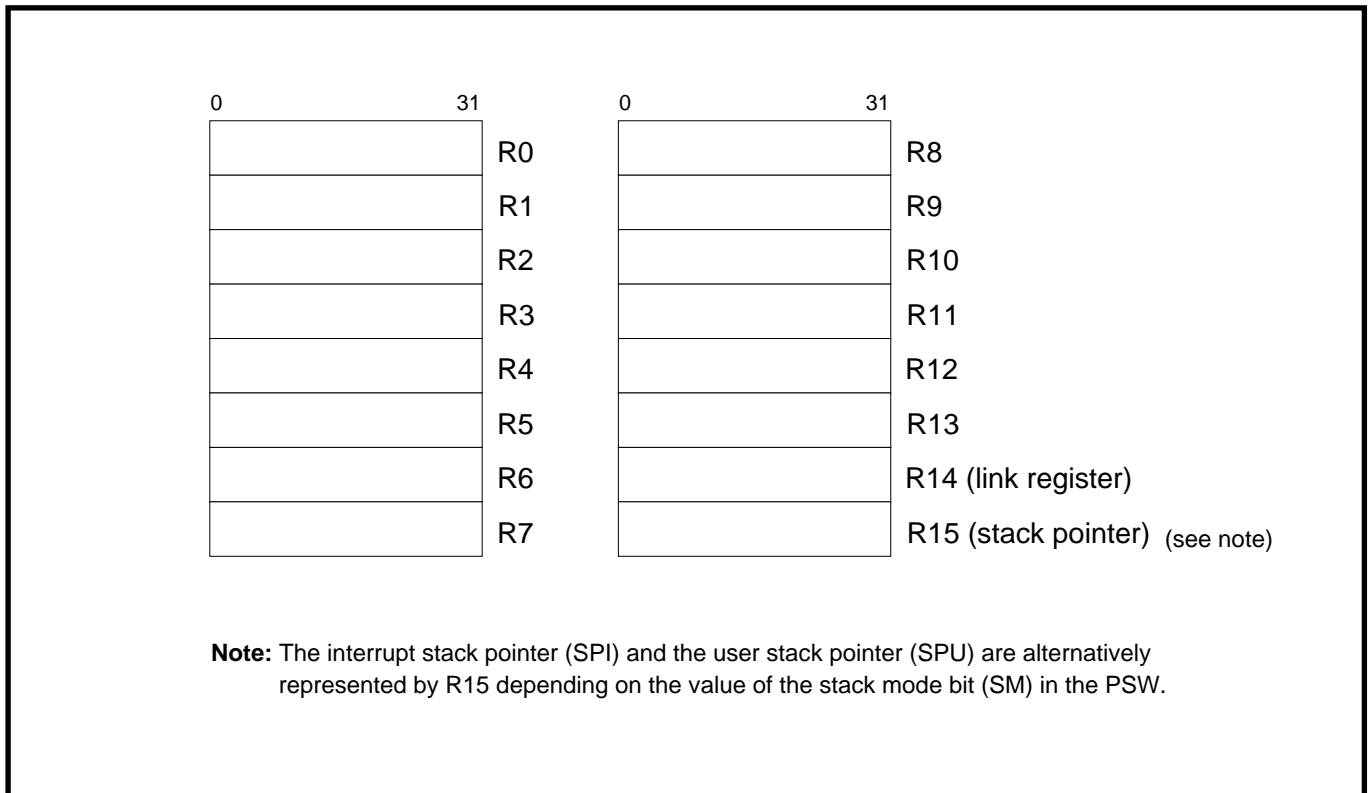


Fig. 1.2.1 General-purpose registers

1.3 Control registers

There are 5 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) and the backup PC (BPC). The **MVTC** and **MVFC** instructions are used for writing and reading these control registers.

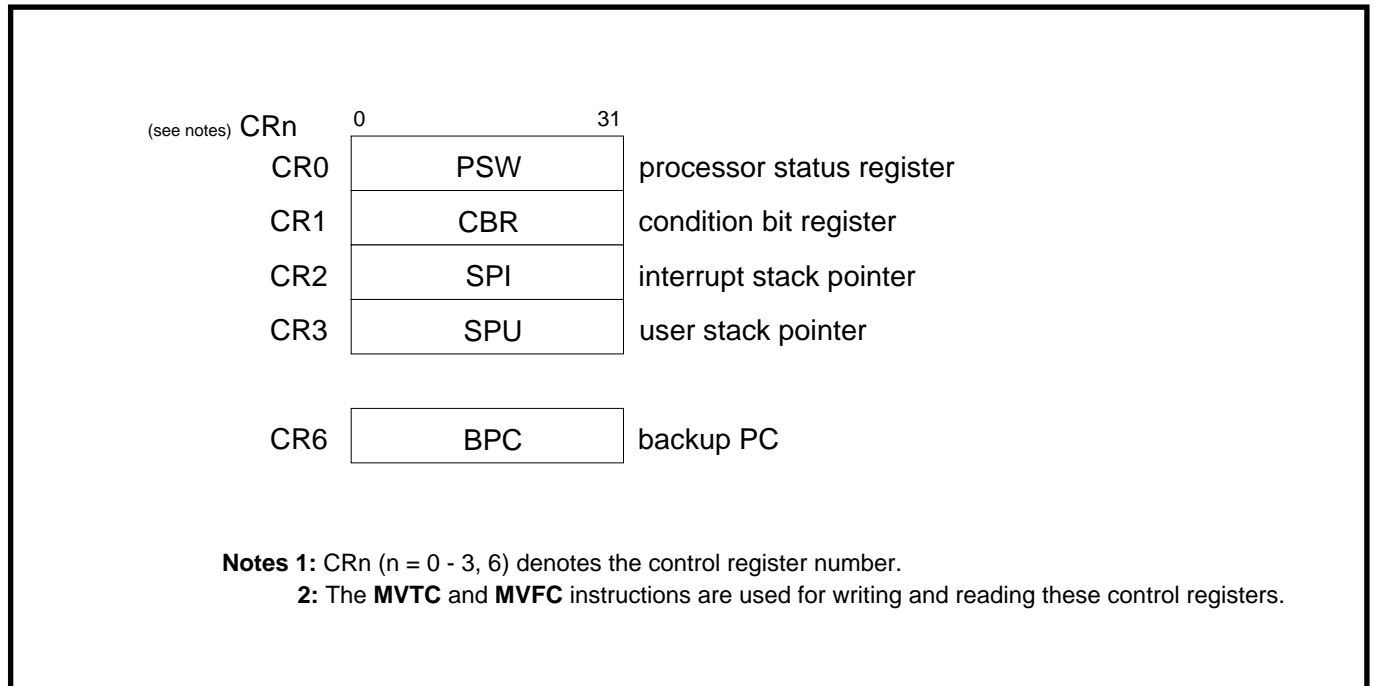


Fig. 1.3.1 Control registers

CPU PROGRAMMING MODEL

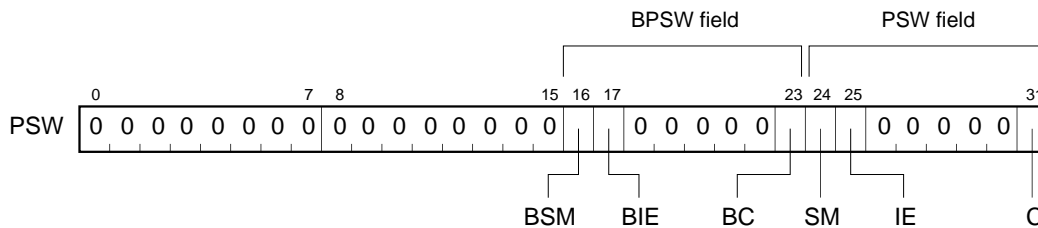
1.3 Control registers

1.3.1 Processor status word register: PSW (CR0)

The processor status word register (PSW) shows the M32R CPU status. It consists of the current PSW field, and the BPSW field where a copy of the PSW field is saved when EIT occurs.

The PSW field is made up of the stack mode bit (SM), the interrupt enable bit (IE) and the condition bit (C).

The BPSW field is made up of the backup stack mode bit (BSM), the backup interrupt enable bit (BIE) and the backup condition bit (BC) .



D	bit name	function	init.	R	W
16	BSM (backup SM)	saves value of SM bit when EIT occurs	undefined	<input type="radio"/>	<input type="radio"/>
17	BIE (backup IE)	saves value of IE bit when EIT occurs	undefined	<input type="radio"/>	<input type="radio"/>
23	BC (backup C)	saves value of C bit when EIT occurs	undefined	<input type="radio"/>	<input type="radio"/>
24	SM (stack mode)	0: uses R15 as the interrupt stack pointer 1: uses R15 as the user stack pointer	0	<input type="radio"/>	<input type="radio"/>
25	IE (interrupt enable)	0: does not accept interrupt 1: accepts interrupt	0	<input type="radio"/>	<input type="radio"/>
31	C (condition bit)	indicates carry, borrow and overflow resulting from operations (instruction dependent)	0	<input type="radio"/>	<input type="radio"/>

Note: "init." ...initial state immediately after reset

"R"... : read enabled

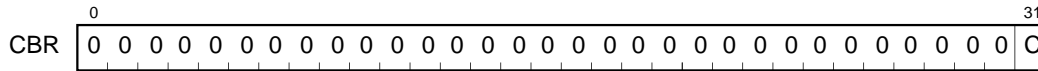
"W"... : write enabled

CPU PROGRAMMING MODEL

1.3 Control registers

1.3.2 Condition bit register: CBR (CR1)

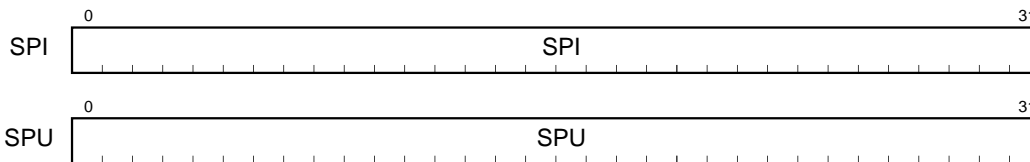
The condition bit register (CBR) is a separate register which contains the condition bit (C) in the PSW. The value of the condition bit (C) in the PSW is reflected in this register. This register is read-only. An attempt to write to the CBR with the **MVTC** instruction is ignored.



1.3.3 Interrupt stack pointer: SPI (CR2)

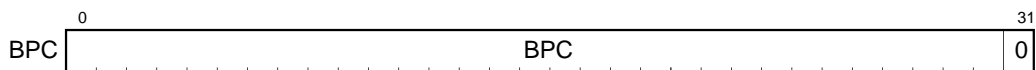
User stack pointer: SPU (CR3)

The interrupt stack pointer (SPI) and the user stack pointer (SPU) retain the current stack address. The SPI and SPU 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 bit (SM) in the PSW.



1.3.4 Backup PC: BPC (CR6)

The backup PC (BPC) is the register where a copy of the PC value is saved when EIT occurs. Bit 31 is fixed at "0". When EIT occurs, the PC value immediately before EIT occurrence or that of the next instruction is set. The value of the BPC is reloaded to the PC when the **RTE** instruction is executed. However, the values of the lower 2 bits of the PC become "00" on returning (It always returns to the word boundary).



CPU PROGRAMMING MODEL

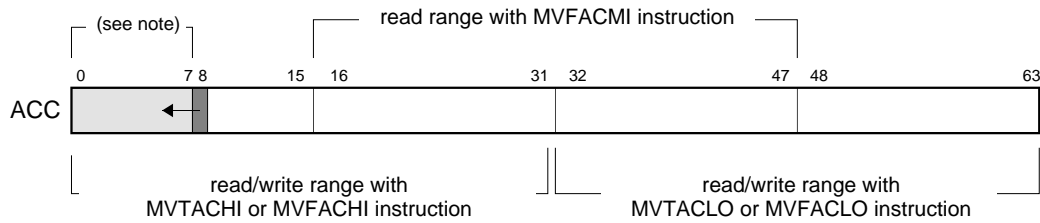
1.4 Accumulator

1.4 Accumulator

The accumulator (ACC) is a 64-bit register used for the DSP function.

Use the **MVTACHI** and **MVTACLO** instructions for writing to the accumulator. The high-order 32 bits (bit 0 - bit 31) can be set with the **MVTACHI** instruction and the low-order 32 bits (bit 32 - bit 63) can be set with the **MVTACLO** instruction. Use the **MVFACHI**, **MVFACLO** and **MVFACMI** instructions for reading from the accumulator. The high-order 32 bits (bit 0 - bit 31) are read with the **MVFACHI** instruction, the low order 32 bits (bit 32 - bit 63) with the **MVFACLO** instruction and the middle 32 bits (bit 16 - bit 47) with the **MVFACMI** instruction.

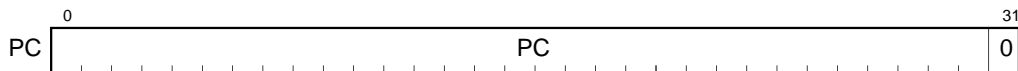
The **MUL** instruction also uses the accumulator and the contents are destroyed when this instruction is executed.



Note: Bits 0 - 7 are always read as the sign-extended value of bit 8.
An attempt to write to this area 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".



1.6 Data format

1.6.1 Data types

Signed and unsigned integers of byte (8 bits), halfword (16 bits), and word (32 bits) types are supported as data in the M32R CPU instruction set. A signed integer is represented in a 2's complement format.

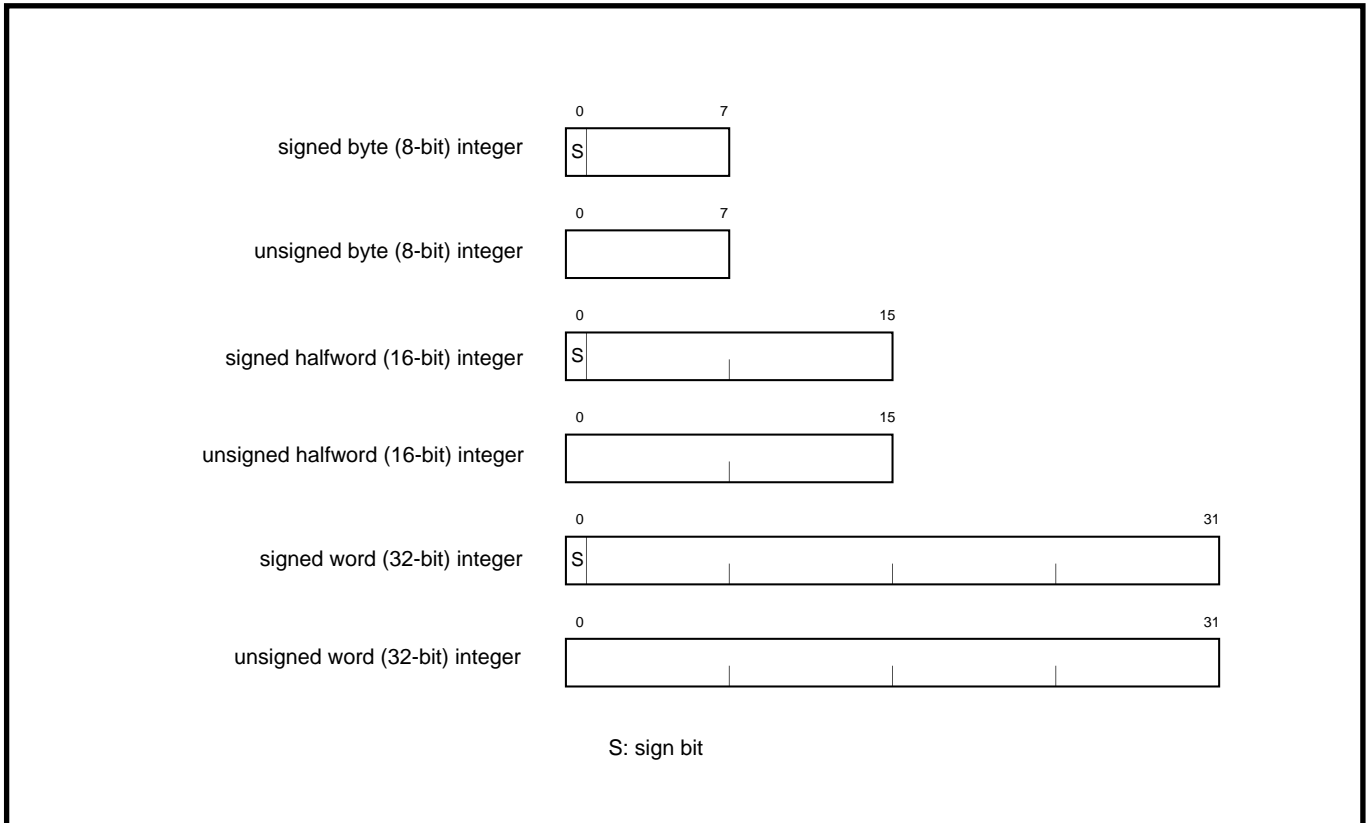


Fig. 1.6.1 Data types

CPU PROGRAMMING MODEL

1.6 Data format

1.6.2 Data formats

(1) Data format in a register

Data size of a register is always a word (32 bits).

Byte (8 bits) and halfword (16 bits) data in memory are sign-extended (the **LDB** and **LDH** instructions) or zero-extended (the **LDUB** and **LDUH** instructions) to 32 bits, and loaded into the register.

Word (32 bits) data in a register is stored to memory by the **ST** instruction. Halfword (16 bits) data in the LSB side of a register is stored to memory by the **STH** instruction. Byte (8 bits) data in the LSB side of a register is stored to memory by the **STB** instruction.

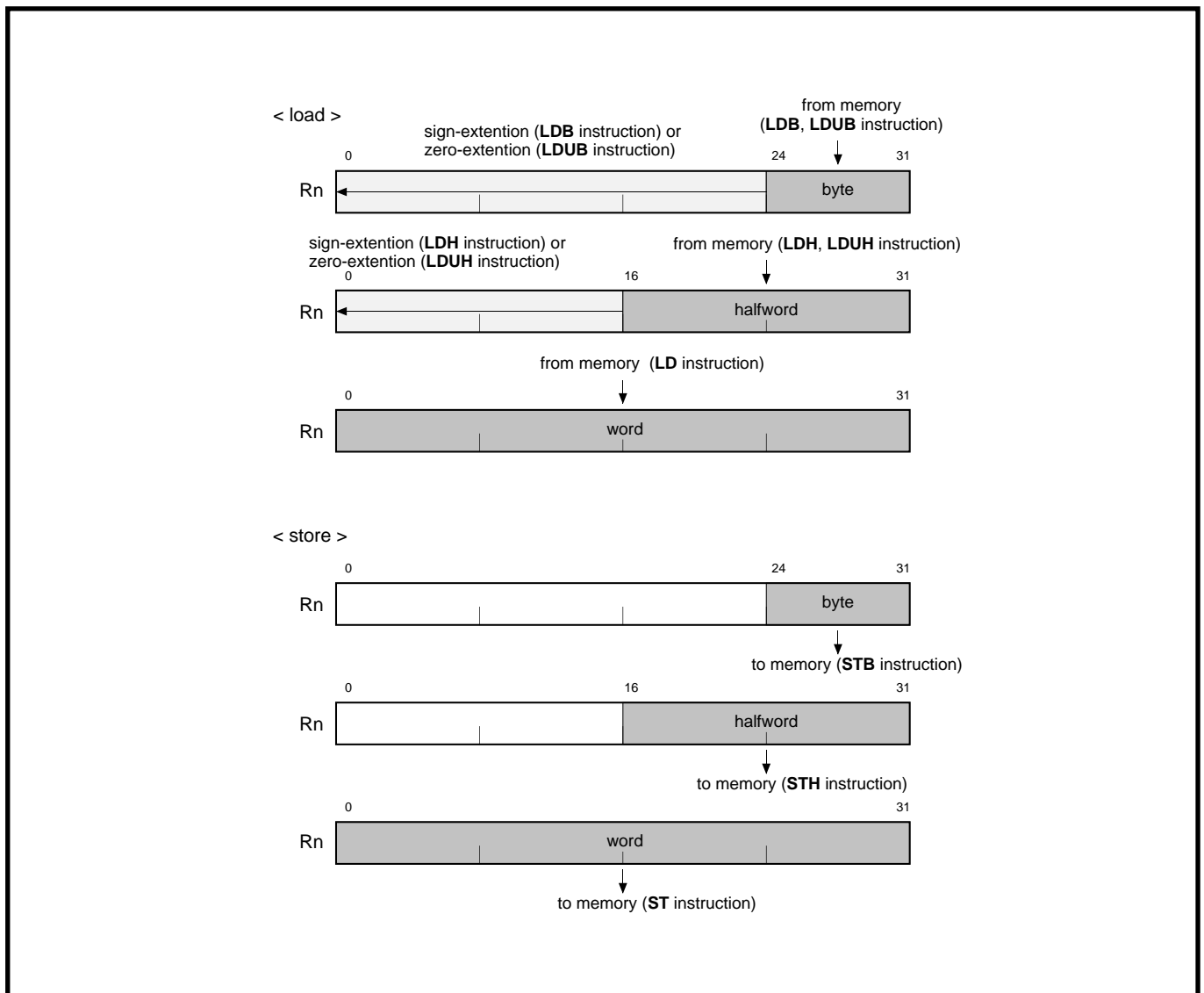


Fig. 1.6.2 Data format in a register

(2) Data format in memory

Data stored in memory can be one of these types: byte (8 bits), halfword (16 bits) or word (32 bits). Although the byte data can be located at any address, the halfword data and the word data can only be located on the halfword boundary and the word boundary, respectively. If an attempt is made to access data in memory which is not located on the correct boundary, an address exception occurs.

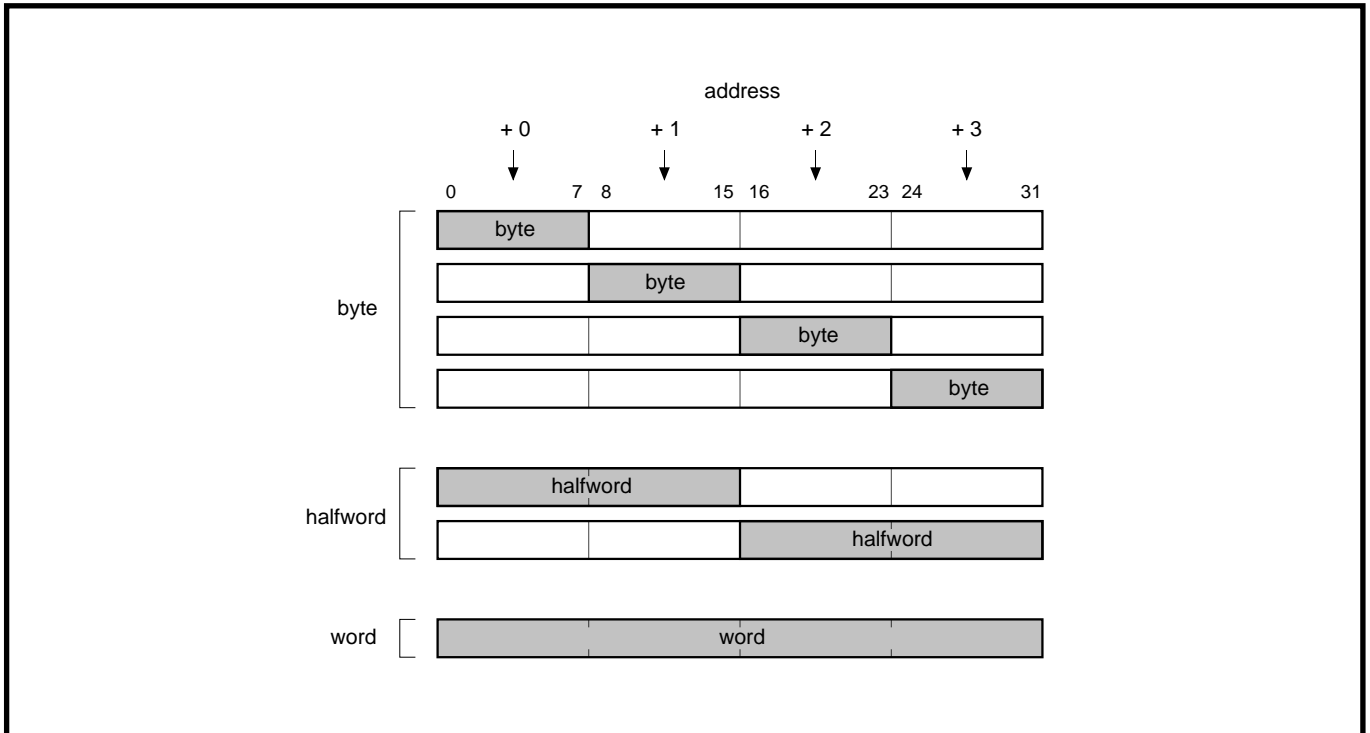


Fig. 1.6.3 Data format in memory

CPU PROGRAMMING MODEL

1.7 Addressing mode

1.7 Addressing mode

M32R 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

- 4 is added to the register contents [R+]
the contents of the register before update specify address of memory
(can be specified with LD instruction).
- 4 is added to the register contents [R+]
the contents of the register after update specify address of memory
(can be specified with ST instruction).
- 4 is subtracted from the register contents [R-]
the contents of the register after update specify address of memory
(can 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.



CHAPTER 2

INSTRUCTION SET

- 2.1 Instruction set overview
- 2.2 Instruction format

INSTRUCTION SET

2.1 Instruction set overview

2.1 Instruction set overview

The M32R CPU 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. A total of 83 instructions are implemented.

M32R supports compound instructions such as "load & address update" and "store & address update" which are useful for high-speed data transfer.

The M32R instruction set overview is explained below.

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

- 4 is added to the register value
the value in the register before update specifies the address
(can be specified only with the **LD** instruction).
- 4 is added to the register value
the value in the register after update specifies address
(can be specified only with the **ST** instruction).
- 4 is subtracted to the register value
the value in the register after update specifies address
(can be specified only with the **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.

INSTRUCTION SET

2.1 Instruction set overview

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

INSTRUCTION SET

2.1 Instruction set overview

2.1.4 Branch instructions

The branch instructions are used to change the program flow.

BC	Branch on C-bit
BEQ	Branch on equal
BEQZ	Branch on equal zero
BGEZ	Branch on greater than or equal zero
BGTZ	Branch on greater than zero
BL	Branch and link
BLEZ	Branch on less than or equal zero
BLTZ	Branch on less than zero
BNC	Branch on not C-bit
BNE	Branch on not equal
BNEZ	Branch on not equal 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.

INSTRUCTION SET

2.1 Instruction set overview

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.

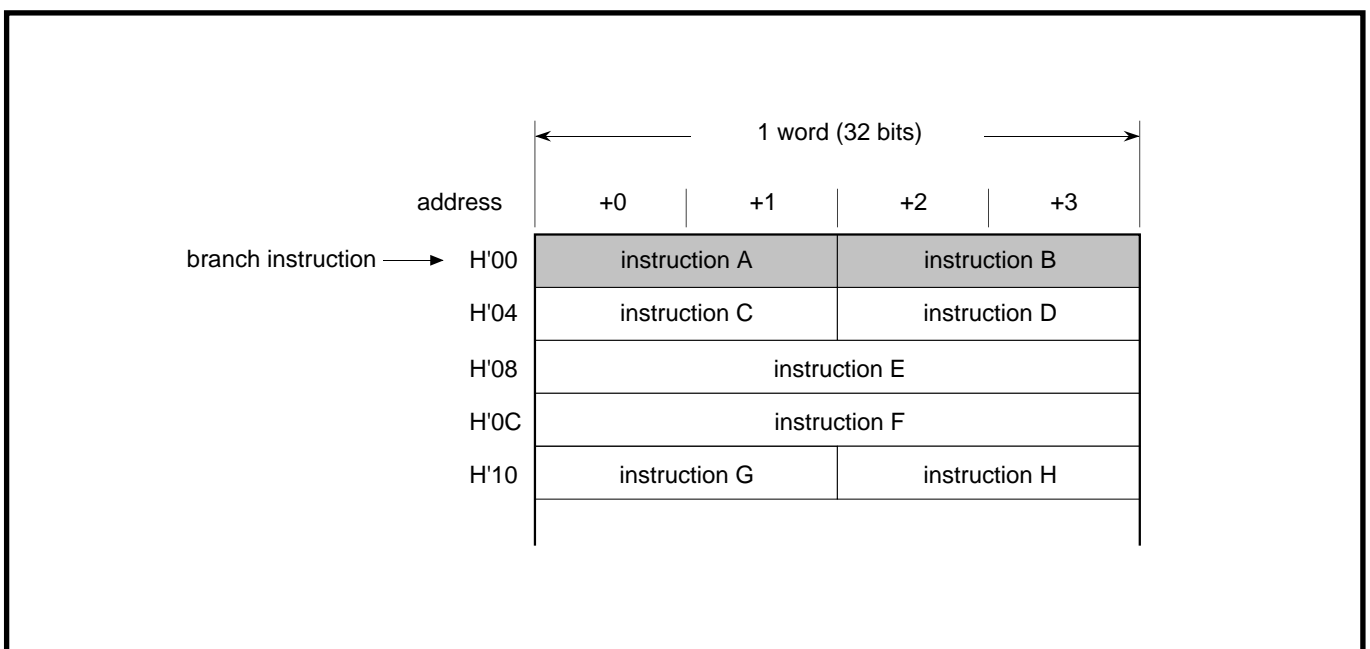


Fig. 2.1.1 Branch addresses of branch instruction

INSTRUCTION SET

2.1 Instruction set overview

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

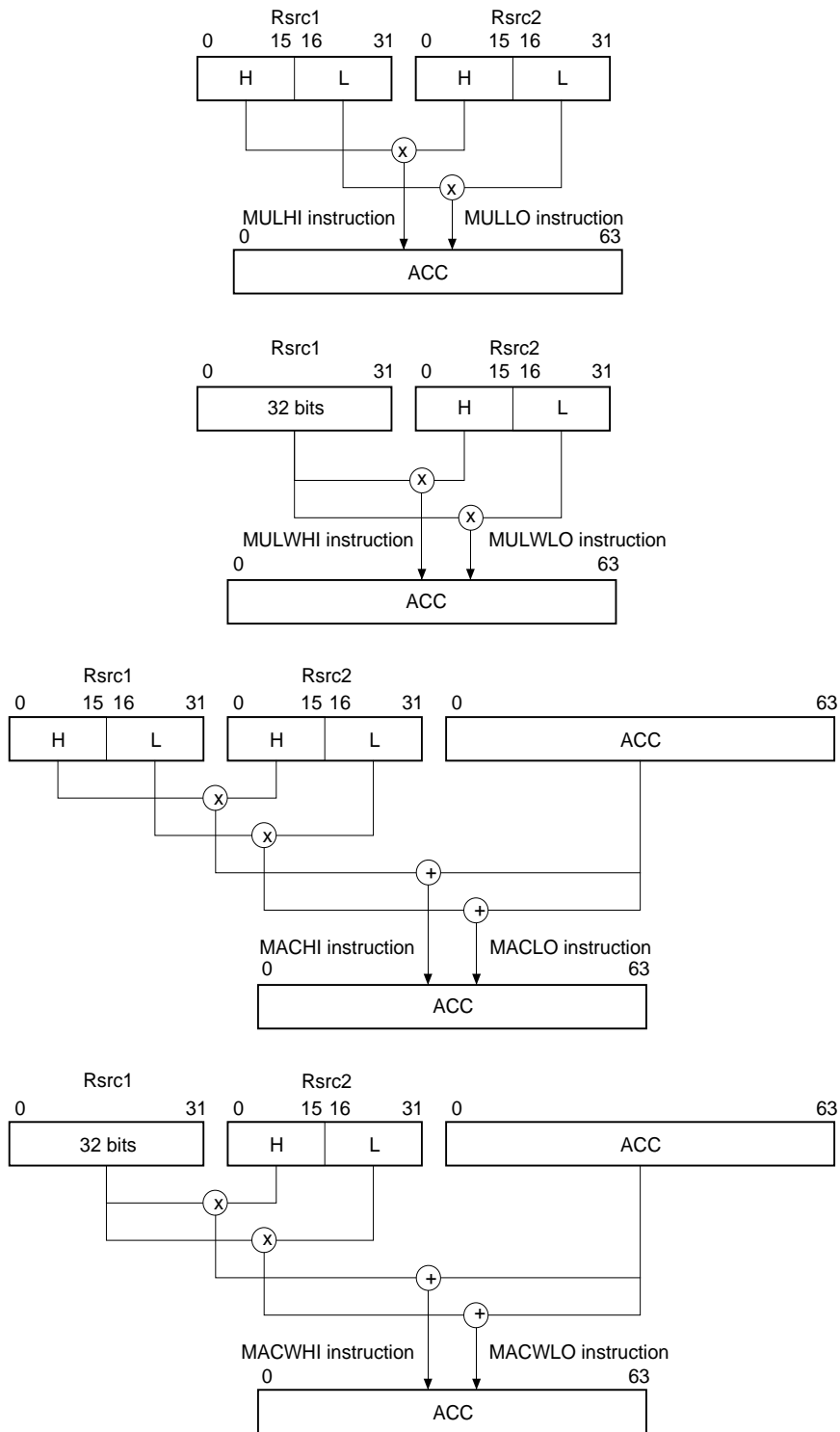
2.1.6 DSP function instructions

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 from accumulator high-order word
MVFACLO	Move from accumulator low-order word
MVFACMI	Move from accumulator middle-order word
MVTACHI	Move to accumulator high-order word
MVTACLO	Move to accumulator low-order word
RAC	Round accumulator
RACH	Round accumulator halfword

INSTRUCTION SET

2.1 Instruction set overview



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)

INSTRUCTION SET

2.1 Instruction set overview

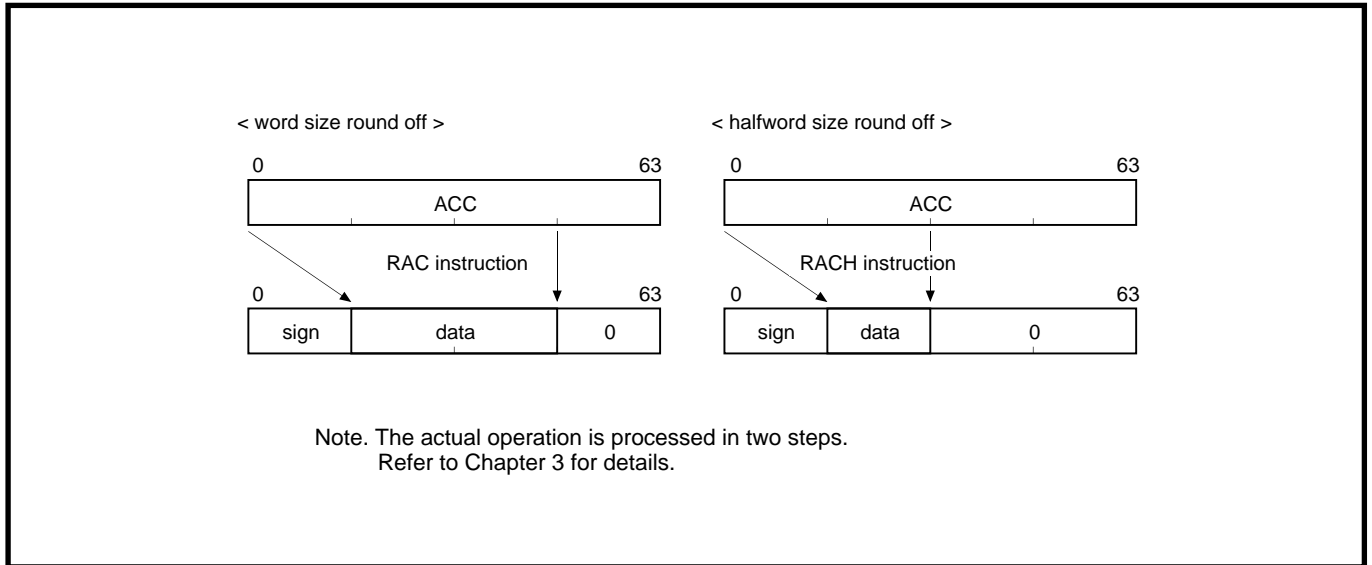


Fig. 2.1.3 DSP function instruction operation 2 (round off)

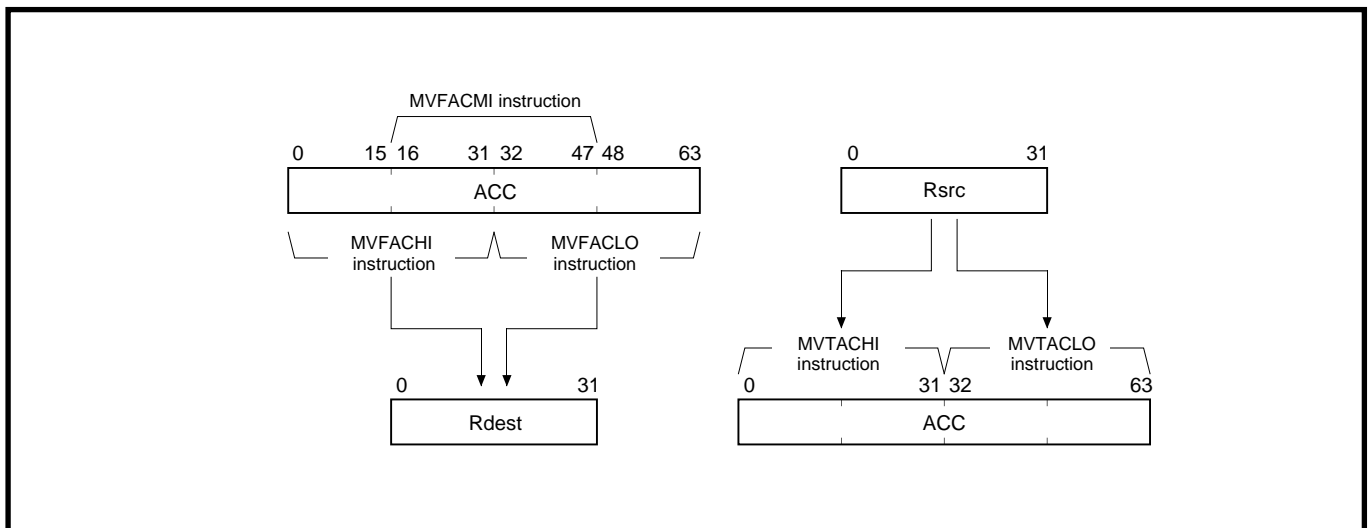


Fig. 2.1.4 DSP function instruction operation 3 (transfer between accumulator and register)

2.2 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 Fig. 2.2.1). Figure 2.2.2 shows the instruction format of M32R family.

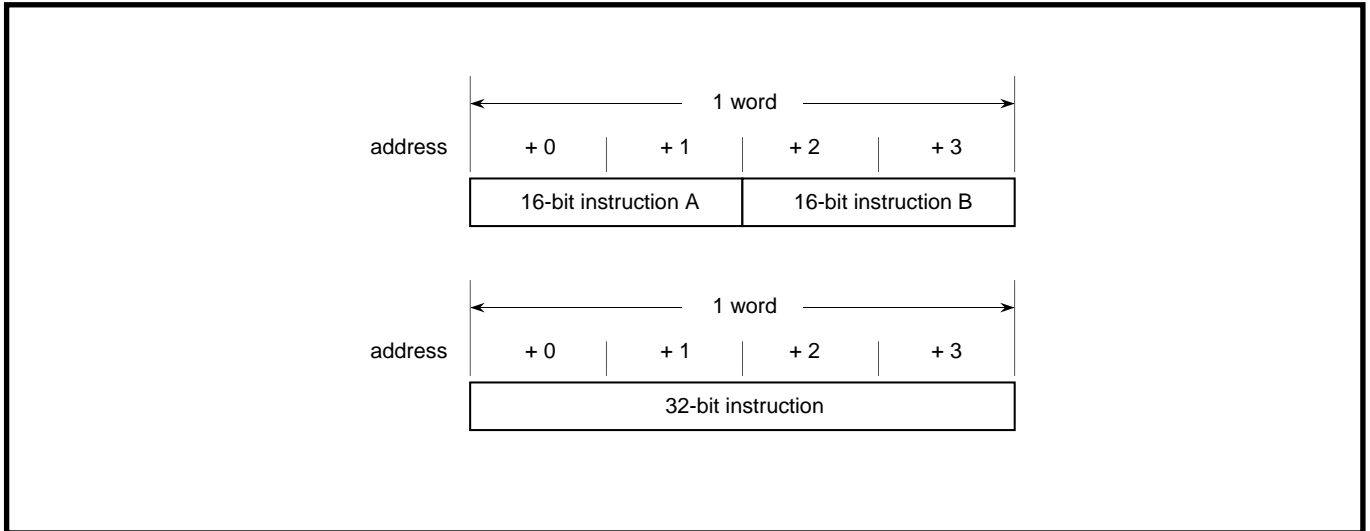


Fig. 2.2.1 16-bit instruction and 32-bit instruction

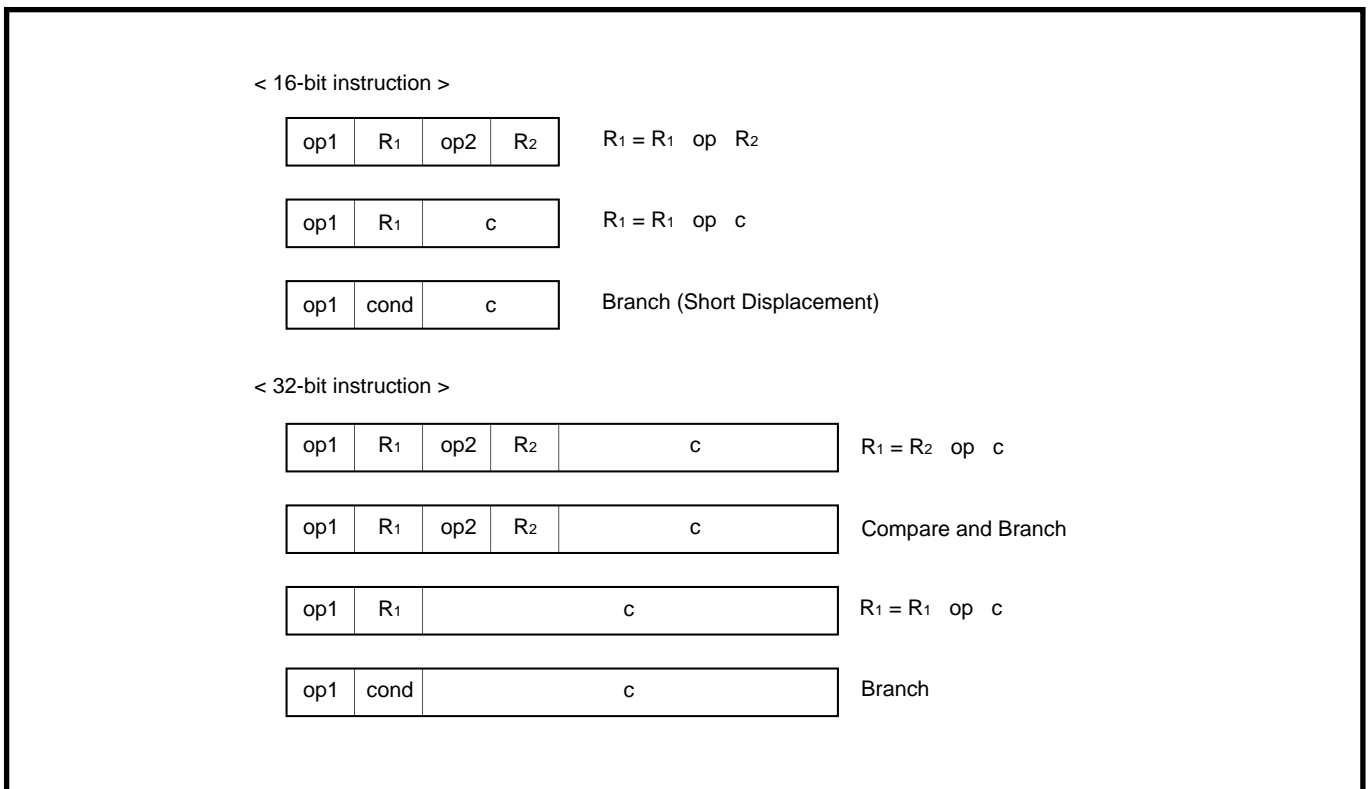


Fig. 2.2.2 Instruction format of M32R family

INSTRUCTION SET

2.2 Instruction format

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 can execute this instruction without requiring any clock cycles.

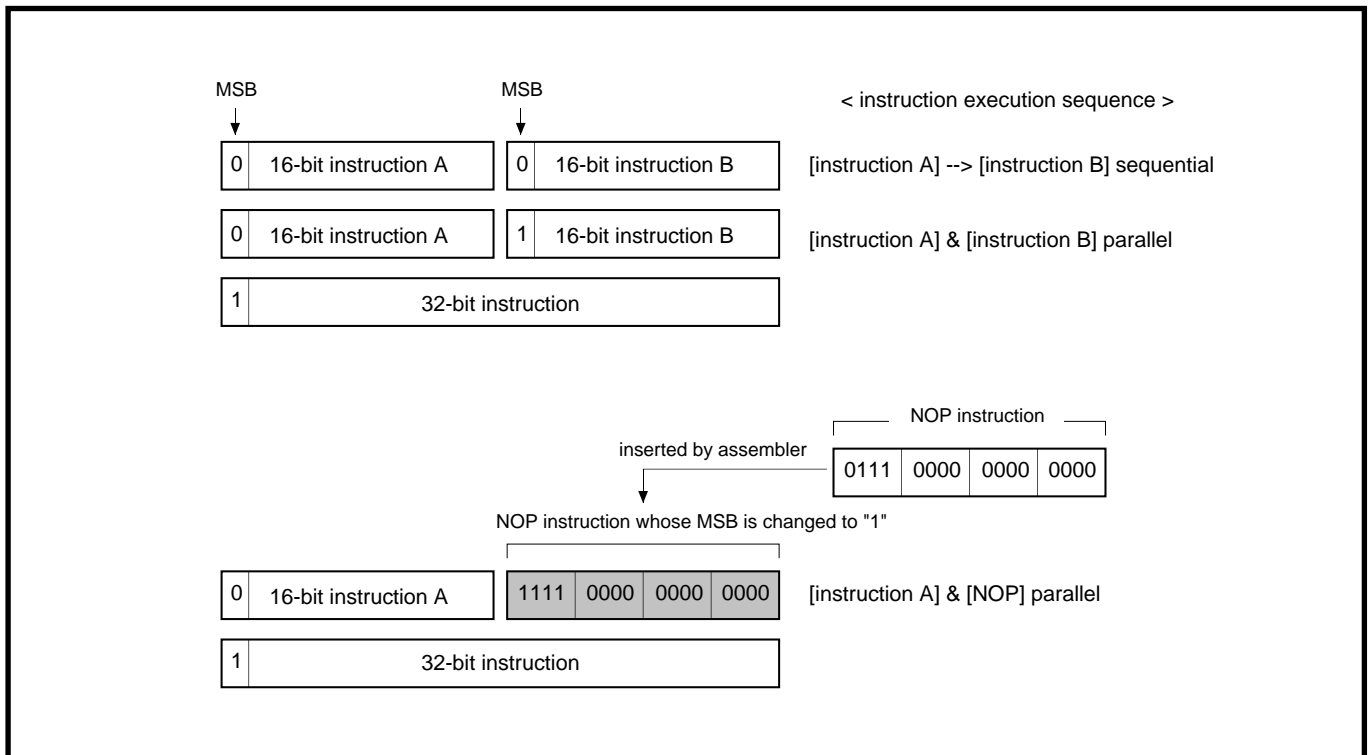


Fig. 2.2.3 Processing of 16-bit instructions



CHAPTER 3

INSTRUCTIONS

- 3.1 Conventions for instruction description
- 3.2 Instruction description

INSTRUCTIONS

3.1 Conventions for instruction description

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	control registers (CR0 = PSW, CR1 = CBR, CR2 = SPI, CR3 = SPU, CR6 = BPC)
@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	4 is added to register contents (memory specified by register contents before update as address)
@+R	register indirect and register update	4 is added to register contents (memory specified by register contents after update as address)
@-R	register indirect and register update	4 is subtracted from register contents (memory specified by register contents after update as address)
#imm	immediate	immediate value (refer to each instruction description)
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) 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)

INSTRUCTIONS

3.1 Conventions for instruction description

Table 3.1.3 Operation expression (operator) (cont.)

operator	meaning
/	division (binomial operator)
%	remainder operation (binomial operator)
++	increment (monomial operator)
--	decrement (monomial operator)
-	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	type	sign	bit length	range
char	integer	yes	8	-128 to +127
short	integer	yes	16	-32,768 to +32,767
int	integer	yes	32	-2,147,483,648 to +2,147,483,647
unsigned char	integer	no	8	0 to 255
unsigned short	integer	no	16	0 to 655,535
unsigned int	integer	no	32	0 to 4,294,967,295
signed64bit	integer	yes	64	signed 64-bit integer (with accumulator)

INSTRUCTIONS

3.1 Conventions for instruction description

[Description]

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

[EIT occurrence]

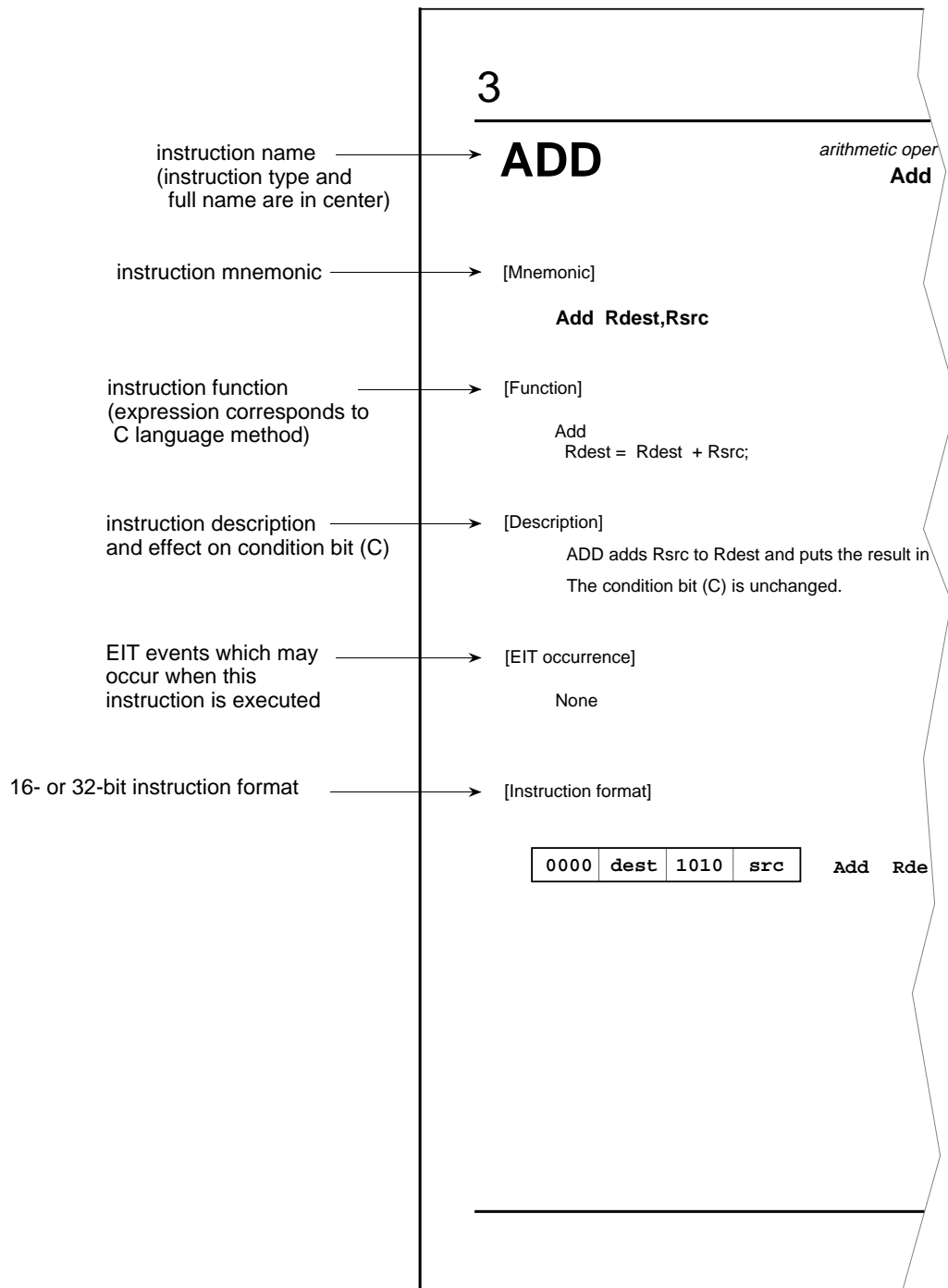
Shows possible EIT events (Exception, Interrupt, Trap) which may occur as the result of the instruction's execution. Only address exception (AE) and trap (TRAP) may result from an instruction execution.

[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 family instructions in alphabetical order. Each page is laid out as shown below.



INSTRUCTIONS

3.2 Instruction description

ADD

arithmetic/logic operation
Add

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

$Rdest = Rsrc + (\text{signed short}) \text{ imm16};$

[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`

INSTRUCTIONS

3.2 Instruction description

ADDI

arithmetic operation instruction
Add immediate

ADDI

[Mnemonic]

`ADDI Rdest, #imm8`

[Function]

Add
 $Rdest = Rdest + (\text{signed char}) \text{imm8};$

[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
 $Rdest = (\text{signed}) Rdest + (\text{signed}) Rsrc;$
 $C = \text{overflow} ? 1 : 0;$

[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

INSTRUCTIONS

3.2 Instruction description

ADDV3 *arithmetic operation instruction* 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

$Rdest = (\text{unsigned}) Rdest + (\text{unsigned}) Rsrc + C;$

$C = \text{carry_out} ? 1 : 0;$

[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	<i>dest</i>	1001	<i>src</i>
------	-------------	------	------------

ADDX *Rdest, Rsrc*

INSTRUCTIONS

3.2 Instruction description

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

INSTRUCTIONS

3.2 Instruction description

BC

branch instruction
Branch on C-bit

BC

[Mnemonic]

- ① BC pcdisp8
- ② BC pcdisp24

[Function]

Branch

- ① if (C==1) PC = (PC & 0xffffffc) + (((signed char) pcdisp8) << 2);
 - ② if (C==1) PC = (PC & 0xffffffc) + (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 |



# BEQ

*branch instruction*  
Branch on equal

# BEQ

### [Mnemonic]

**BEQ** *Rsrc1,Rsrc2,pcdisp16*

### [Function]

Branch

if ( *Rsrc1 == Rsrc2* )  $PC = ( PC \& 0xfffffc ) + ( ( \text{signed short } pcdisp16 ) \ll 2 );$

### [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 | <i>src1</i> | 0000 | <i>src2</i> | <i>pcdisp16</i> |  |  |  |
|------|-------------|------|-------------|-----------------|--|--|--|

**BEQ** *Rsrc1,Rsrc2,pcdisp16*

# INSTRUCTIONS

## 3.2 Instruction description

---

# BEQZ

*branch instruction*  
Branch on equal zero

# BEQZ

### [Mnemonic]

**BEQZ** *Rsrc,pcdisp16*

### [Function]

Branch

if ( *Rsrc* == 0 )  $PC = ( PC \& 0xfffffc ) + ( ( \text{signed short } pcdisp16 ) \ll 2 );$

### [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 zero

# BGEZ

### [Mnemonic]

**BGEZ** *Rsrc,pcdisp16*

### [Function]

Branch

if ( (signed) Rsrc >= 0 ) PC = ( PC & 0xfffffc ) + ( ( (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*

# INSTRUCTIONS

## 3.2 Instruction description

---

# BGTZ

*branch instruction*  
Branch on greater than zero

# BGTZ

### [Mnemonic]

**BGTZ** *Rsrc,pcdisp16*

### [Function]

Branch

if ((signed) *Rsrc* > 0) PC = (PC & 0xffffffc) + ( (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

# BL

### [Mnemonic]

- ① **BL** `pcdisp8`
- ② **BL** `pcdisp24`

### [Function]

Subroutine call (PC relative)

- ①  $R14 = ( PC \& 0\text{xffffffc} ) + 4;$   
 $PC = ( PC \& 0\text{xffffffc} ) + ( ( \text{signed char} ) \text{pcdisp8} ) \ll 2 ;$
- ②  $R14 = ( PC \& 0\text{xffffffc} ) + 4;$   
 $PC = ( PC \& 0\text{xffffffc} ) + ( \text{sign\_extend} ( \text{pcdisp24} ) \ll 2 );$   
 where  
 $\#define \text{sign\_extend}(x) ( ( \text{signed} ) ( x) \ll 8 ) \gg 8 )$

### [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 |

# INSTRUCTIONS

## 3.2 Instruction description

---

# BLEZ

*branch instruction*

Branch on less than or equal zero

# BLEZ

### [Mnemonic]

**BLEZ** *Rsrc,pcdisp16*

### [Function]

Branch

if ((signed) *Rsrc* <= 0)  $PC = (PC \& 0xffffffc) + (((\text{signed short}) \text{pcdisp16}) \ll 2);$

### [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)  $PC = (PC \& 0xfffffc) + (((signed\ short)\ pcdisp16) \ll 2);$

### [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*

# INSTRUCTIONS

## 3.2 Instruction description

---

# BNC

*branch instruction*  
Branch on not C-bit

# BNC

### [Mnemonic]

- ① **BNC** `pcdisp8`
- ② **BNC** `pcdisp24`

### [Function]

Branch

- ① if (C==0)  $PC = ( PC \& 0xfffffc ) + ( ( \text{signed char} ) \text{pcdisp8} ) \ll 2$  ;
  - ② if (C==0)  $PC = ( PC \& 0xfffffc ) + ( \text{sign\_extend} ( \text{pcdisp24} ) \ll 2$  ) ;
- 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	<code>pcdisp8</code>
1111	1101		pcdisp24	BNC	<code>pcdisp24</code>

BNE

branch instruction
Branch on not equal

BNE

[Mnemonic]

BNE *Rsrc1,Rsrc2,pcdisp16*

[Function]

Branch

if (*Rsrc1* != *Rsrc2*) PC = (PC & 0xfffffc) + (((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	<i>src1</i>	0001	<i>src2</i>	<i>pcdisp16</i>			
------	-------------	------	-------------	-----------------	--	--	--

BNE *Rsrc1,Rsrc2,pcdisp16*

INSTRUCTIONS

3.2 Instruction description

BNEZ

branch instruction
Branch on not equal zero

BNEZ

[Mnemonic]

BNEZ *Rsrc,pcdisp16*

[Function]

Branch

if (*Rsrc* != 0) $PC = (PC \& 0\text{xfffffc}) + ((\text{signed short } pcdisp16) \ll 2);$

[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 $PC = (PC \& 0xfffffc) + ((\text{signed char}) \text{pcdisp8}) \ll 2 ;$
- ② $PC = (PC \& 0xfffffc) + (\text{sign_extend} (\text{pcdisp24}) \ll 2) ;$
 where
 $\#define \text{sign_extend}(x) (((\text{signed}) (x) \ll 8) \gg 8)$

[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	<code>pcdisp8</code>	BRA	<code>pcdisp8</code>
1111	1111		BRA	<code>pcdisp24</code>

INSTRUCTIONS

3.2 Instruction description

CMP

compare instruction
Compare

CMP

[Mnemonic]

CMP **Rsrc1,Rsrc2**

[Function]

Compare

$C = ((\text{signed}) Rsrc1 < (\text{signed}) Rsrc2) ? 1:0;$

[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

$C = ((\text{signed}) Rsrc < (\text{signed short}) \text{imm16}) ? 1:0;$

[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

INSTRUCTIONS

3.2 Instruction description

CMPU

compare instruction
Compare unsigned

CMPU

[Mnemonic]

CMPU **Rsrc1,Rsrc2**

[Function]

Compare

$C = ((\text{unsigned}) Rsrc1 < (\text{unsigned}) Rsrc2) ? 1:0;$

[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

$C = ((\text{unsigned}) Rsrc < (\text{unsigned}) ((\text{signed short}) imm16)) ? 1:0;$

[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`

INSTRUCTIONS

3.2 Instruction description

DIV

multiply and divide instruction
Divide

DIV

[Mnemonic]

DIV *Rdest, Rsrc*

[Function]

Signed division

$Rdest = (\text{signed}) Rdest / (\text{signed}) Rsrc;$

[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

$Rdest = (\text{unsigned}) Rdest / (\text{unsigned}) Rsrc;$

[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**

INSTRUCTIONS

3.2 Instruction description

JL

branch instruction
Jump and link

JL

[Mnemonic]

JL Rsrc

[Function]

Subroutine call (register direct)
 $R14 = (PC \& 0\text{xfffffc}) + 4;$
 $PC = Rsrc \& 0\text{xfffffc};$

[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 & 0xfffffc;

[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**

INSTRUCTIONS

3.2 Instruction description

LD

load/store instruction
Load

LD

[Mnemonic]

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

[Function]

Load

- 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

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]



LD24 **Rdest, #imm24**

INSTRUCTIONS

3.2 Instruction description

LDB

load/store instruction
Load byte

LDB

[Mnemonic]

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

[Function]

Load

- 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		<code>LDB Rdest, @Rsrc</code>
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

- ① $Rdest = *(\text{signed short } *) Rsrc;$
- ② $Rdest = *(\text{signed short } *) (Rsrc + (\text{signed short }) \text{disp16});$

[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)

INSTRUCTIONS

3.2 Instruction description

LDI

transfer instruction
Load immediate

LDI

[Mnemonic]

- ① `LDI Rdest, #imm8`
- ② `LDI Rdest, #imm16`

[Function]

Load

- 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

- ① $Rdest = *(\text{unsigned char } *) Rsrc;$
- ② $Rdest = *(\text{unsigned char } *) (Rsrc + (\text{signed short }) \text{disp16});$

[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)**

INSTRUCTIONS

3.2 Instruction description

LDUH

load/store instruction
Load unsigned halfword

LDUH

[Mnemonic]

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

[Function]

Load

- 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

LOCK

[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 internal to the CPU and cannot be accessed directly except by using the **LOCK** or **UNLOCK** instructions.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	dest	1101	src
------	------	------	-----

`LOCK Rdest,@Rsrc`

INSTRUCTIONS

3.2 Instruction description

MACHI

DSP function instruction
Multiply-accumulate
high-order halfword

MACHI

[Mnemonic]

MACHI Rsrc1, Rsrc2

[Function]

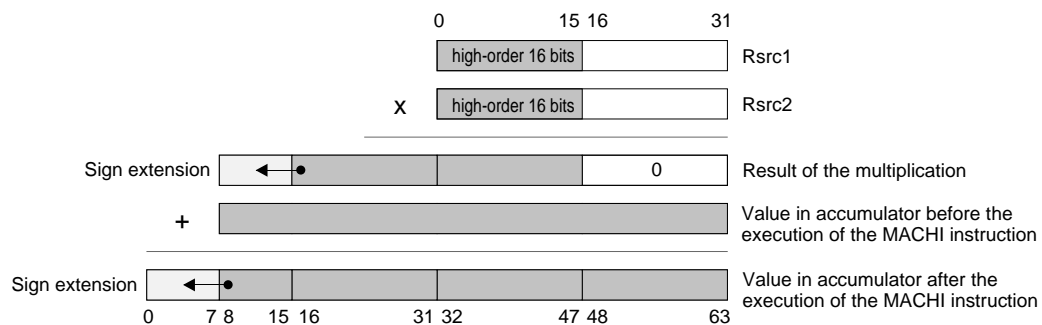
Multiply and add
accumulator += ((signed) (Rsrc1 & 0xffff0000) * (signed short) (Rsrc2 >> 16));

[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 halfword

MACLO

[Mnemonic]

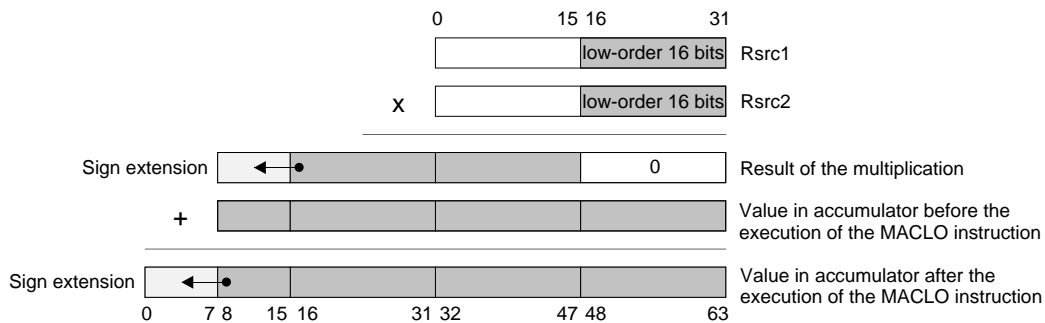
MACLO **Rsrc1, Rsrc2**

[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]



INSTRUCTIONS

3.2 Instruction description

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

[Mnemonic]

MACWHI Rsrc1,Rsrc2

[Function]

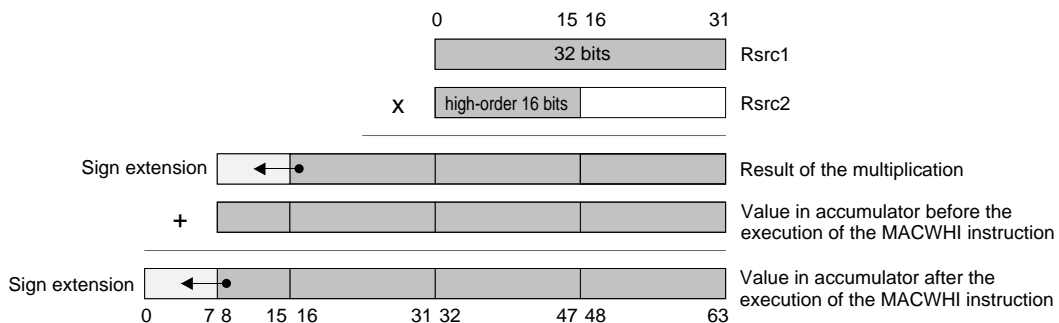
Multiply and add
accumulator += ((signed) Rsrc1 * (signed short) (Rsrc2 >> 16));

[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 MACWLO

Multiply-accumulate word and low-order halfword

[Mnemonic]

MACWLO **Rsrc1, Rsrc2**

[Function]

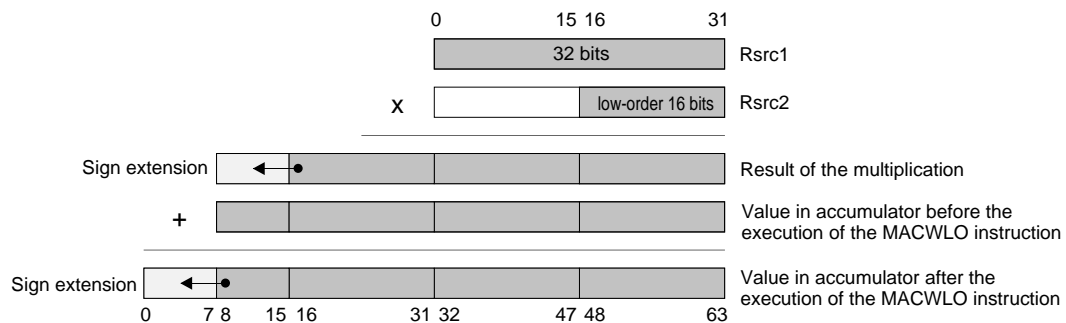
Multiply and add
 $\text{accumulator} += ((\text{signed}) \text{Rsrc1} * (\text{signed short}) \text{Rsrc2});$

[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
------	------	------	------	--------	--------------

INSTRUCTIONS

3.2 Instruction description

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 condition bit (C) is unchanged. The contents of the accumulator are destroyed by this instruction.

[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

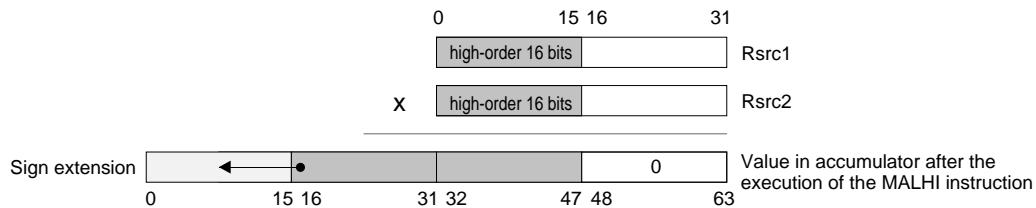
accumulator = ((signed) (Rsrc1 & 0xffff000) * (signed short) (Rsrc2 >> 16));

[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

INSTRUCTIONS

3.2 Instruction description

MULLO

DSP function instruction
Multiply low-order halfwords

MULLO

[Mnemonic]

MULLO Rsrc1,Rsrc2

[Function]

Multiply

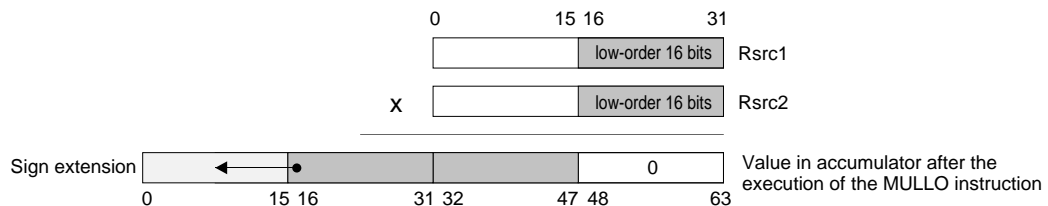
accumulator = ((signed) (Rsrc1 << 16) * (signed short) Rsrc2);

[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

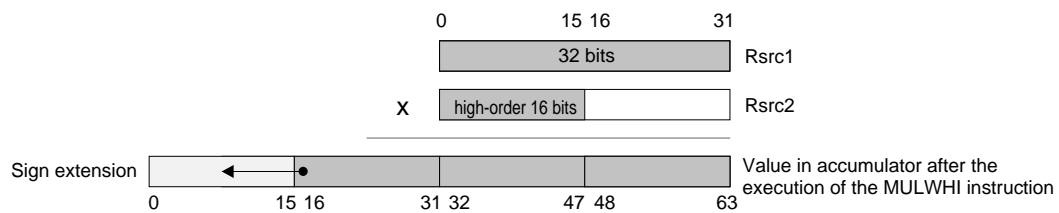
accumulator = ((signed) Rsrc1 * (signed short) (Rsrc2 >> 16));

[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
------	------	------	------	---------------	---------------------

INSTRUCTIONS

3.2 Instruction description

MULWLO

DSP function instruction
Multiply word and
low-order halfword

MULWLO

[Mnemonic]

MULWLO **Rsrc1,Rsrc2**

[Function]

Multiply

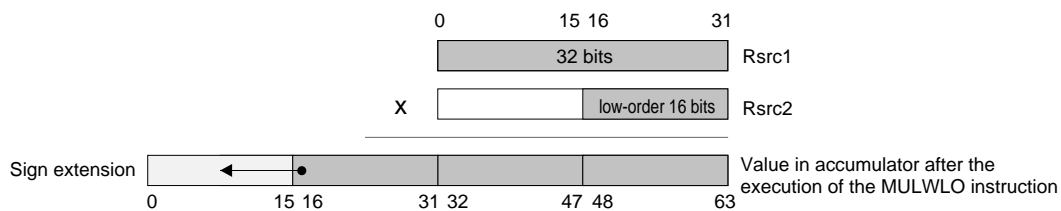
accumulator = ((signed) Rsrc1 * (signed short) Rsrc2);

[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**

INSTRUCTIONS

3.2 Instruction description

MVFACHI

DSP function instruction
Move from accumulator
high-order word

MVFACHI

[Mnemonic]

MVFACHI **Rdest**

[Function]

Transfer from accumulator to register
 $Rdest = (int) (accumulator \gg 32) ;$

[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 MVFACLO

Move from accumulator
low-order word

[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

INSTRUCTIONS

3.2 Instruction description

MVFACMI *DSP function instruction* MVFACMI

Move from accumulator
middle-order word

[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**

INSTRUCTIONS

3.2 Instruction description

MVTACHI

DSP function instruction
Move to accumulator
high-order word

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 to accumulator low-order word **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

INSTRUCTIONS

3.2 Instruction description

MVTC

transfer instruction
Move to control register

MVTC

[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*

INSTRUCTIONS

3.2 Instruction description

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
 $Rdest = \sim Rsrc$;

[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`

INSTRUCTIONS

3.2 Instruction description

OR

logic operation instruction
OR

OR

[Mnemonic]

OR Rdest, Rsrc

[Function]

Logical OR
 $Rdest = Rdest \mid Rsrc$;

[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

INSTRUCTIONS

3.2 Instruction description

RAC

DSP function instruction
Round accumulator

RAC

[Mnemonic]

RAC

[Function]

```
{ signed64bit tmp;  
if( 0x0000 3fff ffff 8000 =< accumulator )  
    tmp = 0x0000 3fff ffff 8000;  
else if( accumulator =< 0xffff c000 0000 0000 )  
    tmp = 0xffff c000 0000 0000;  
else {  
    tmp = accumulator + 0x0000 0000 0000 4000;  
    tmp = tmp & 0xffff ffff ffff 8000;}  
accumulator = tmp << 1;}
```

[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
------	------	------	------

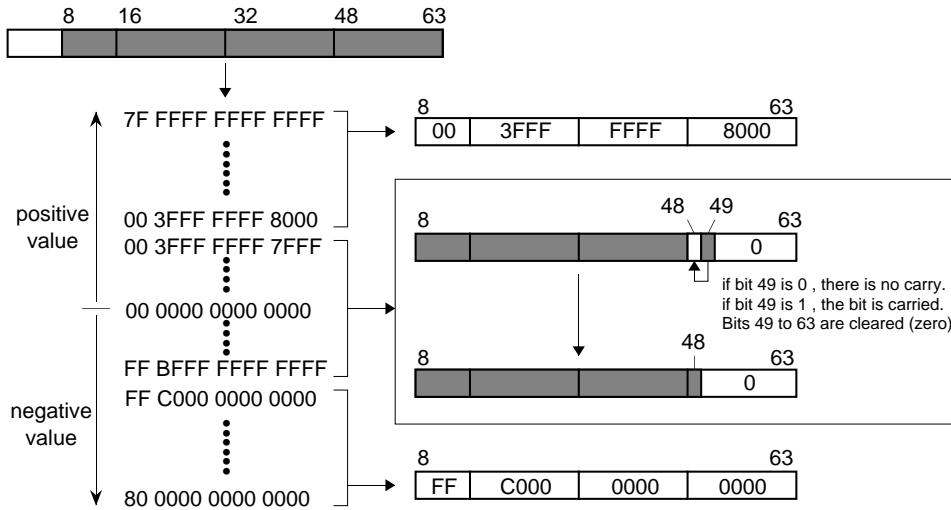
 RAC

[Supplement]

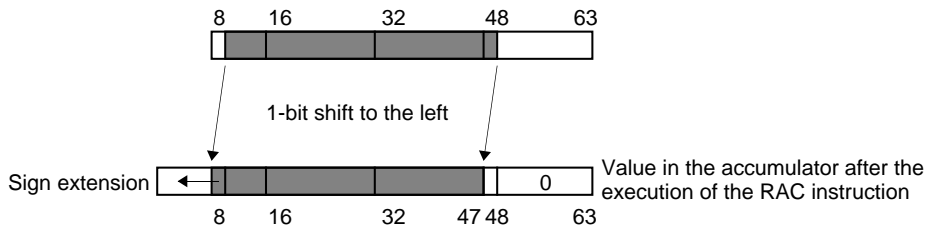
This instruction is executed in two steps as shown below:

<step 1>

The value in the accumulator is altered depending on the value of bits 8 through 63.



<step 2>



INSTRUCTIONS

3.2 Instruction description

RACH

DSP function instruction
Round accumulator halfword

RACH

[Mnemonic]

RACH

[Function]

```
{ signed64bit tmp;  
if( 0x0000 3fff 8000 0000 =< accumulator )  
    tmp = 0x0000 3fff 8000 0000;  
else if( accumulator =< 0xffff c000 0000 0000 )  
    tmp = 0xffff c000 0000 0000;  
else {  
    tmp = accumulator + 0x0000 0000 4000 0000;  
    tmp = tmp & 0xffff ffff 8000 0000;}  
accumulator = tmp << 1;}
```

[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
------	------	------	------

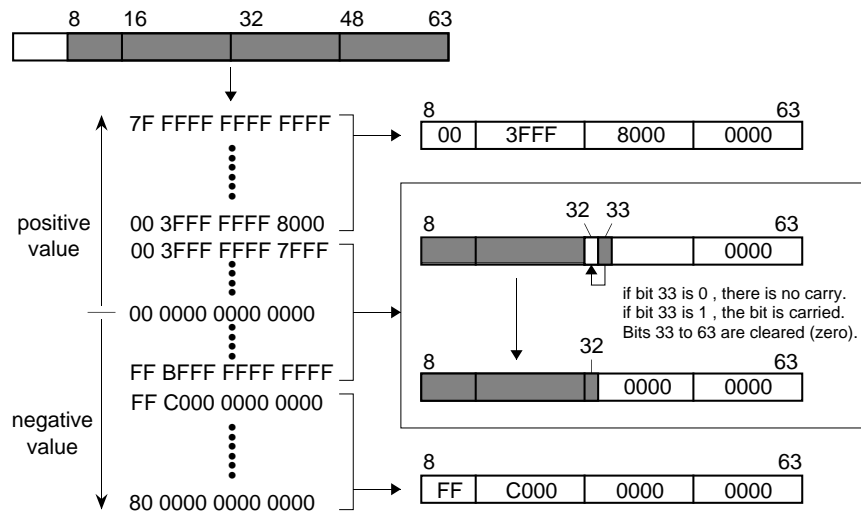
RACH

[Supplement]

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

<step 1>

Value in the accumulator is altered depending on the value of bits 8 through 63.



<step 2>



INSTRUCTIONS

3.2 Instruction description

REM

multiply and divide instruction
Remainder

REM

[Mnemonic]

REM **Rdest, Rsrc**

[Function]

Signed division

$Rdest = (\text{signed}) Rdest \% (\text{signed}) Rsrc ;$

[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 division

$Rdest = (\text{unsigned}) Rdest \% (\text{unsigned}) Rsrc ;$

[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**

INSTRUCTIONS

3.2 Instruction description

RTE

EIT-related instruction
Return from EIT

RTE

[Mnemonic]

RTE

[Function]

Return from EIT
SM = BSM ;
IE = BIE ;
C = BC ;
PC = BPC & 0xffffffc ;

[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.

[EIT occurrence]

None

[Encoding]

0001	0000	1101	0110
------	------	------	------

RTE

SETH

transfer instruction
Set high-order 16-bit

SETH

[Mnemonic]

SETH *Rdest*, #imm16

[Function]

Transfer instructions
 $Rdest = (short) imm16 \ll 16 ;$

[Description]

SETH loads 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

INSTRUCTIONS

3.2 Instruction description

SLL

shift instruction
Shift left logical

SLL

[Mnemonic]

SLL *Rdest, Rsrc*

[Function]

Logical left shift
 $Rdest = Rdest \ll (Rsrc \& 31) ;$

[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
 $Rdest = Rsrc \ll (imm16 \& 31) ;$

[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`

INSTRUCTIONS

3.2 Instruction description

SLLI

shift instruction
Shift left logical immediate

SLLI

[Mnemonic]

`SLLI Rdest, #imm5`

[Function]

Logical left shift
 $Rdest = Rdest \ll imm5$;

[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

$Rdest = (\text{signed}) Rdest \gg (Rsrc \& 31) ;$

[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**

INSTRUCTIONS

3.2 Instruction description

SRA3

shift instruction
Shift right arithmetic 3-operand

SRA3

[Mnemonic]

SRA3 **Rdest,Rsrc,#imm16**

[Function]

Arithmetic right shift

$Rdest = (\text{signed}) Rsrc \gg (\text{imm16} \& 31) ;$

[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
 $Rdest = (\text{signed}) Rdest \gg imm5$;

[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**

INSTRUCTIONS

3.2 Instruction description

SRL

shift instruction
Shift right logical

SRL

[Mnemonic]

SRL *Rdest, Rsrc*

[Function]

Logical right shift

$Rdest = (\text{unsigned}) Rdest \gg (Rsrc \& 31) ;$

[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

$Rdest = (\text{unsigned}) Rsrc \gg (\text{imm16} \& 31) ;$

[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**

INSTRUCTIONS

3.2 Instruction description

SRLI

shift instruction
Shift right logical immediate

SRLI

[Mnemonic]

SRLI **Rdest, #imm5**

[Function]

Logical right shift

$Rdest = (\text{unsigned}) Rdest \gg (\text{imm5} \& 31) ;$

[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)

INSTRUCTIONS

3.2 Instruction description

[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		<code>STB Rsrc1,@Rsrc2</code>
1010	src1	0000	src2	disp16	

`STB Rsrc1,@(disp16,Rsrc2)`

INSTRUCTIONS

3.2 Instruction description

STH

load/store instruction
Store halfword

STH

[Mnemonic]

- ① **STH** **Rsrc1,@Rsrc2**
- ② **STH** **Rsrc1,@(disp16,Rsrc2)**

[Function]

Store

- ① * (short *) Rsrc2 = Rsrc1;
- ② * (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 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
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`

INSTRUCTIONS

3.2 Instruction description

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

$Rdest = (\text{unsigned}) Rdest - (\text{unsigned}) Rsrc - C;$

$C = \text{borrow} ? 1 : 0;$

[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**

INSTRUCTIONS

3.2 Instruction description

TRAP

EIT-related instruction
Trap

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.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	src1	0101	src2
------	------	------	------

 UNLOCK Rsrc1,@Rsrc2

INSTRUCTIONS

3.2 Instruction description

XOR

logic operation instruction
Exclusive OR

XOR

[Mnemonic]

`XOR Rdest, Rsrc`

[Function]

Exclusive OR

$Rdest = (\text{unsigned}) Rdest \wedge (\text{unsigned}) Rsrc;$

[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

$Rdest = (\text{unsigned}) Rsrc \wedge (\text{unsigned short}) imm16;$

[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

Appendix A Instruction list

Appendix B Pipeline stages

Appendix C Instruction execution time

APPENDICES

Appendix A Instruction list

Appendix A Instruction list

The M32R family 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)	–
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)	–
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	–
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	–

APPENDICES

Appendix A Instruction list

mnemonic	function		condition bit (C)
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 = *(uh*)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 = accumulator >> 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
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 & 0xfffffff, PSW[SM,IE,C] = PSW[BSM,BIE,BC]	change

APPENDICES

Appnedix A Instruction list

mnemonic	function		condition bit (C)
SETH	Rdest,#imm16	Rdest = imm16 << 16	-
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	-
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	-
XOR	Rdest,Rsrc	Rdest = Rdest ^ Rsrc	-
XOR3	Rdest,Rsrc,#imm16	Rdest = Rsrc ^ (uh)imm16	-

where:

```
typedef signed 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 B Pipeline stages

B.1 Overview of pipeline processing

The M32R CPU has five pipeline stages.

(1) 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.

(2) 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.

(3) E stage (execution stage)

Operations and address calculations (OP) are processed in the E stage.

(4) 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.

(5) WB stage (write back stage)

The operation results and fetched data are written to the registers in the WB stage.

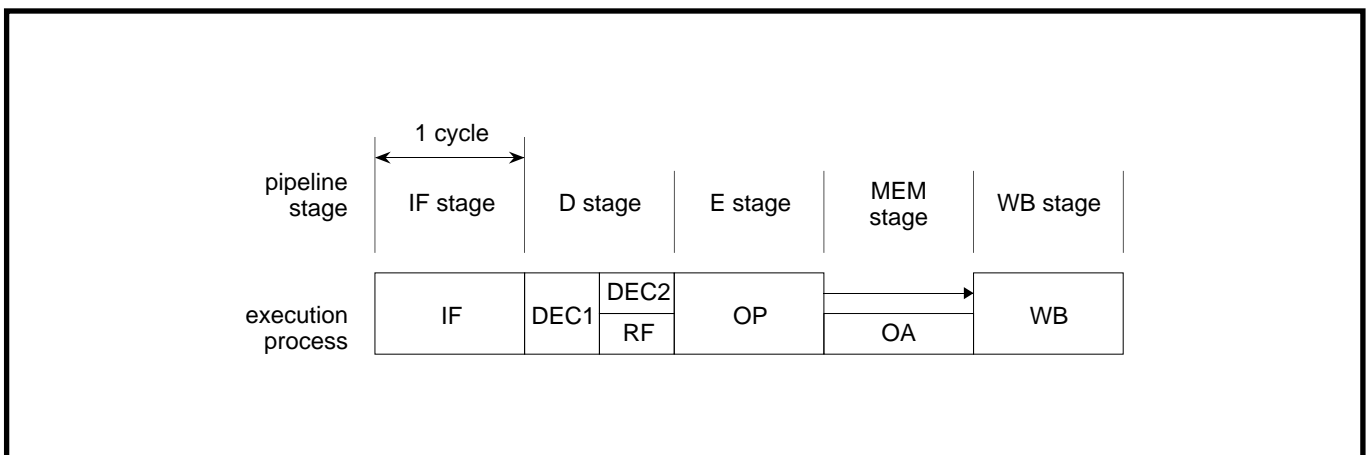


Fig. B.1 Pipeline structure and processing

APPENDICES

Appendix B Pipeline stages

B.2 Instructions and pipeline processing

The M32R pipeline has five stages. However, the MEM stage is used only when the load/store instruction is executed, other instructions are processed in a 4-stage pipeline.

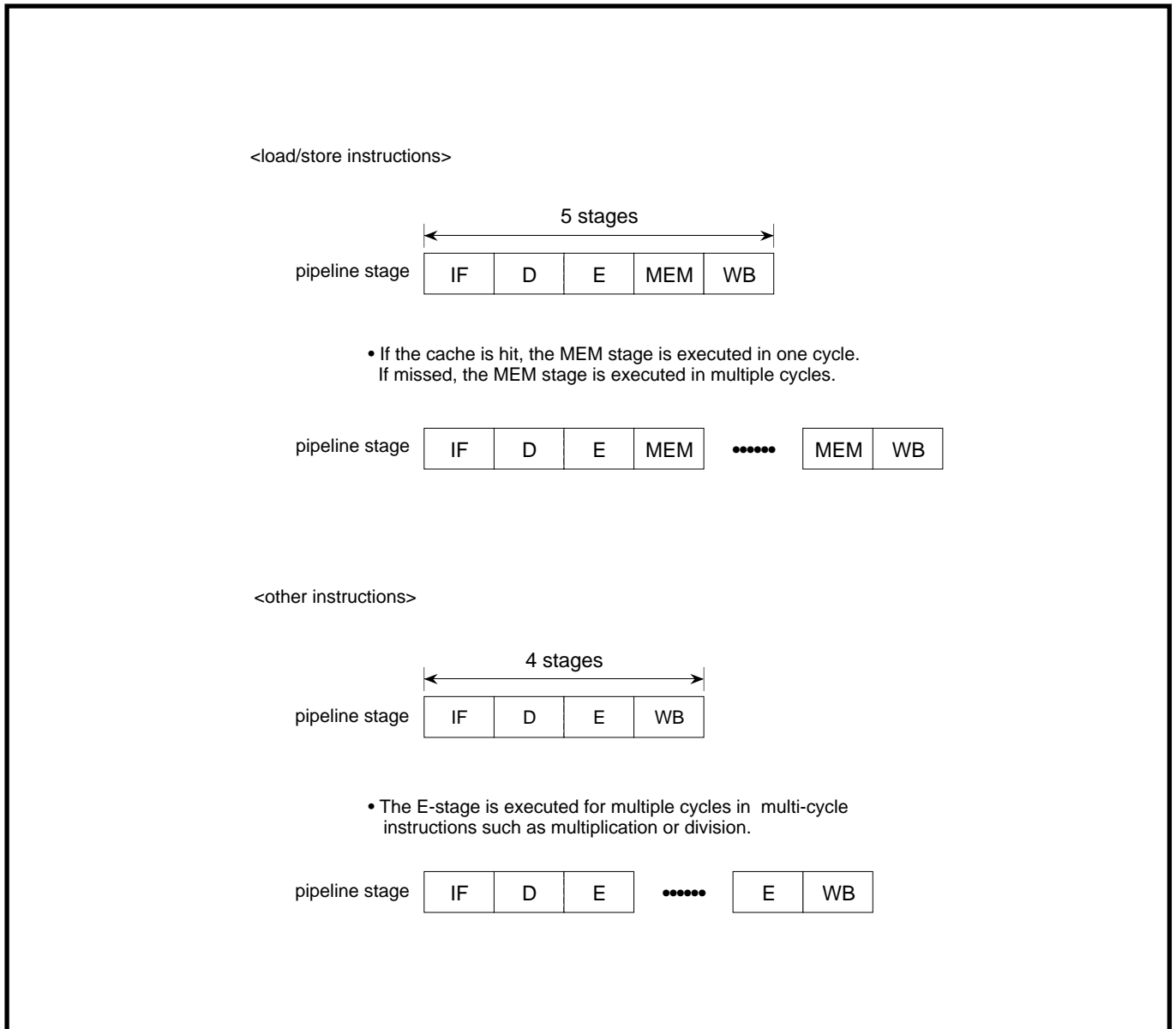
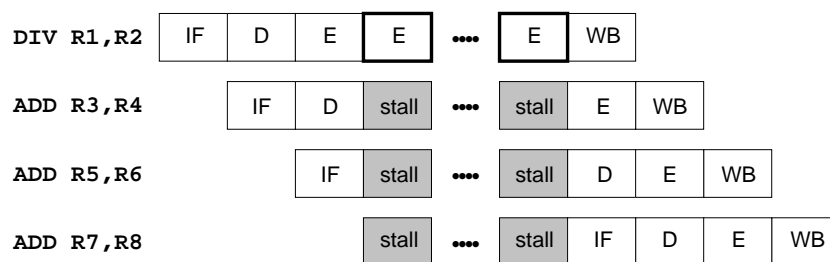


Fig. B.2 Instructions and pipeline processing

B.3 Pipeline processing

In perfect pipeline processing, each stage is executed in one cycle. However, the pipeline stall may be caused at each stage of processing or by the execution of a branch instruction. Each case is described in Figure B.3 and B.4.

< case 1 multiple cycles are required for the E-stage execution >



< case 2 operand access is not complete in one cycle >

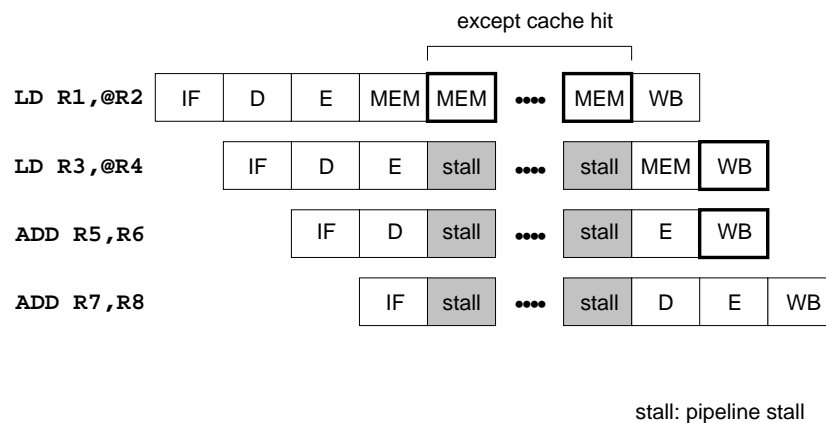
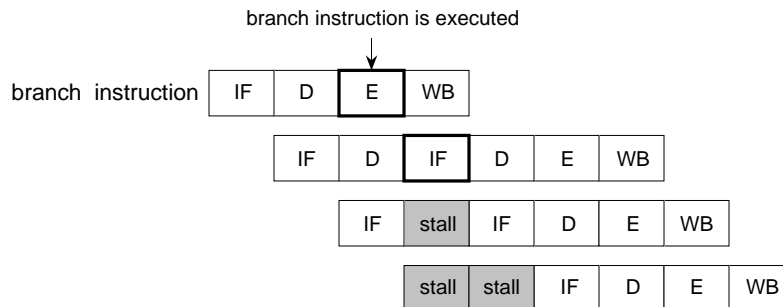


Fig. B.3 Pipeline stall 1

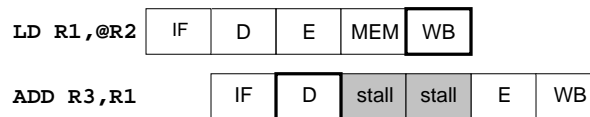
APPENDICES

Appendix B Pipeline stages

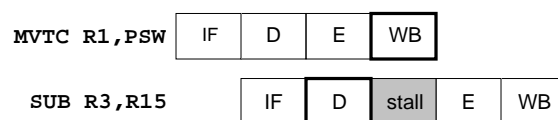
< case 3 branch instruction is executed >
 (except for the case where no branch occurs at a conditional branch instruction)



< case 4 the subsequent instruction uses an operand read from memory >



< case 5 R15 is read after the SM bit in the PSW is written by an MVTC instruction and the subsequent instruction reads R15 >

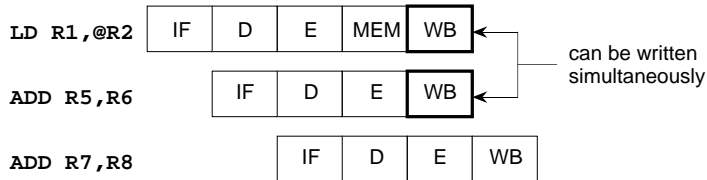


stall: pipeline stall

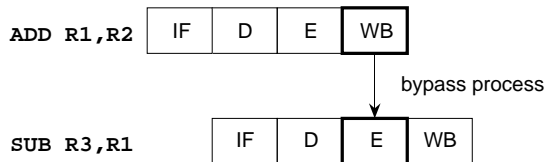
Fig. B.4 Pipeline stall 2

The cases shown in Figure B.5 are special and pipeline stall does not occur.

<when the WB stages of load and another instruction occur simultaneously>
(pipeline processing is not stalled because the values can be written simultaneously)



<when the register written by the one instruction is used by the subsequent instruction>
(the pipeline processing is not stalled because of the bypass process due to operation between registers)



<a subsequent instruction writes to a register before a load instruction is completed>
(the WB stage of the load instruction is canceled)

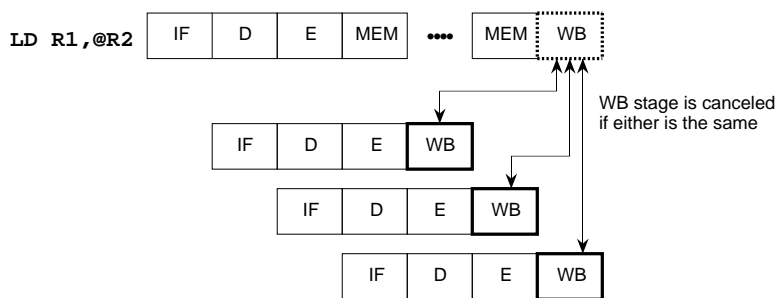


Fig. B.5 Special case (pipeline stall does not occur)

APPENDICES

Appendix C Instruction execution time

Appendix C 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.

Table C.1 Instruction execution cycles in each stage

instruction	the number of execution cycles in each stage				
	IF	D	E	MEM	WB
load instruction (LD, LDB, LDUB, LDH, LDUH, LOCK)	R (note 1)	1	1	R (note 1)	1
store instruction (ST, STB, STH, UNLOCK)	R (note 1)	1	1	W (note 1)	(1) (note 2)
multiply instruction (MUL)	R (note 1)	1	3	–	1
divide/remainder instruction (DIV, DIVU, REM, REMU)	R (note 1)	1	37	–	1
other instructions	R (note 1)	1	1	–	1

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

2 If the addressing mode of the store instructions is register indirect and register update, 1 cycle needs for WB stage.

REVISION DESCRIPTION LIST

M32R family software manual

Rev. No.	Revision Description	Rev. date
1.0	First edition	970331
1.1	<ul style="list-style-type: none"> • "Only a word-aligned (word boundary) address can be specified for the branch address. <u>If an unaligned address is specified, an address exception occurs.</u>" an underlined part eliminated. (line 18, page 2-6) • [Encoding] "OR3 Rdest, Rsrc, #imm16" revised. (line 13, page 3-63) 	971031
1.2	<ul style="list-style-type: none"> • "ADDV3 Add 3-operand with overflow checking" revised (line 21, page 2-4) • "ADDV3 Add 3-operand with overflow checking" revised (line 2, page 3-10) • Logical right shift "Rdest = (unsigned) Rsrc >> (imm16 & 31);" revised (line 7, page 3-79) • "Trap occurrence BPC = PC + 4; BSM = SM; BIE = IE; BC = C ; IE = 0; C = 0; call_trap_handler(imm4);" revised (line 7, page 3-88) • "BL pcdisp24 R14=PC+4,PC=PC+((s24)pcdisp24<<2)" revised (line 19, page A-2) • "BNC pcdisp24 if(!C) PC=PC+((s24)pcdisp24<<2)" revised (line 23, page A-2) • "TRAP #n PSW[BSM,BIE,BC] = PSW[SM,IE,C] change PSW[SM,IE,C] = PSW[SM,0,0] Call trap-handler number-n" revised (line 23, page A-4) 	980701

MITSUBISHI 32-BIT SINGLE-CHIP MICROCOMPUTER
M32R Family Software Manual

July 1998 : Revised edition

Copyright (C) 1998 MITSUBISHI ELECTRIC CORPORATION

Notice:

This book, or parts thereof, may not be reproduced in any form
without permission of MITSUBISHI ELECTRIC CORPORATION.

Renesas Technology Corp.

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

Printed in Japan

(C) 1998 MITSUBISHI ELECTRIC CORPORATION

Revised publication, effective July 1998
Specifications subject to change without notice.