REJ09B0135-0100Z

32

OPSP
Software Manual

RENESAS 32-BIT OPEN PLATFORM SYNTHESIZABLE PROCESSOR

Software Manual

Rev.1.00
Revision date : Mar 01,2004

Renesas Technology
www.renesas.com

Keep safety first in your circuit designs!

1. Renesas Technology 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 nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

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

REVISION HISTORY	OPSP Software Manual

Rev.	Date	Description	
		Page	Summary
1.00	Mar 01,2004	-	First edition issued

This page is blank for reasons of layout.

Table of contents

CHAPTER 1 CPU PROGRAMMING MODEL

CHAPTER 2 INSTRUCTION SET

CHAPTER 3 INSTRUCTIONS

APPENDICES

CHAPTER 1
CPU PROGRAMMING MODEL

1.1 Processor Modes

The OPSP-CPU core (hereafter abbreviated “OPSP-CPU”) provides two processor modes: Supervisor Mode and User Mode. A hierarchical resource protection mechanism can be realized by using these processor modes. Each processor mode has designated rights with respect to memory access and executable instructions, which are higher for supervisor mode than for user mode.

When an EIT event occurs, the CPU goes to supervisor mode. The processor mode in which the CPU was immediately before the EIT event occurred is stored in the backup PM (BPM) bit of the Processor Status Word Register (PSW). When the RTE instruction is executed, the CPU returns to the previous processor mode that is stored in the BPM bit.

1.1.1 Privileged Instructions

Privileged instructions are those that can only be executed in supervisor mode. If a privileged instruction is executed in user mode, a privileged instruction exception occurs. The privileged instructions include RTE, MVTC, SETPSW, and CLRPSW.

1.2 CPU Registers

The OPSP-CPU has 16 general-purpose registers, 6 control registers, 2 accumulators, and a program counter. The accumulators are configured with 64 bits, while all other registers are configured with 32 bits.

1.3 General-purpose Registers

The general-purpose registers are 32 bits wide, and there are 16 of them (R0 to R15). These registers are used to hold data and base addresses. Of these, R14 and R15 are used as a link register and a stack pointer (SPI or SPU), respectively. The link register is used to hold the return address when executing a subroutine call instruction. The stack pointer is switched between a stack pointer for interrupt (SPI) and a stack pointer for user (SPU) depending on the value of the SM bit in the Processor Status Word Register (PSW).

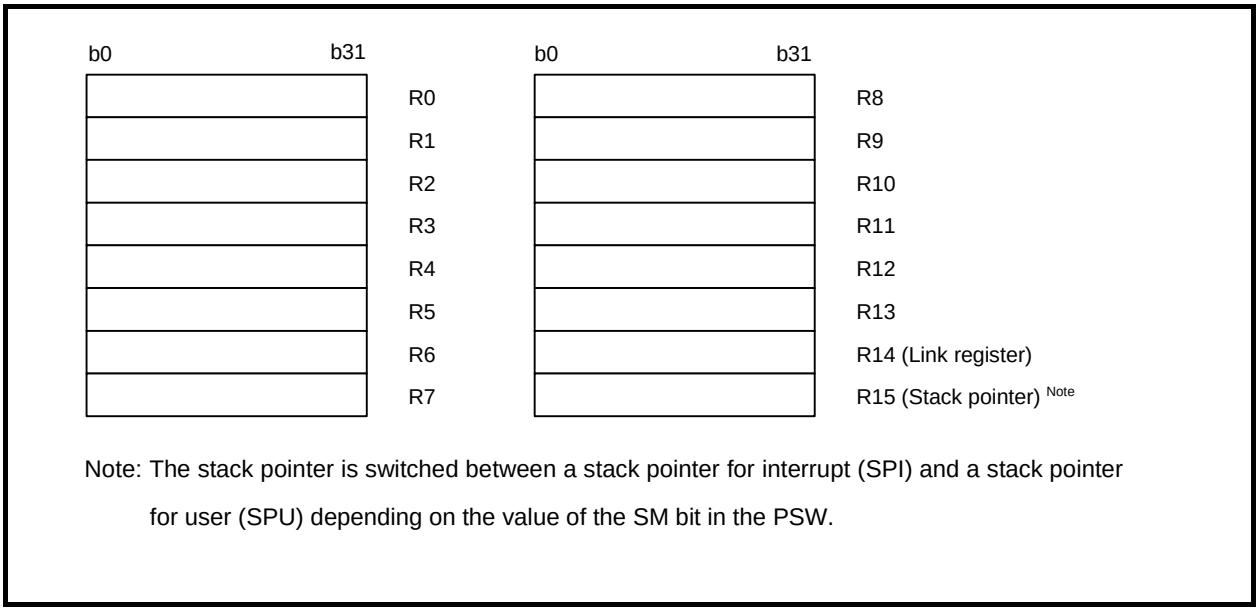


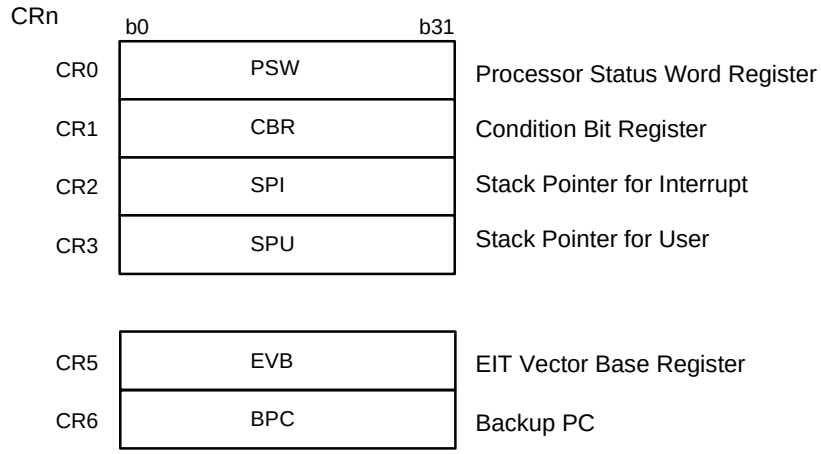
Figure 1.3.1 General-purpose Registers

1.4 Control Registers

There are six control registers: Processor Status Word Register (PSW), Condition Bit Register (C), Stack Pointer for Interrupt (SPI), Stack Pointer for User (SPU), EIT Vector Base Register (EVB), and Backup PC (BPC).

Dedicated MVTC and MVFC instructions are used to set and read these control registers. Furthermore, SETPSW and CLRPSW instructions can be used for the PSW.

MVTC, SETPSW, and CLRPSW are the privileged instructions that can only be executed when the CPU is operating in supervisor mode. Which processor mode is active is determined by the processor mode (PM) bit in the Processor Status Word Register (PSW).



Note 1: CRn (n = 0–3, 5, 6) denotes a control register number.

Note 2: Dedicated MVTC and MVFC instructions are used to set and read the control registers.

Figure 1.4.1 Control Registers

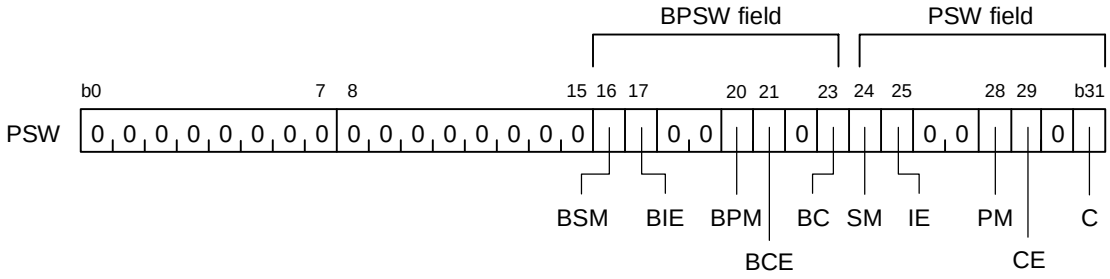
1.4.1 Processor Status Word Register: PSW (CR0)

The Processor Status Word Register (PSW) indicates the status of the OPSP-CPU. It consists of two bit fields: the PSW field that is normally used, and the BPSW field in which the PSW field is saved when an EIT occurs.

The PSW field further consists of the stack mode bit (SM), interrupt enable bit (IE), processor mode bit (PM), coprocessor interrupt enable bit (CE), and condition bit (C). Similarly, the BPSW field consists of the backup SM bit (BSM), backup IE bit (BIE), backup PM bit (BPM), backup CE bit (BCE), and backup C bit (BC).

After reset, the BSM, BIE, BPM, BCE, and BC are indeterminate. All other bits are 0.

To switch the processor mode, set BPM = 1 using the MVTC instruction and then execute the RTE instruction to branch to the user space(H'0000 0000 – H'7FFF FFFF). If the PM bit needs to be altered directly with the MVTC instruction, always be sure to alter it in the user space.



<After reset: "B'0000 0000 0000 0000 ??00 ??0? 0000 0000">

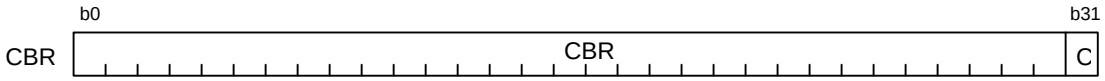
b	Bit Name	Function	R	W
0–15	No functions assigned. Fix these bits to 0.		0	0
16	BSM Backup SM bit	Save the value of the SM bit when an EIT is accepted.	R	W
17	BIE Backup IE bit	Save the value of the IE bit when an EIT is accepted.	R	W
18–19	No functions assigned. Fix these bits to 0.		0	0
20	BPM Backup PM bit	Save the value of the PM bit when an EIT is accepted.	R	W
21	BCE Backup CE bit	Save the value of the CE bit when an EIT is accepted.	R	W
22	No functions assigned. Fix these bits to 0.		0	0
23	BC Backup C bit	Save the value of the C bit when an EIT is accepted.	R	W
24	SM Stack mode bit	0: Stack pointer for interrupt is used. 1: Stack pointer for user is used.	R	W
25	IE Interrupt enable bit	0: Interrupt acceptance disabled 1: Interrupt acceptance enabled	R	W
26–27	No functions assigned. Fix these bits to 0.		0	0
28	PM Processor mode bit	0: Supervisor mode 1: User mode	R	W
29	CE Coprocessor interrupt enable bit	0: Coprocessor interrupt not accepted 1: Coprocessor interrupt accepted	R	W

30	No functions assigned. Fix these bits to 0.		0	0
31	C Condition bit	Indicate whether instruction execution resulted in a carry, borrow, or overflow.	R	W

1.4.2 Condition Bit Register: CBR (CR1)

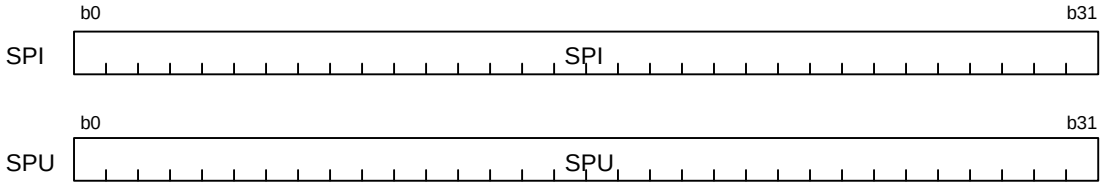
The Condition Bit Register (CBR) is derived from the condition bit (C) of the PSW to serve as a separate register. The value written to the condition bit in the PSW is reflected in this register. This register can only be read. (Writing to this register with the MVTC instruction is ignored.)

After reset, the CBR is H'0000 0000.



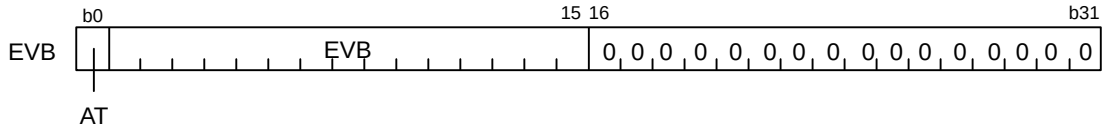
1.4.3 Stack Pointer for Interrupt: SPI (CR2) and Stack Pointer for User: SPU (CR3)

The Stack Pointer for Interrupt (SPI) and the Stack Pointer for User (SPU) hold the address of the current stack pointer. These registers can be accessed as the general-purpose register R15. Whether R15 is used as the SPI or as the SPU is determined by the stack mode bit (SM) in the PSW.



1.4.4 EIT Vector Base Register: EVB (CR5)

The EIT Vector Base Register (EVB) holds the EIT vector entry start address. The 16 high-order bits of the EIT vector entry start address comprise the value of the 16 high-order bits in this register.



<After reset: H'0000 0000>

b	Bit Name	Function	R	W
0	AT Address translation mode bit	Address translation mode	R	N
1–15	EVB Vector base bit	Set A1–A15 of EIT vector entry in these bits.	R	W
16–31	No functions assigned. Fix these bits to 0.		0	0

(1) AT (address translation mode) bit (b0)

This bit is a copy of the address translation mode bit (AT) in the MATM register, and is a read-only bit.

(2) EVB (EIT vector base) bits (b1–b15)

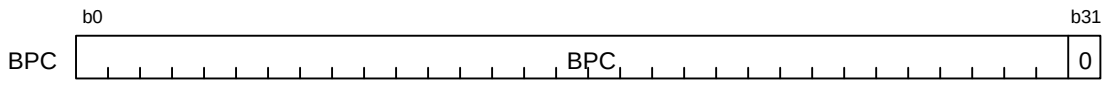
These bits set the EIT vector entry start address A1–A15. However, the reset interrupt (RI) vector is located at the address H'0000 0000 no matter how the EIT vector base bits are set.

Note: The EVB register can be set only once immediately after reset. Write to the EVB register should be performed at the beginning of a reset handler.

1.4.5 Backup PC: BPC (CR6)

The Backup PC (BPC) is used to save the value of the program counter (PC) when an EIT occurs. Bit 31 is fixed to 0.

When an EIT occurs, the PC value at which the EIT occurred or the PC value for the next instruction is set in the BPC depending on the type of the EIT that occurred. The value of the BPC is returned to the PC when the RTE instruction is executed.



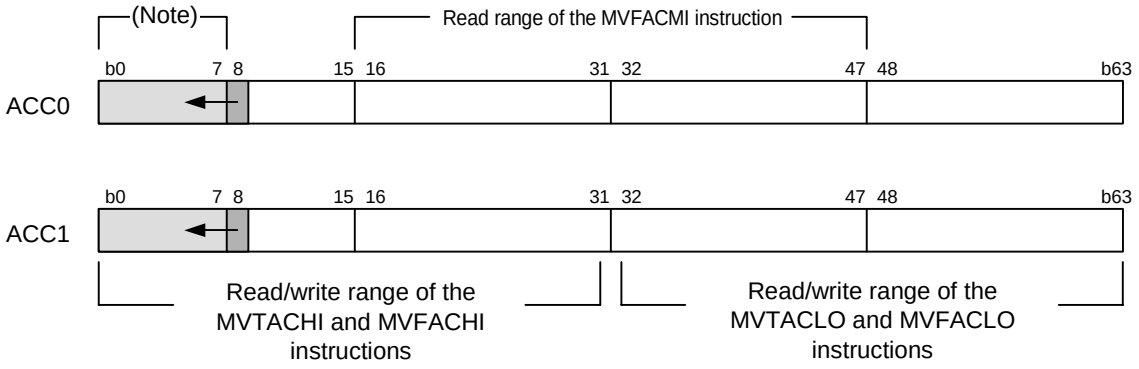
1.5 Accumulators

The accumulator is a 56-bit register used in the instructions for the DSP function. There are two of such accumulators, ACC0 and ACC1. During read or write, the accumulator is handled as a 64-bit register. In this case, bits 0–7 in the accumulator are sign-extended with the value of bit 8 during read, and are ignored during write. The accumulator is also used in the multiplication instruction “MUL.” Be aware that when this instruction is executed, the value of the accumulator, whether ACC0 or ACC1, is destroyed.

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

Use the MVFACHI, MVFACLO, and MVFACMI instructions to read the accumulator. The MVFACHI, MVFACLO, and MVFACMI instructions read data from the 32 high-order bits (bits 0–31), the 32 low-order bits (bits 32–63), and the 32 middle bits (bits 16–47) in the accumulator, respectively.

After reset, ACC0 and ACC1 are indeterminate.

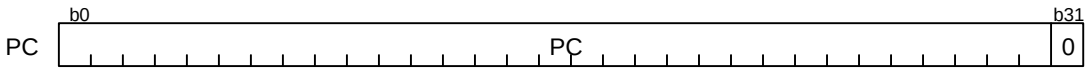


Note: Bits 0–7 when read always show the value that is sign-extended with the value of bit 8. Write to this bit field is ignored.

1.6 Program Counter (PC)

The Program Counter (PC) is a 32-bit counter that holds the address of the currently executed instruction. Since the instructions in the OPSP-CPU begin from even addresses, the LSB (bit 31) in the PC is always 0.

After reset, the PC is H’0000 0000.



1.7 Data Formats

1.7.1 Bi-endian Function

The OPSP-CPU supports the bi-endian function that allows either data format, big endian or little endian, to be adopted.

This manual is written for operation in big endian mode.

1.7.2 Data Types

The data types that the instruction set of the OPSP-CPU can handle are signed or unsigned 8, 16, and 32-bit integers. Signed integer values are represented by the 2's complement.

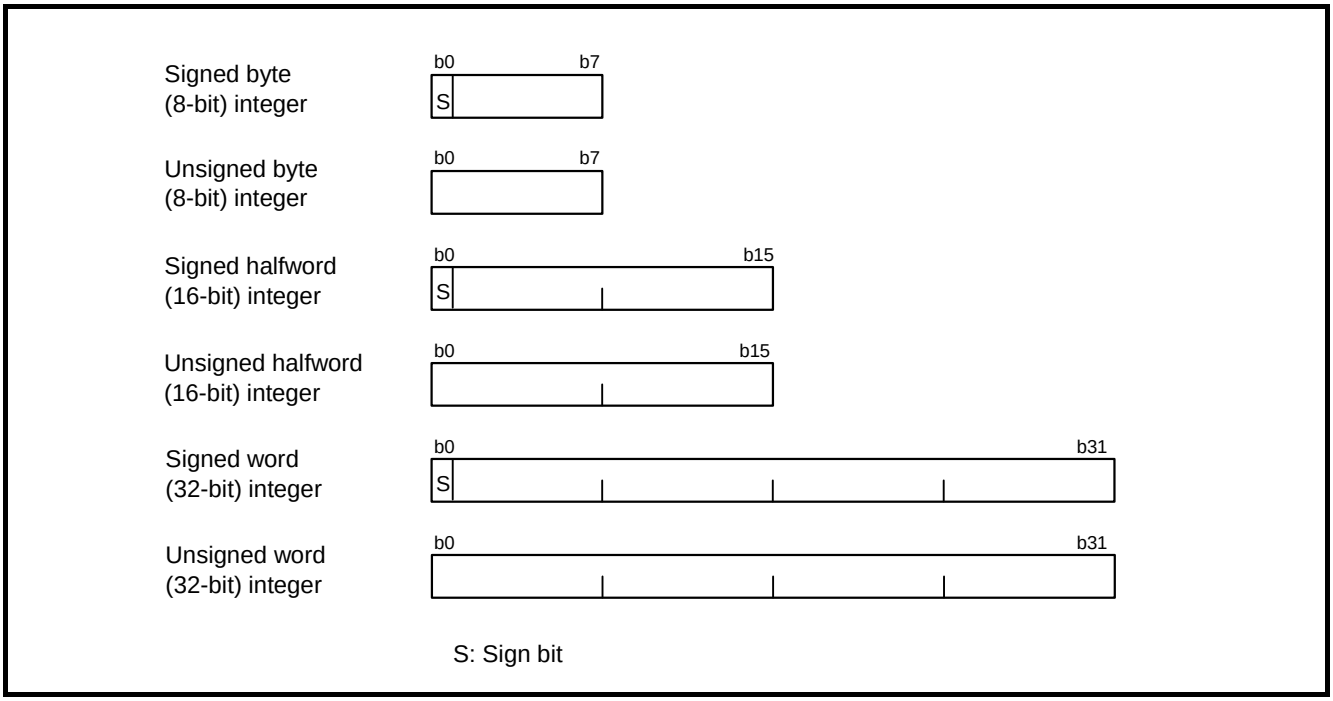


Figure 1.7.1 Data Types

1.7.3 Data Formats

(1) Data formats in the OPSP-CPU registers

The data size in the OPSP-CPU registers is always the word (32 bits). When byte (8-bit) or halfword (16-bit) data in memory is loaded into a register, the data is sign-extended (LDB, LDH instructions) or zero-extended (LDUB, LDUH instructions) to the word (32-bit) quantity before being stored in the register.

When data in an OPSP-CPU register is stored into memory, the ST, STH, or STB instruction is used. The ST, STH, and STB instructions store the full 32-bit data, the lower 16-bit data, or the least significant 8-bit data of the register in memory, respectively.

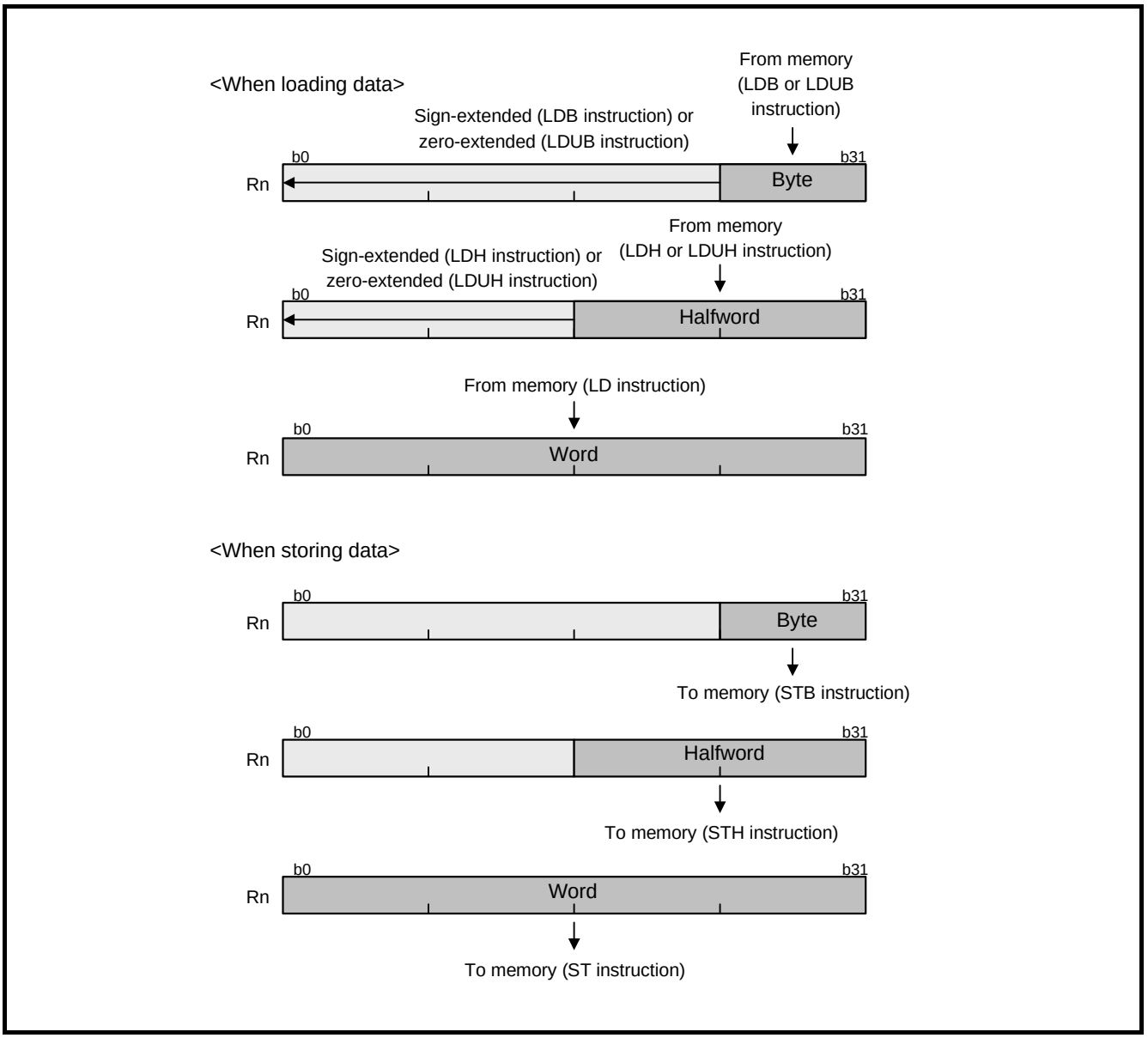


Figure 1.7.2 Data Formats in Registers

(2) Data formats in memory

The data in memory has one of three data sizes: byte (8 bits), halfword (16 bits), or word (32 bits). Although byte data can be located at any address, halfword and word data must be located at halfword-aligned addresses (least significant address bit = 0) and word-aligned addresses (two least significant address bits = 00), respectively. If access to misaligned memory data is attempted, an address exception occurs.

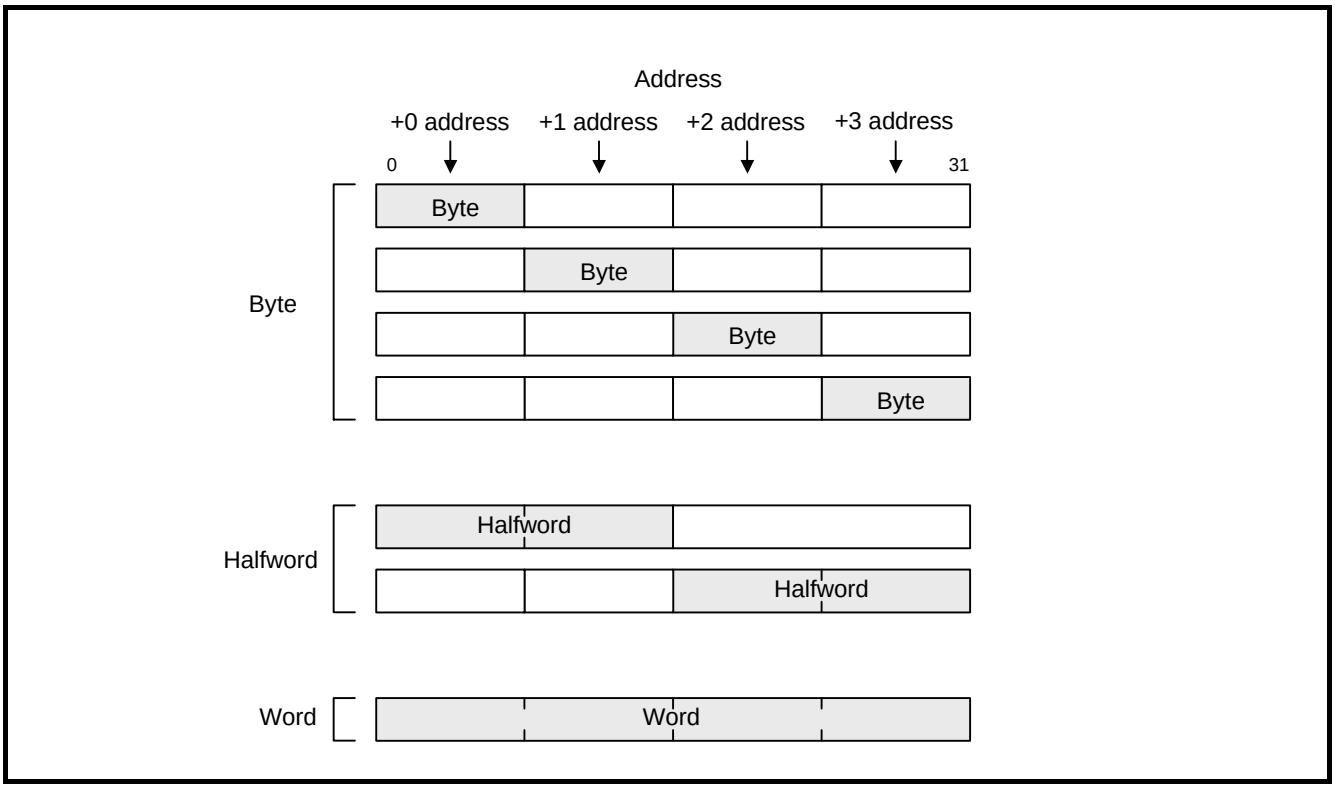


Figure 1.7.3 Data Formats in Memory

1.8 Addressing Modes

The OPSP-CPU has the following addressing modes:

(1) Register direct [expressed as R or CR or A[Note]]

A general-purpose or control register or an accumulator is specified directly as the target to be operated on.

(2) Register indirect [expressed as @R]

The address is indicated indirectly by a register value. (This addressing mode can be specified in all load and store instructions.)

(3) Register relative indirect (expressed as @(disp,R)]

The address is indicated indirectly by (register value) + (16-bit displacement which is sign-extended to 32 bits).

(4) Register indirect + register update

- Register value incremented by 1
  The address is indicated by a preupdate register value (specifiable in only STB instruction)
- Register value incremented by 2
  The address is indicated by a preupdate register value (specifiable in only STH instruction)
- Register value incremented by 4
  The address is indicated by a preupdate register value (specifiable in only LD instruction)
- Register value incremented by 4
  The address is indicated by an updated register value (specifiable in only ST instruction)
- Register value decremented by 4
  The address is indicated by an updated register value (specifiable in only ST instruction)

(5) Immediate [expressed as #imm]

1, 4, 5, 8, 16, or 24-bit immediate value. (For details on how the value is handled, refer to the detailed description of each instruction in the latter part of this manual.)

(6) PC relative [expressed as pcdisp]

The address is indicated by (PC value) + (8, 16, or 24-bit displacement which is sign-extended to 32 bits and then shifted left 2 bits).

Note: The accumulators ACC0 and ACC1 are mnemonically expressed as A0 and A1, respectively.

This page is blank for reasons of layout.

CHAPTER 2
INSTRUCTION SET

2.1 Outline of the Instruction Set

The OPSP-CPU has 115 distinct instructions. A RISC architecture is adopted for the OPSP-CPU, so that memory access basically is accomplished by using load and store instructions. Arithmetic/logical operations are executed by register-to-register operation. Furthermore, compound instructions such as Load & Address Update and Store & Address Update are supported.

2.2 Instruction Set

The instruction set of the OPSP-CPU is shown below.

New instructions that have been added in the OPSP-CPU from the M32R family instruction set are marked by double asterisks (**), and function-extended instructions are marked by a single asterisk (*).

2.2.1 Load and Store Instructions (10 instructions)

These instructions perform data transfer between memory and a register.

	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

Following three addressing modes can be specified in the load and store instructions.

(1) Register indirect

The address is indicated indirectly by a register value. (This addressing mode can be specified in all load and store instructions.)

(2) Register relative indirect

The address is indicated indirectly by (register value) + (16-bit displacement which is sign-extended to 32 bits). (This addressing mode can be specified in all load and store instructions other than LOCK and UNLOCK.)

(3) Register indirect + register update

- Register value incremented by 1
  The address is indicated by a preupdate register value (specifiable in only STB instruction)
- Register value incremented by 2
  The address is indicated by a preupdate register value (specifiable in only STH instruction)
- Register value incremented by 4
  The address is indicated by a preupdate register value (specifiable in only LD instruction)
- Register value incremented by 4
  The address is indicated by an updated register value (specifiable in only ST instruction)
- Register value decremented by 4
  The address is indicated by an updated register value (specifiable in only ST instruction)

Whichever addressing mode is used, rules for the data formats in memory must be observed. To access halfword or word data, a halfword aligned or word aligned address must be specified, respectively. (The two least significant bits of the accessed address must be “00” or “10” for halfword data, or “00” for word data.) If a misaligned address is specified, an address exception occurs.

If byte or halfword data is accessed in a load instruction, the data has its high order bits sign or zero-extended to become 32-bit data before being stored in a register.

2.2.2 Transfer Instructions (6 instructions)

These instructions perform a register to register transfer or a register to immediate transfer.

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

2.2.3 Arithmetic/Logical Instructions (46 instructions)

These instructions perform register to register comparison, arithmetic/logical operation, multiplication/division, or shift operation.

■ Comparison (7 instructions)

	CMP	Compare
**	CMPEQ	Compare equal to
	CMPI	Compare immediate
	CMPU	Compare unsigned
	CMPUI	Compare unsigned immediate
**	CMPZ	Compare equal to zero
**	PCMPBZ	Parallel compare byte to zero

■ Arithmetic operation (10 instructions)

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

■ Logical operation (7 instructions)

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

■ Multiplication/division (13 instructions)

	DIV	Divide
**	DIVB	Divide byte
**	DIVH	Divide halfword
	DIVU	Divide unsigned
**	DIVUB	Divide unsigned byte
**	DIVUH	Divide unsigned halfword
	MUL	Multiply
	REM	Reminder
**	REMB	Reminder byte
**	REMH	Reminder halfword
	REMU	Reminder unsigned
**	REMUB	Reminder unsigned byte
**	REMUH	Reminder unsigned halfword

■ Shift (9 instructions)

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

2.2.4 Branch Instructions (21 instructions)

These instructions are used to change the program flow.

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

Only the word aligned addresses (those aligned with word boundaries) can be specified as the jump address.

For the BRA, BL, BC, BNC, BCL and BNCL instructions, an 8-bit or 24-bit immediate value can be specified in addressing mode. For the BEQ, BNE, BEQZ, BNEZ, BGTZ, BLTZ, BGEZ and BLEZ instructions, a 16-bit immediate value should be specified in addressing mode.

For the JMP, JL, JC and JNC instructions, the jump address is specified by a register value. However, the two least significant address bits are ignored.

For the SC and SNC instructions, the jump address is indicated by (PC value of the branch instruction) + 4.

For other branch instructions, the jump address is indicated by (PC value of branch instruction) + (sign-extended immediate value that is shifted two bits left). However, the two least significant bits of the PC value are cleared to 0 when an addition is performed. In Figure2.2.1, for example, assume that instruction A or instruction B is the branch instruction, and that the program is to jump to instruction G. Then the immediate value, in either case, is 4.

For the JL, BL, BCL and BNCL instructions that are used for subroutine calls, the PC value for the return address is stored in R14 at the same time the program branches off. The value stored in R14 is (PC value of branch instruction + 4), with the two least significant bits of the PC value cleared to 0. In Figure2.2.1, for example, assume that instruction A or instruction B is the JL, BL, BCL, or BNCL instruction. Then the return address, in either case, is the “instruction C.”

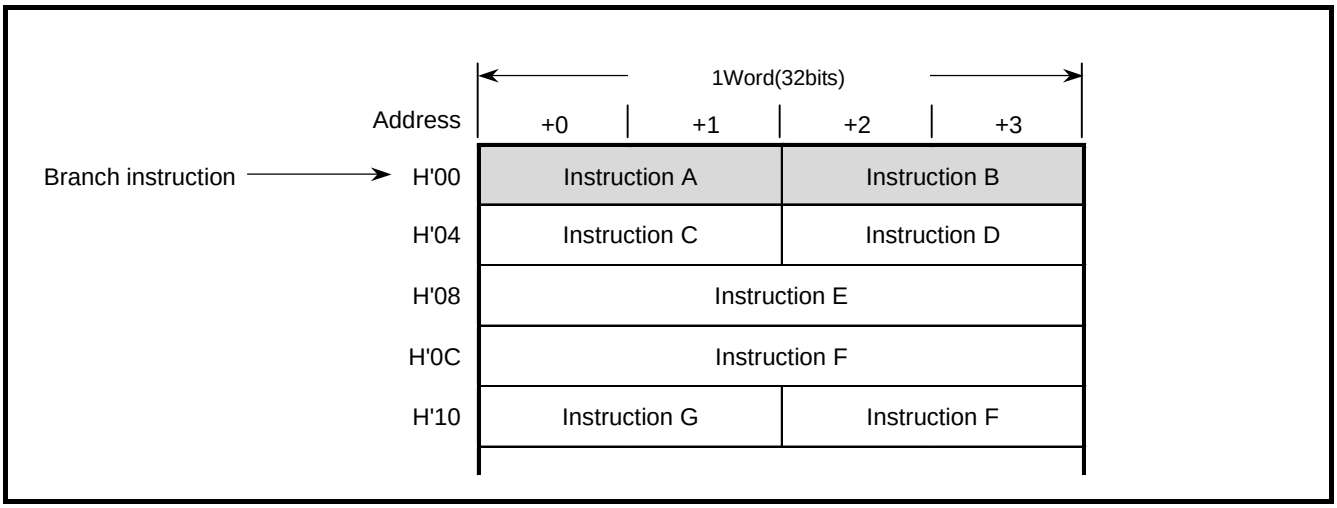


Figure2.2.1 Jump Address of a Branch Instruction

2.2.5 Bit Manipulating Instructions (5 instructions)

These instructions set or clear the bits in memory or registers and those in the Processor Status Word Register (PSW).

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

2.2.6 EIT Related Instructions (2 instructions)

These instructions are provided for EIT events (Exception, Interrupt and Trap). These include an instruction to invoke a trap and an instruction to return from EIT handling.

*	TRAP	Trap
*	RTE	Return from EIT

2.2.7 DSP Function Instructions (22 instructions)

In the OPSP-CPU, the DPS function instructions of the M32R family instruction set have been extended as follows:

- ■ There are two accumulators, compared to one in the past.
- ■ Multiply-accumulate operations are enhanced.
- ■ New general-purpose register rounding instructions are added.

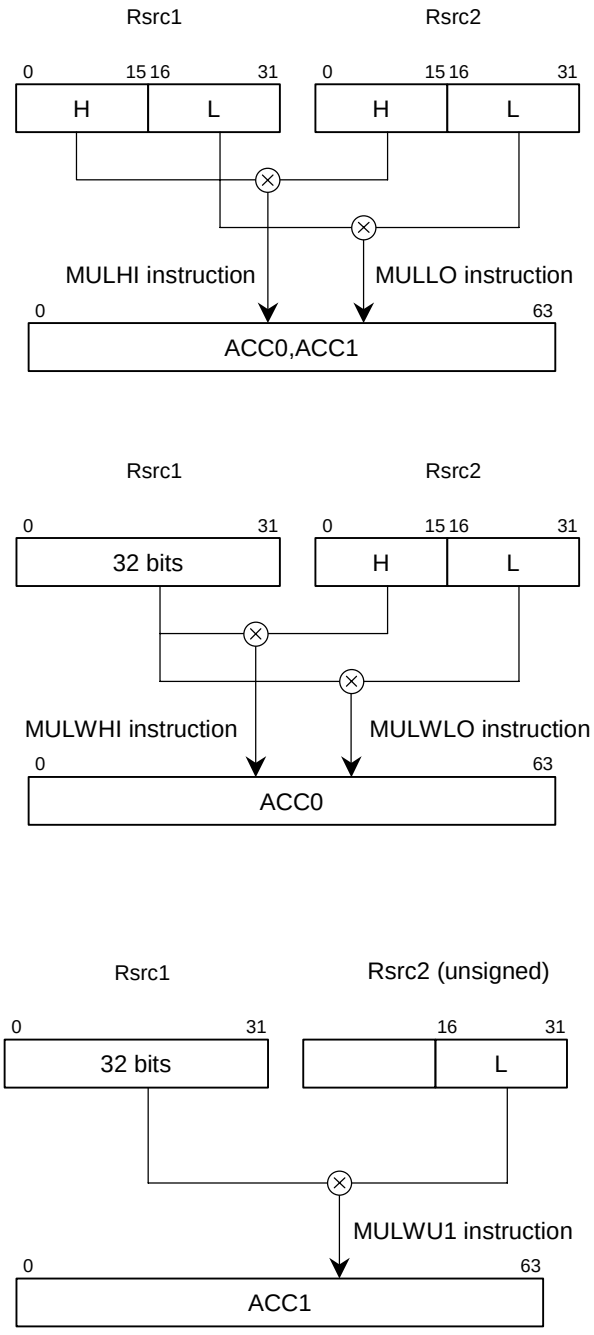
The DPS function instructions of the OPSP-CPU are shown below.

New instructions that have been added in the OPSP-CPU from the M32R family instruction set are marked by double asterisks (**), and function-extended instructions are marked by a single asterisk (*).

These instructions include those that perform 32 bit × 16 bit or 16 bit × 16 bit multiply or multiply-accumulate operations. Also included are those that round the data in an accumulator or general-purpose register or perform data transfer between an accumulator and general-purpose register.

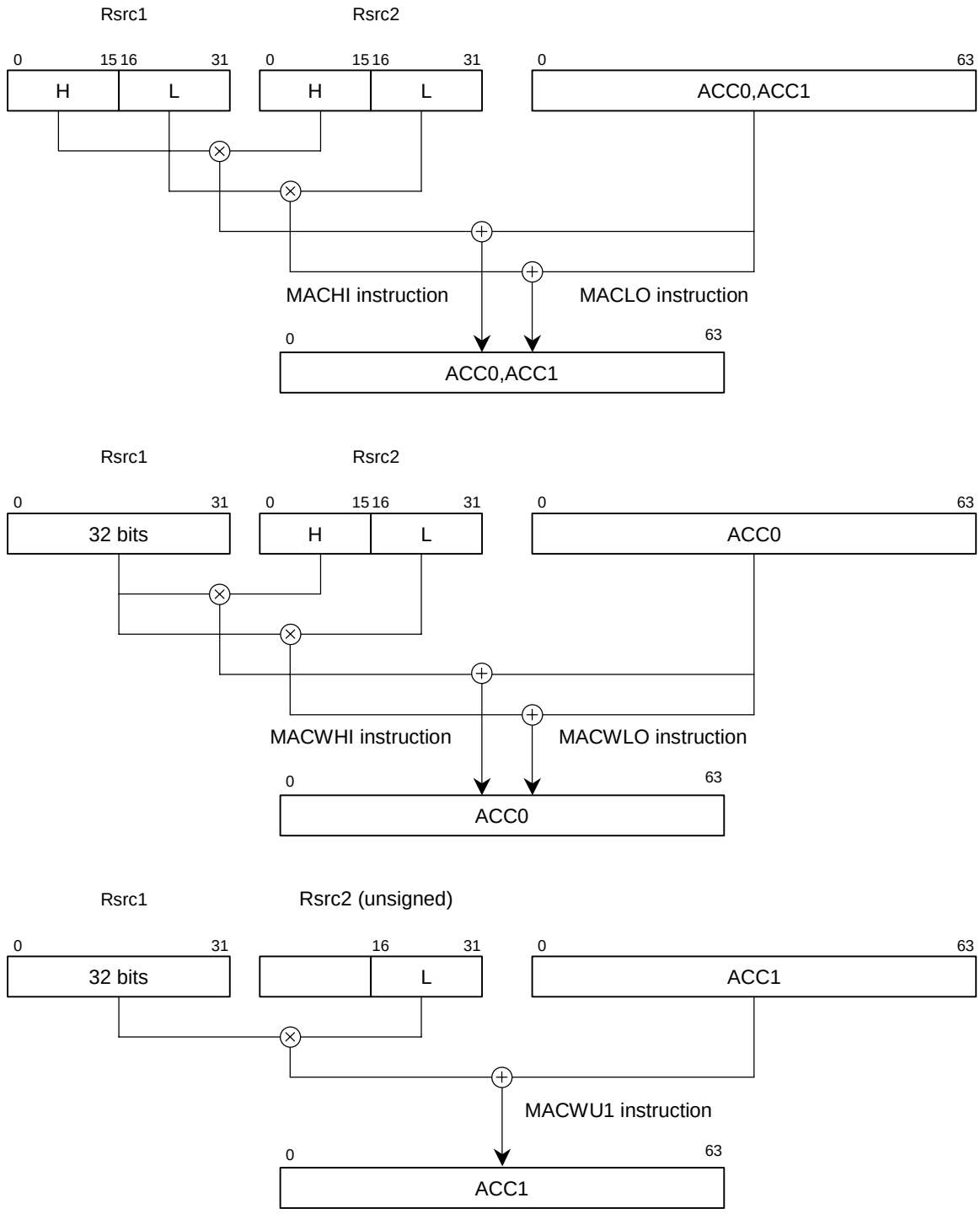
*	MACHI	Multiply-accumulate high-order halfwords
**	MACLH1	Multiply-accumulate low-order halfword and high-order halfword using accumulator1
*	MACLO	Multiply-accumulate low-order halfwords
	MACWHI	Multiply-accumulate word and high-order halfword
	MACWLO	Multiply-accumulate word and low-order halfword
**	MACWU1	Multiply-accumulate word and unsigned low-order halfword using accumulator1
**	MSBLO	Multiply low-order halfwords and subtract
*	MULHI	Multiply high-order halfwords
*	MULLO	Multiply low-order halfwords
	MULWHI	Multiply word and high-order halfword
	MULWLO	Multiply word and low-order halfword
**	MULWU1	Multiply word and unsigned low-order halfword using accumulator1
*	MVFACHI	Move high-order word from accumulator
*	MVFACLO	Move low-order word from accumulator
*	MVFACMI	Move middle-order word from accumulator
*	MVTACHI	Move high-order word to accumulator
*	MVTACLO	Move low-order word to accumulator
*	RAC	Round accumulator
*	RACH	Round accumulator halfword
**	SADD	Add accumulators
**	SATB	Saturate word into byte
**	SATH	Saturate word into halfword

Operation of these instructions are schematically shown in the next pages.



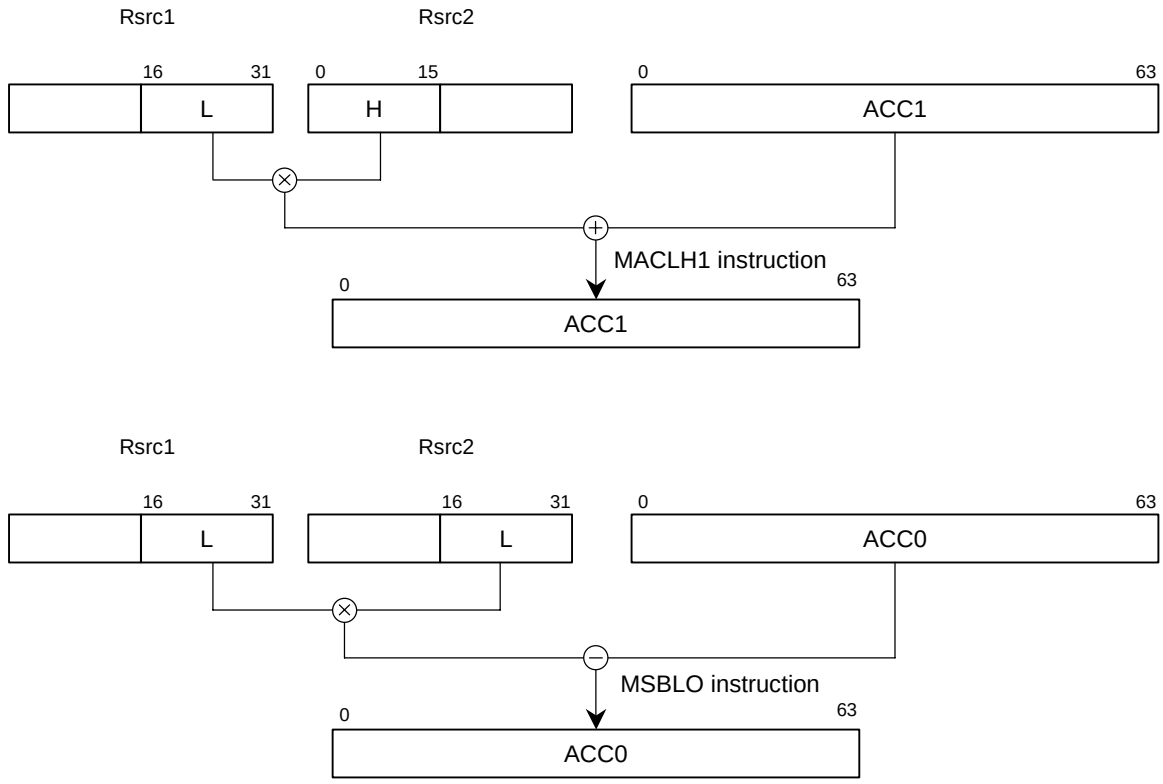
Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, “Instructions".

Figure2.2.2 Operation of the DSP Function Instructions 1 (Multiplication)



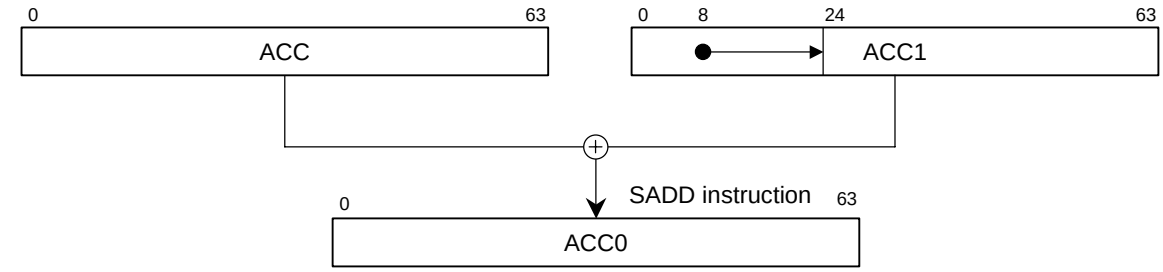
Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, “Instructions”.

Figure2.2.3 Operation of the DSP Function Instructions 2 (Multiply-Accumulate Operation)



Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, “Instructions.”

Figure2.2.4 Operation of the DSP Function Instructions 3 (Multiply-Accumulate Operation)



Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, “Instructions.”

Figure2.2.5 Operation of the DSP Function Instructions 4 (Addition)

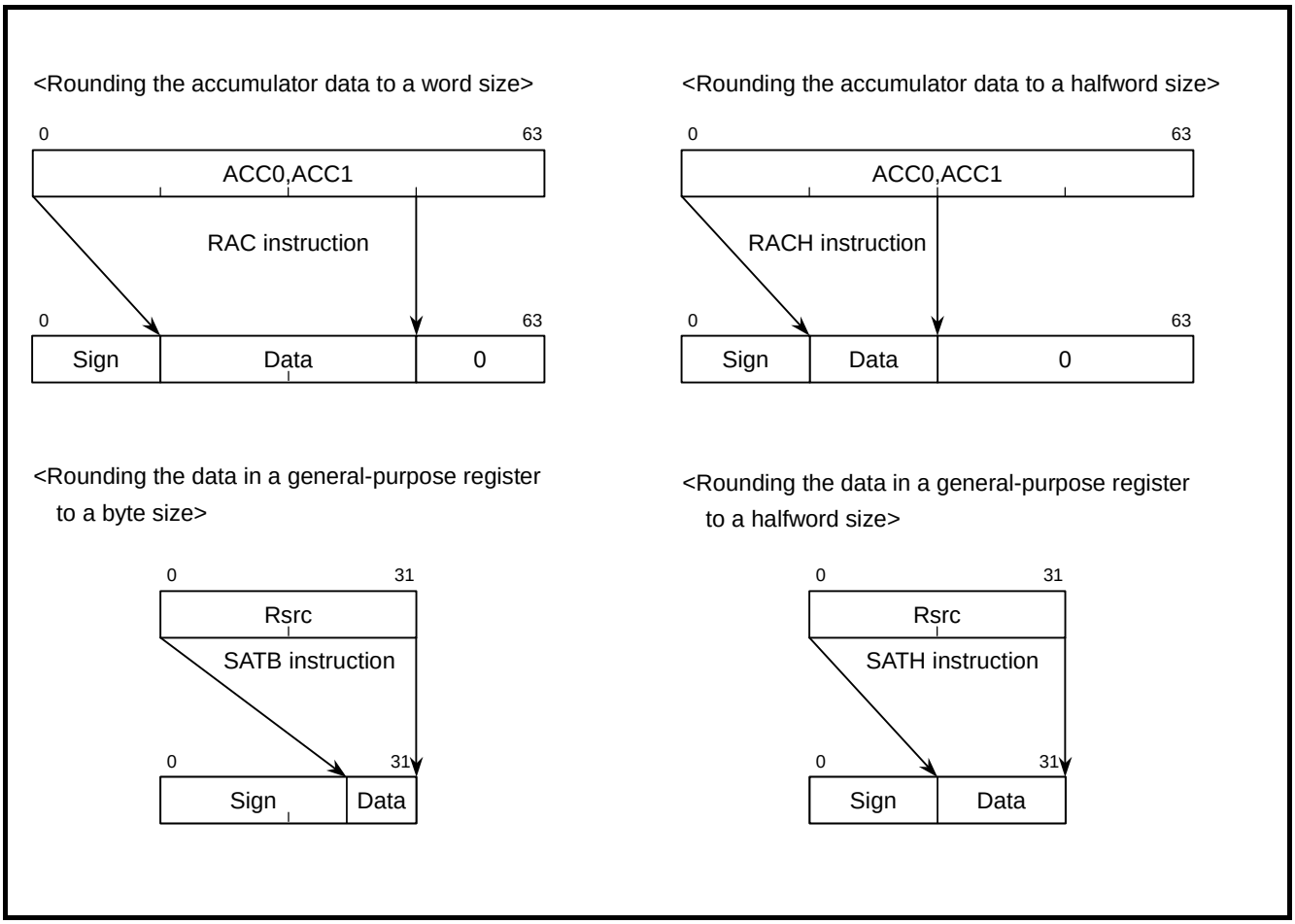


Figure2.2.6 Operation of the DSP Function Instructions 5 (Rounding)

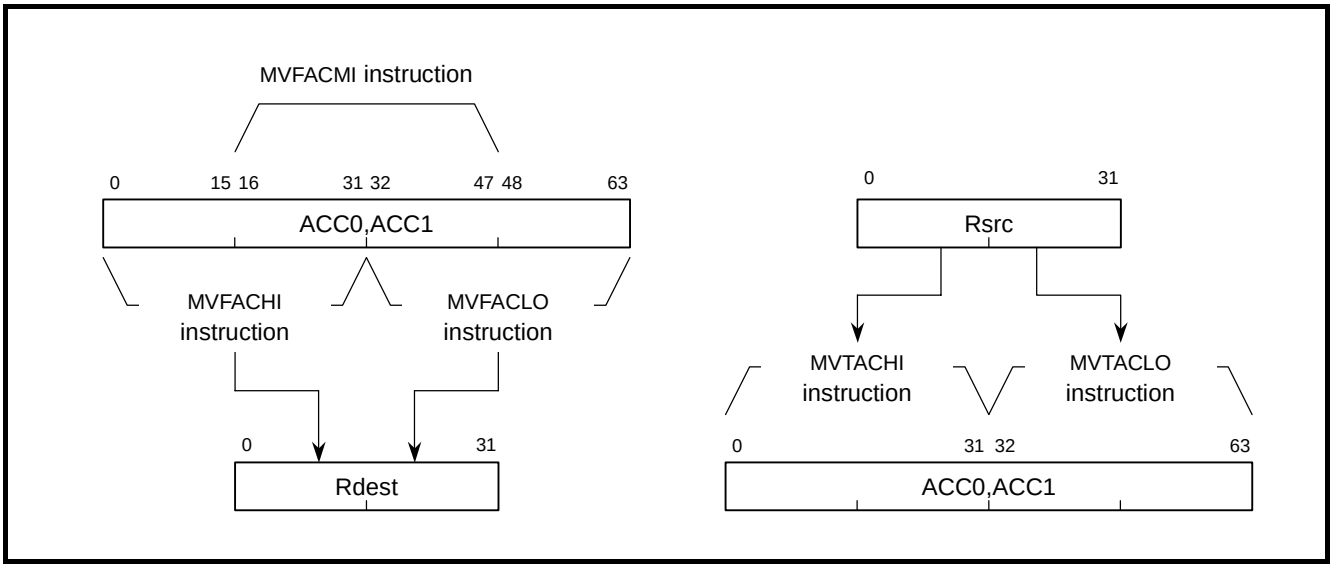


Figure2.2.7 Operation of the DSP Function Instructions 6 (Transfer between Accumulator and Register)

2.2.8 Coprocessor Support Instructions (3 instructions)

These instructions are used for interfacing with a coprocessor, as shown below.

*	MVTCP	Move to Coprocessor register
*	MVFCP	Move from Coprocessor register
*	OPECP	Operate Coprocessor

2.3 List of OPSP Extended Instruction Set

The instruction set of the OPSP-CPU has 27 new instructions that have been added as extensions from the M32R family instruction set and 21 conventional instructions which have had their functionality extended.

2.3.1 New Extended Instructions of the OPSP-CPU

Table2.3.1 List of new extended instructions

Classification	Mnemonic	Functional outline
Comparison instructions	CMPEQ	Compare between registers
	CMPZ	Compare register and immediate 0
	PCMPBZ	Compare register and immediate 0 bytewise
Multiplication/ division instructions	DIVB	Divide 8-bit signed integer
	DIVH	Divide 16-bit signed integer
	DIVUB	Divide 8-bit unsigned integer
	DIVUH	Divide 16-bit unsigned integer
	REMB	Remainder of 8-bit signed integer
	REMH	Remainder of 16-bit signed integer
	REMUB	Remainder of 8-bit unsigned integer
	REMUH	Remainder of 16-bit unsigned integer
Branch instructions	BCL	Branch if condition bit (C) = 1 and store return address in R14
	BNCL	Branch if condition bit (C) = 0 and store return address in R14
	JC	Branch if condition bit (C) = 1
	JNC	Branch if condition bit (C) = 0
	SC	Skip parallel execution pair if condition bit (C) = 1
	SNC	Skip parallel execution pair if condition bit (C) = 0
DSP function instructions	MACLH1	Multiply-accumulate operation (register × register + accumulator A1 → accumulator A1)
	MACWU1	Multiply-accumulate operation (register × register + accumulator A1 → accumulator A1)
	MSBLO	Multiply-accumulate operation (accumulator A0 – register × register → accumulator A0)
	MULWU1	Multiplication (register × register → accumulator A1)
	SADD	Addition (accumulator A0 + accumulator A1 → accumulator A0)
	SATB	Round register data to byte size
	SATH	Round register data to halfword size
Coprocessor support instructions	MVTCP	Move to coprocessor register
	MVFCP	Move from coprocessor register
	OPECP	Coprocessor operation

Note: In the table, the accumulators ACC0 and ACC1 are mnemonically expressed as A0 and A1, respectively.

2.3.2 Function-Extended Instructions of the OPSP-CPU

Table2.3.2 List of function-extended instructions

Classification	Mnemonic	Function-extended content
DSP function instructions	MACHI	Accumulator A0 or A1 can be specified in the operand description.
	MACLO	
	MULHI	
	MULLO	
	MVFACHI	
	MVFACLO	
	MVFACMI	
	MVTACHI	
	MVTACLO	
	RAC	Accumulator A0 or A1 can be specified in the operand description. In addition, the result deriving after left-shifting the accumulator bit specified by an immediate (imm1) is rounded.
	RACH	
Arithmetic/logical instructions	SLL	The parallel-executed instruction category has been changed from the left-side instruction (O–) to the both-side instruction (OS). (For details about the instruction category, refer to Section 2.5.3, “16-Bit Instruction List by Category.”
	SLLI	
	SRA	
	SRAI	
	SRL	
	SRLI	
Load/store instructions	STB	Register update has been added to addressing modes.
	STH	
EIT related instructions	TRAP	The run-time BPC value has been changed from BPC = PC + 4 to BPC = PC of the next instruction.
	RTE	Return to the halfword boundary is possible.

Note: In the table, the accumulators ACC0 and ACC1 are mnemonically expressed as A0 and A1, respectively.

2.4 Instruction Formats

The OPSP-CPU has two instruction formats: a 16-bit instruction, two of which are stored in pairs within the 32-bit word boundary, and a 32-bit instruction. (See Figure2.4.1.)

The basic instruction formats of the OPSP-CPU are shown in Figure2.4.2.

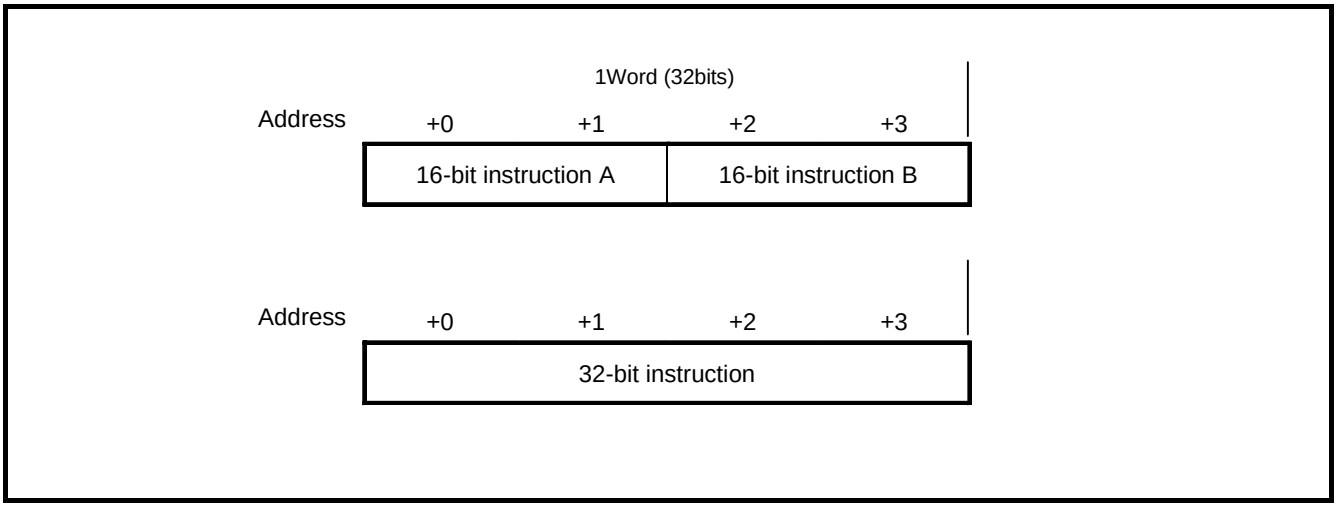


Figure2.4.1 16-Bit and 32-Bit Instructions

■ 16-bit instruction

<Instruction format>

				<Operation of the instruction>	<Example instruction>	
op1	R_1	op2	R_2	R1 = R1 op R2	AND	Rdest , Rsrc
op1	R_1	c		R1 = R1 op c	ADD	Rdest , #imm8
op1	cond	c		Branch (Short Displacement)	BC	pcdisp8

■ 32-bit instruction

<Instruction format>

					<Operation of the instruction>	<Example instruction>	
op1	R_1	op2	R_2	c	R1 = R2 op c	SRL3	Rdest , Rsrc , #imm16
op1	R_1	op2	R_2	c	Compare and Branch	BEQ	Rsrc1 , Rsrc2 ,
op1	R_1	c			R1 = R1 op c	LD24	Rdest , #imm24
op1	cond	c			Branch	BC	pcdisp24

Figure2.4.2 Basic Instruction Formats

2.5 Parallel Instruction Execution

2.5.1 Instruction Formats

The OPSP-CPU instruction set architecture supports parallel instruction execution for two 16-bit instructions that are stored in pairs within the word boundary. Whether two instructions are executed in parallel is determined by the value of the most significant bit (MSB) of each 16-bit instruction. (The MSB of each instruction only determines the method of instruction execution and does not affect the functionality of the instruction.)

The MSB of any 16-bit instruction that exists in the upper halfword location is always 0. If the MSB of the instruction that follows is also 0, then the two instructions are executed sequentially; if the MSB = 1, the two instructions are executed in parallel.

If the MSB of instruction B in Figure2.5.2 is 0, then instruction A and instruction B are executed sequentially. If the MSB of instruction B is 1, then instruction A and instruction B are executed in parallel. If instruction B needs to be executed in parallel, it is automatically altered to an instruction whose MSB is set to 1 by the assembler. For the same reason, NOP instructions used to adjust the word alignment have always their MSB set to 1 by the assembler.

The MSB of all 32-bit instructions is always 1, so that they are not executed in parallel.

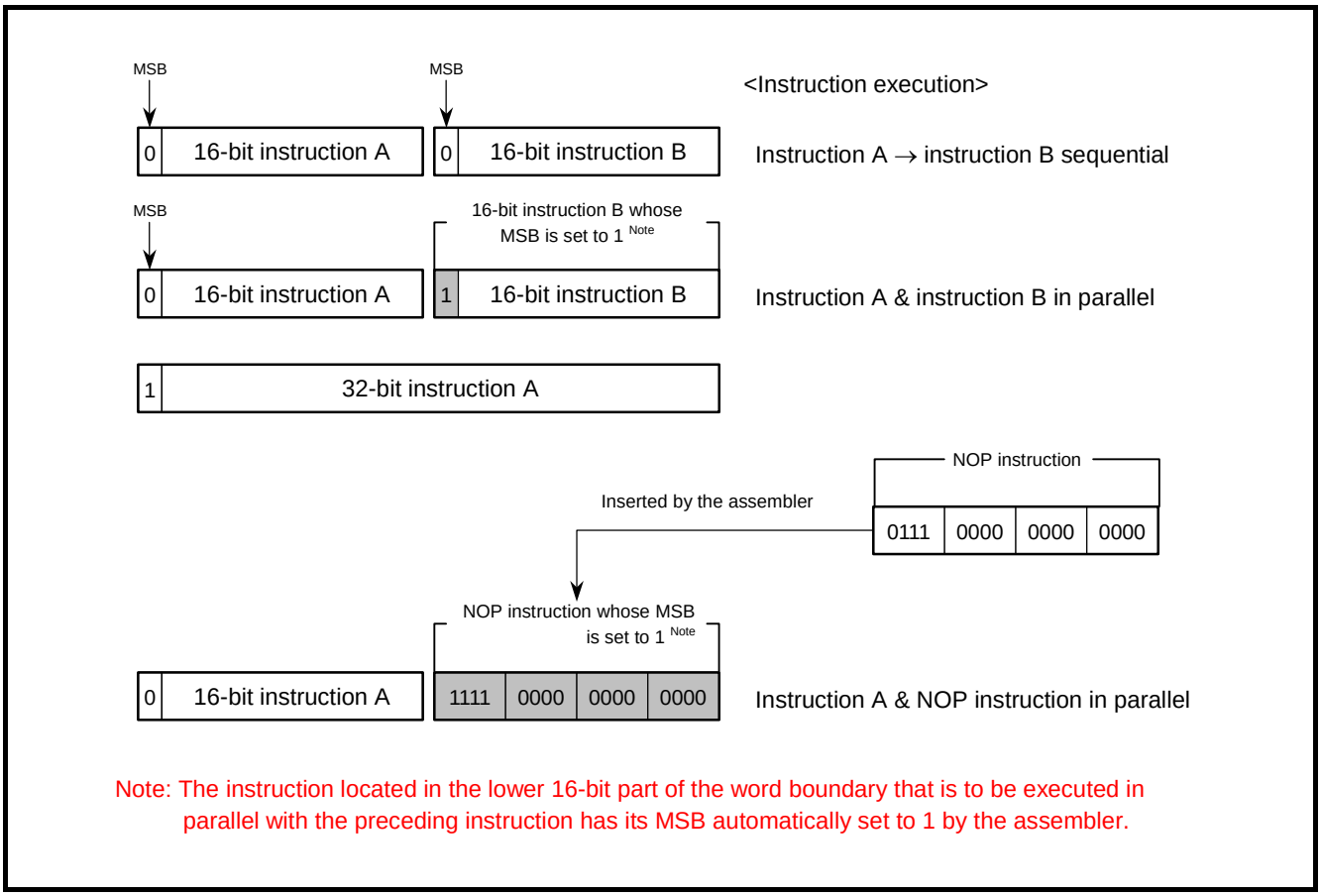


Figure 2.5.1 Instruction Processing

2.5.2 Parallel Instruction Execution in the OPSP

The OPSP-CPU has two pipelines: O pipe and S pipe. Two 16-bit instructions are executed in parallel using these two pipelines.

The 16-bit instructions executed in the S pipe include DSP function instructions and multiplication, arithmetic operation, logical operation, shift, comparison, transfer and NOP instructions. In the O pipe, on the other hand, all 16-bit instructions except DSP function and multiplication instructions can be executed.

Note that 32-bit instructions are not executed in parallel, and that all of them are executed in the O pipe.

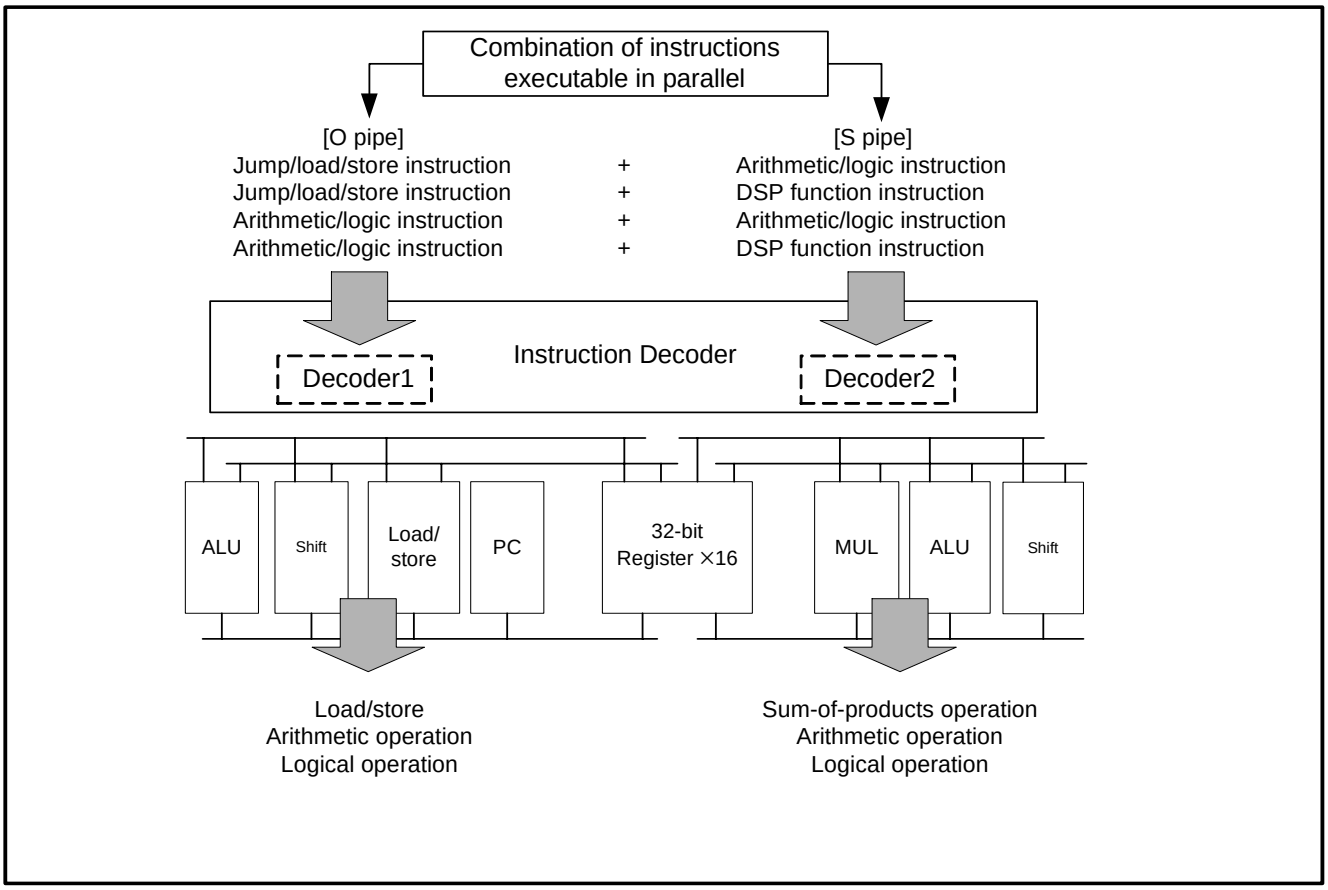


Figure2.5.2 Parallel Instruction Execution Mechanism of the OPSP-CPU

2.5.3 16-Bit Instruction List by Category

The 16-bit instructions that can be executed in parallel are classified into three categories by the executable pipeline.

- Instructions that can be executed in only the O pipe (left-side instruction: O–)
- Instructions that can be executed in only the S pipe (right-side instruction: –S)
- Instructions that can be executed in both O and S pipes (both-side instruction: OS)

The 16-bit instructions classified by category are listed in Table2.5.1.

Table2.5.1 16-Bit Instruction List by Category

O– (left-side instructions)	OS (both-side instructions)	–S (right-side instructions)
BC	ADD	MACHI
BCL	ADDI	MACLH1
BL	ADDV	MACLO
BNC	ADDX	MACWHI
BNCL	AND	MACQLO
BRA	CMP	MACWU1
BTST	CMPEQ	MSBLO
CLRPSW	CMPU	MUL
JC	CMPZ	MULHI
JL	LDI	MULLO
JMP	MV	MULWHI
JNC	NEG	NULWLO
LD	NOP	NULWU1
LDB	NOT	MVFACHI
LDH	OR	MVFACLO
LDUB	PCMPBZ	MVFACMI
LDUH	SLL	MVTACHI
LOCK	SLLI	MVTACLO
MVFC	SRA	RAC
MVTC	SRAI	RACH
RTE	SRL	SADD
SETPSW	SRLI	
SC	SUB	
SNC	SUBV	
ST	SUBX	
STB	XOR	
STH		
TRAP		
UNLOCK		

2.5.4 Positions of Parallel Executed Instructions

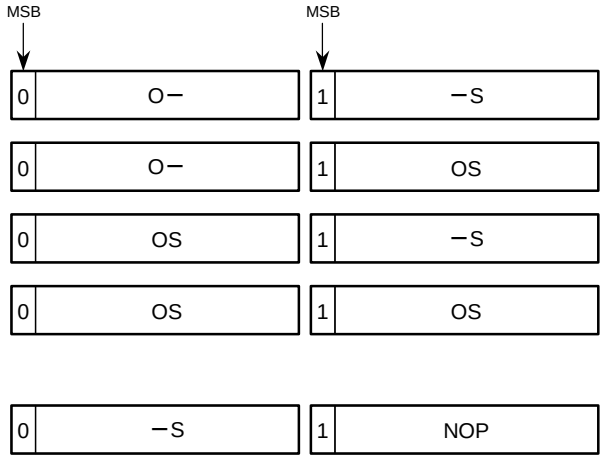
The 16-bit instruction pairs that can be executed in parallel are limited to the following four combinations of instruction categories.

- Left-side instruction and right-side instruction (O– and –S)
- Left-side instruction and both-side instruction (O– and OS)
- Both-side instruction and right-side instruction (OS and –S)
- Both-side instruction and both-side instruction (OS and OS)

The locations of instruction categories when 16-bit instruction pairs are executed in parallel are shown below.

- Left-side instruction (O–) located in the upper 16-bit part and the right-side instruction (–S) located in the lower 16-bit part
- Left-side instruction (O–) located in the upper 16-bit part and the both-side instruction (OS) located in the lower 16-bit part
- Both-side instruction (OS) located in the upper 16-bit part and the right-side instruction (–S) located in the lower 16-bit part
- Both-side instruction (OS) located in the upper 16-bit part and the other both-side instruction located in the lower 16-bit part

However, if a NOP instruction is located in the lower 16-bit part for the purpose of word alignment, a right-side instruction (–S) may be located in the upper 16-bit part as an instruction pair to be executed in parallel.



Note: The instruction located in the lower 16-bit part of the word boundary that is to be executed in parallel with the preceding instruction has its MSB automatically set to 1 by the assembler.

Figure 2.5.3 Locations of Parallel Executable Instruction Categories

2.5.5 Operand Interferences

Two parallel executed 16-bit instructions are executed independently of each other, and not sequenced in time. When executed in parallel, the two instructions are handled as having no mutual dependency with regard to the operand, so that they are not subject to interlock processing. Please be aware of this point when writing a program.

The value of the source operand referenced by a parallel executed instruction pair is one that was stored in the operand immediately before the CPU started executing the instructions in parallel. For example, if in a parallel executed instruction pair, one instruction writes to a register and the other instruction references it, the register value that is referenced is one that was stored in the register immediately before the CPU started executing the instruction pair in parallel. Furthermore, after the instruction pair was executed, the result is written to the register.

Note, however, that if two instructions are executed in parallel that write to the same register (collision of writes to a register), program operation cannot be guaranteed.

(1) Examples of operand interferences in general-purpose registers

The following shows typical examples of operand interferences in general-purpose registers attributable to two transfer instructions (MV instructions).

```
Example 1: MV    R1,R0   ||  MV    R2,R1

Example 2: MV    R1,R0   ||  MV    R1,R2
```

Note: the symbol || denotes that two instructions are executed in parallel.

In example 1, one of the two instructions in pairs writes to a register (R1) and the other references it. In this case, R1 is assigned the value of R0. Similarly, R2 is assigned the value of R1 before assignment to R1 (i.e., the value of R1 before the instruction “MV R1,R0” is executed).

Example 2 is an example where two instructions in the instruction pair write to the same register (collision of writes to a register). In this case, the registers accessed for write by two MV instructions both are R1, so that the value of R1 after instruction execution is indeterminate.

(2) Examples of operand interferences in control registers

In addition to general-purpose registers, operand interferences will occur in control registers such as the PSW and CBR that include the condition bit (C).

```
Example 3: When two instructions are executed successively
        CMP    R1,R0
        BC     _label

Example 4: When two instructions are executed in parallel
        CMP    R1,R0   ||  BC    _label
```

Note: the symbol || denotes that two instructions are executed in parallel.

In example 3, the comparison instruction (CMP) is executed before the conditional branch instruction (BC) is executed. In this case, the condition bit (C) is updated as a result of the CMP instruction executed, and the BC instruction references this updated condition bit (C) to determine whether or not to branch.

In example 4, the CMP and the BC instructions are executed in parallel. The BC instruction references the condition bit (C) before the CMP instruction is executed, to determine whether or not to branch. Be aware that the condition bit (C) is referenced before it is operated on by execution of the CMP instruction. The result of the CMP instruction executed is reflected in the condition bit (C) after parallel instruction execution.

Furthermore, if two instructions are executed in parallel that will change the condition bit (C), the value of the condition bit (C) after instruction execution becomes indeterminate as in the case of a collision of writes to a register that occurs in general-purpose registers. Shown below are examples where the condition bit (C) becomes indeterminate after an instruction pair is executed in parallel.

```
Example 5:CMP    R1,R2    ||    ADDX   R3,R4

Example 6: MVTC  R1,PSW   ||    ADDX   R1,R2

Example 7: TRAP  #1       ||    CMP    R3,R4

Example 8: RTE            ||    ADDX   R3,R4
```

Note: the symbol || denotes that two instructions are executed in parallel.

This page is blank for reasons of layout.

CHAPTER 3
INSTRUCTIONS

3.1 Guide to Detailed Instruction Description

The following outlines each item that is described in the detailed description of instructions in the pages to follow.

[Mnemonic]

The mnemonics of the OPSP-CPU consist of an instruction and the operand description that follows. The operand is the target to be operated on by the instruction.

Table 3.1.1 List of Operand description

Operand description[Note]	Addressing mode	Target to be operated on by instruction
R	Register direct	General-purpose register of the OPSP-CPU (R0–R15)
CR	Control register	Control register of the OPSP-CPU (CR0=PSW, CR1=CBR, CR2=SPI, CR3=SPU, CR5=EVB, CR6=BPC)
CPR	Coprocessor register	Register of the coprocessor connected to the OPSP-CPU
A	Accumulator	Content of the OPSP-CPU accumulator (A0, A1)
@Rn	Register indirect	Memory content whose address is indicated by a register value
@(disp, Rn)	Register relative indirect	Memory content whose address is indicated by (register value) + (16-bit constant that is sign-extended to 32 bits)
@Rn+	Register indirect + register update	Register value incremented by 4, 2, or 1 (Memory content whose address is indicated by a preupdate register value)
@+Rn	Register indirect + register update	Register value incremented by 4 (Memory content whose address is indicated by an updated register value)
@−Rn	Register indirect + register update	Register value decremented by 4 (Memory content whose address is indicated by an updated register value)
#imm	Immediate	Immediate value (For details on how the value is handled, refer to the detailed description of each instruction.)
pcdisp	PC relative	Memory content whose address is indicated by (PC value) + (8, 16, or 24-bit displacement which is sign-extended to 32 bits and then shifted left 2 bits).

Note: In operand descriptions “Rsrc” and “Rdest,” src and dest each represent a general-purpose register number (0–15). In operand descriptions “CRsrc” and “CRdest,” src and dest each represent a control register number (0–3, 5, or 6). In operand descriptions “Asrc” and “Adest,” src and dest each represent an accumulator number (0 or 1).

[Function]

Operation of each instruction is described by first outlining what the instruction does and then showing a C language based description of the operation. The description of instruction operation is outlined below.

Table 3.1.2 Description of Operations (Operators)

Operator	Operation performed
+	Addition (binary operator)
—	Subtraction (binary operator)
*	Multiplication (binary operator)
/	Division (binary operator)
%	Remainder calculation (binary operator)
++	Increment (unary operator)
- -	Decrement (unary operator)
-	Sign inversion (unary operator)
=	Assign right side to left side (assignment operator)
+=	Add left and right side variables and assign the result to left side (assignment operator)
-=	Subtract right side variable from left side variable and assign the result to left side (assignment operator)
>	Greater than (relational operator)
<	Smaller than (relational operator)
>=	Greater than or equal (relational operator)
<=	Smaller than or equal (relational operator)
==	Equal (relational operator)
!=	Not equal (relational operator)
&&	AND (logical operator)
||	OR (logical operator)
|	NOT (logical operator)
?:	Create conditional expression (conditional operator)

Table 3.1.3 Description of Operations (Bitwise Operators)

Operator	Operation performed
<<	Shift the bit left
>>	Shift the bit right
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive-OR (EXOR)
~	Bit inversion

Table 3.1.4 Data Types

Representation	Type	Signed or unsigned	Bit length	Range of values
char	Integer	Signed	8	-128 to +127
short	Integer	Signed	16	-32,768 to +32,767
int	Integer	Signed	32	-2,147,483,648 to +2,147,483,647
unsigned char	Integer	Unsigned	8	0 to 255
unsigned short	Integer	Unsigned	16	0 to 655,535
unsigned int	Integer	Unsigned	32	0 to 4,294,967,295
signed64bit	Integer	Signed	64	Signed 64-bit integer (when operating on accumulators)

[Description]

The function of each instruction is detailed here. Furthermore, changes of the condition bit (C) in the PSW register that occur as a result of execution of the instruction are described.

[EIT occurrence]

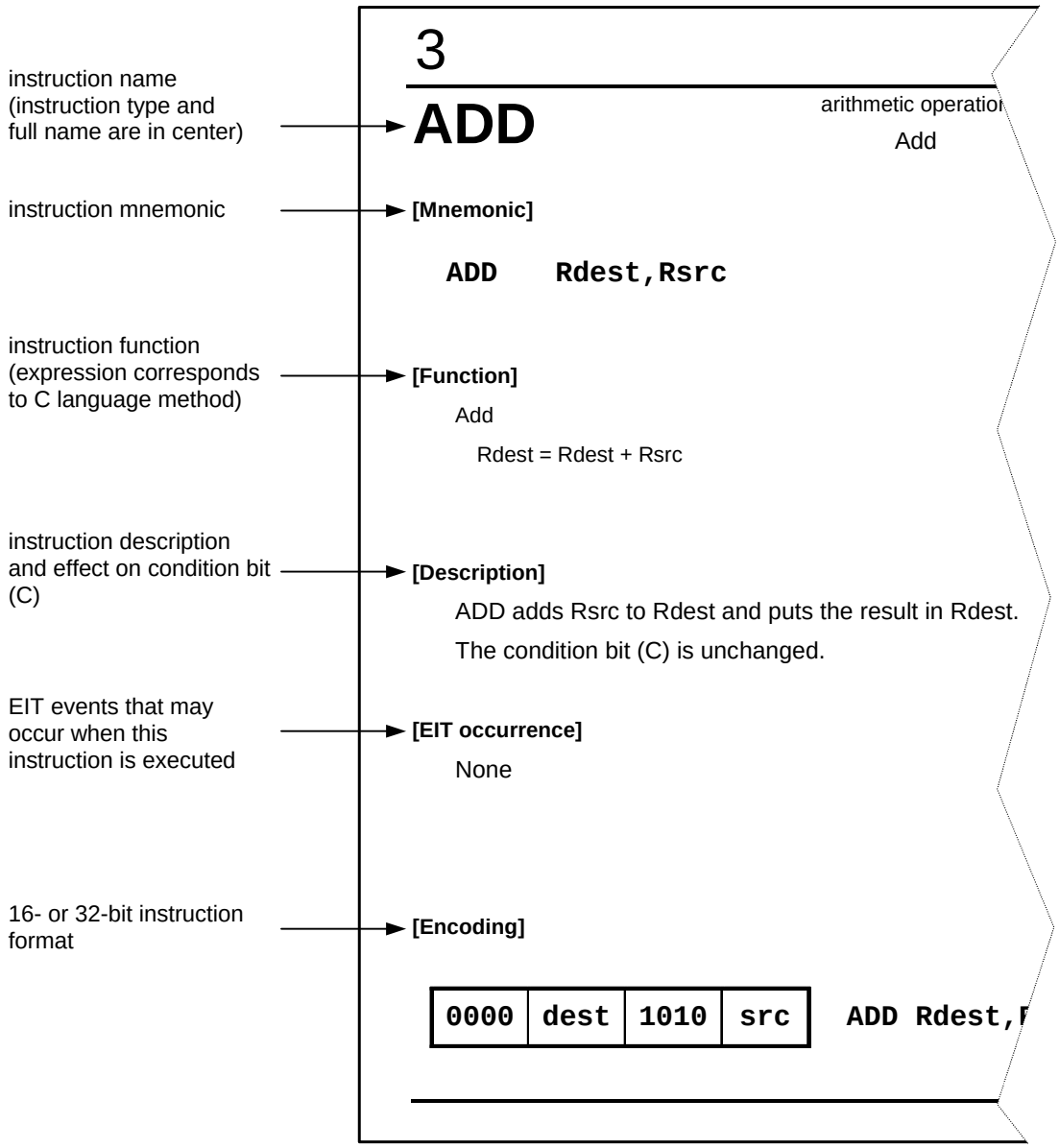
A generated EIT means an EIT event (exception, interrupt, or trap) that may occur as a result of execution of the instruction. The EIT events that are likely to occur as a result of instruction execution include an address exception, trap and a privileged instruction exception.

[Encoding]

A 16-bit or 32-bit instruction bit pattern is shown. In the instruction format, src and dest each represent the corresponding register number, while imm and disp represent immediate and displacement values, respectively. (The magnitude of the numeric value that is assigned to each bit field is determined by the field width.) For details about the instruction format, refer to Section 2.3, “Instruction Formats,” in Chapter 2.

3.2 Detailed Description of Instructions

Each instruction of the OPSP-CPU is described in detail beginning with the next page. Instructions are listed in alphabetical order. Note that each page consists of the items described below.



ADD

arithmetic operation instruction
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) dose not changed.

[EIT occurrence]

None

[Encoding]

0000	dest	1010	src

ADD Rdest,Rsrc

ADD3

arithmetic operation instruction
Add 3-operand

ADD3

[Mnemonic]

ADD3 Rdest,Rsrc,#imm16

[Function]

Add

Rdest = Rsrc + (signed short) imm16;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1000	dest	1010	src

imm16

ADD3 Rdest,Rsrc,#imm16

ADDI

arithmetic operation instruction
Add immediate

ADDI

[Mnemonic]

ADDI Rdest,#imm8

[Function]

Add

Rdest = Rdest + (signed char) imm8;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0100	dest	imm8	ADDI Rdest,#imm8

ADDV

arithmetic operation instruction
Add with overflow checking

ADDV

[Mnemonic]

ADDV Rdest,Rsrc

[Function]

Add

Rdest = (signed) Rdest + (signed) Rsrc;

C = overflow ? 1 : 0

[Description]

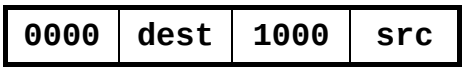
ADDV adds Rsrc to Rdest and puts the result in Rdest.

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

[EIT occurrence]

None

[Encoding]



ADDV Rdest,Rsrc

ADDV3

arithmetic operation instruction
Add 3-operand with overflow checking

ADDV3

[Mnemonic]

ADDV3 Rdest,Rsrc,#imm16

[Function]

Add

Rdest = (signed) Rsrc+ (signed)((signed short)imm16);

C = overflow ? 1 : 0

[Description]

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

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

[EIT occurrence]

None

[Encoding]

1000	dest	1000	src

imm16

ADDV3 Rdest,Rsrc,#imm16

ADDX

arithmetic operation instruction
Add with carry

ADDX

[Mnemonic]

ADDX Rdest,Rsrc

[Function]

Add

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

C = carry_out ? 1 : 0;

[Description]

ADDX adds Rsrc and C to Rdest, and puts the result in Rdest. The condition bit (C) is set when the addition result cannot be represented by a 32-bit unsigned integer; otherwise it is cleared.

[EIT occurrence]

None

[Encoding]

0000	dest	1001	src

ADDX Rdest,Rsrc

AND

logic operation instruction

AND

AND

[Mnemonic]

```
AND Rdest,Rsrc
```

[Function]

Logical AND

Rdest = Rdest & Rsrc;

[Description]

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

The condition bit (C) dose not changed.

[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) dose not changed.

[EIT occurrence]

None

[Encoding]

1000	dest	1100	src

imm16

AND3 Rdest,Rsrc,#imm16

BC

branch instruction
Branch on C-bit

BC

[Mnemonic]

(1) **BC pcdisp8**
(2) **BC pcdisp24**

[Function]

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

[Description]

BC causes a branch to the specified label when the condition bit (C) is 1.
There are two instruction formats; which allows software, such as an assembler, to decide on the better format.
The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0111	1100	pcdisp8

BC pcdisp8

1111	1100	

pcdisp24

BC pcdisp24

BCL

branch instruction
Branch and link on C-bit

BCL

[Mnemonic]

(1) **BCL pcdisp8**
(2) **BCL pcdisp24**

[Function]

Branch

```
(1) if ( C = = 1 ) {
      R14 = ( PC & 0xfffffffc ) + 4 ;
      PC = ( PC & 0xfffffffc ) + ( ( ( signed char ) pcdisp8 ) << 2 ) ;
    }
(2) if ( C = = 1 ) {
      R14 = ( PC & 0xfffffffc ) + 4 ;
      PC = ( PC & 0xfffffffc ) + ( sign_extend ( pcdisp24 ) << 2 ) ;
    }

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

[Description]

When the condition bit (C) = 1, BCL causes a branch to the specified label and store 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) does not change.

[EIT occurrence]

None

[Encoding]

0111	1000	pcdisp8

BCL pcdisp8

1111	1000	

pcdisp24

BCL pcdisp24

BCLR

bit operation instruction
Bit clear

BCLR

[Mnemonic]

BCLR #bitpos,@(disp16,Rsrc)

[Function]

Bit operation for memory contents. Set a specified bit to 0.

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

[Description]

BCLR reads byte data in memory from the address specified by Rsrc and a 16-bit displacement and stores the read value after changing its bit specified by bitpos to 0.

The displacement is sign-extended before address calculation. bitpos is specified for bits 0–7 where MSB = 0 and LSB = 7. Memory is accessed in bytes.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

1010	0	bitpos	0111	src

disp16

BCLR #bitpos,@(disp16,Rsrc)

BEQ

branch instruction
Branch on equal

BEQ

[Mnemonic]

BEQ Rsrc1,Rsrc2,pcdisp16

[Function]

Branch

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

[Description]

BEQ causes a branch to the specified label when Rsrc1 is equal to Rsrc2.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

BEQ Rsrc1,Rsrc2,pcdisp16

BEQZ

branch instruction
Branch on equal to zero

BEQZ

[Mnemonic]

BEQZ Rsrc,pcdisp16

[Function]

Branch

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

[Description]

BEQZ causes a branch to the specified label when Rsrc is equal to zero.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

BEQZ Rsrc,pcdisp16

BGEZ

branch instruction

Branch on greater than or equal to zero

BGEZ

[Mnemonic]

BGEZ Rsrc,pcdisp16

[Function]

Branch

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

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1011	0000	1011	src	pcdisp16

BGEZ Rsrc,pcdisp16

BGTZ

branch instruction
Branch on greater than zero

BGTZ

[Mnemonic]

BGTZ Rsrc,pcdisp16

[Function]

Branch

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

[Description]

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

The condition bit (C) dose not changed.

[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]

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

[Description]

BL causes an unconditional branch to the address specified by the label and puts the return address in R14. There are two instruction formats; this allows software, such as an assembler, to decide on the better format.
The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0111	1110	pcdisp8

BL pcdisp8

1111	1110	

pcdisp24

BL pcdisp24

BLEZ

branch instruction
Branch on less than or equal to zero

BLEZ

[Mnemonic]

BLEZ Rsrc,pcdisp16

[Function]

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

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1011	0000	1100	src

pcdisp16

BLEZ Rsrc,pcdisp16

BLTZ

branch instruction
Branch on less than zero

BLTZ

[Mnemonic]

BLTZ Rsrc,pcdisp16

[Function]

Branch

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

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1011	0000	1010	src

pcdisp16

BLTZ Rsrc,pcdisp16

BNC

branch instruction
Branch on not C-bit

BNC

[Mnemonic]

(1) **BNC pcdisp8**
(2) **BNC pcdisp24**

[Function]

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

[Description]

BNC causes a branch 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) dose not changed.

[EIT occurrence]

None

[Encoding]

0111	1101	pcdisp8

BNC pcdisp8

1111	1101	

pcdisp24

BNC pcdisp24

BNCL

branch instruction
Branch and link on not C-bit

BNCL

[Mnemonic]

(1) **BNCL pcdisp8**
(2) **BNCL pcdisp24**

[Function]

Branch

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

[Description]

When the condition bit (C) = 0, BNCL causes a branch to the specified label and stores 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) does not change.

[EIT occurrence]

None

[Encoding]

0111	1001	pcdisp8

BNCL pcdisp8

1111	1001	

pcdisp24

BNCL pcdisp24

BNE

branch instruction
Branch on not equal to

BNE

[Mnemonic]

BNE Rsrc1,Rsrc2,pcdisp16

[Function]

Branch

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

[Description]

BNE causes a branch to the specified label when Rsrc1 is not equal to Rsrc2.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1011	src1	0001	src2	pcdisp16

BNE Rsrc1,Rsrc2,pcdisp16

BNEZ

branch instruction
Branch on not equal to zero

BNEZ

[Mnemonic]

BNEZ Rsrc,pcdisp16

[Function]

Branch

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

[Description]

BNEZ causes a branch to the specified label when Rsrc is not equal to zero.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1011	0000	1001	src

pcdisp16

BNEZ Rsrc,pcdisp16

BRA

branch instruction
Branch

BRA

[Mnemonic]

(1) **BRA pcdisp8**
(2) **BRA pcdisp24**

[Function]

Branch

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

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

where

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

[Description]

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

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

[EIT occurrence]

None

[Encoding]

0111	1111	pcdisp8

BRA pcdisp8

1111	1111	

pcdisp24

BRA pcdisp24

BSET

bit operation instruction
Bit set

BSET

[Mnemonic]

BSET #bitpos, @(disp16,Rsrc)

[Function]

Bit operation on memory content. Set a specified bit to 1.

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

[Description]

BSET reads byte data in memory from the address specified by Rsrc and a 16-bit displacement and stores the read value after changing its bit specified by bitpos to 1.

The displacement is sign-extended before address calculation. bitpos is specified for bits 0–7 where MSB = 0 and LSB = 7. Memory is accessed in bytes.

[EIT occurrence]

None

[Encoding]

BSET #bitpos,@(disp16,Rsrc)

BTST

bit operation instruction
Bit test

BTST

[Mnemonic]

BTST #bitpos,Rsrc

[Function]

Bit operation to extract a specified register bit.

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

[Description]

BTST extracts a bit specified by bitpos from the 8 low-order bits of Rsrc and sets it in the condition bit (C). "bitpos" is specified for bits 0–7 where LSB = 7.

[EIT occurrence]

None

[Encoding]

0000	0	bitpos	1111	src

BTST #bitpos, Rsrc

CLRPSW

bit operation instruction
Clear PSW

CLRPSW

[Mnemonic]

CLRPSW #imm8

[Function]

Set SM, IE, PM, CE or C bit in the PSW to 0.

PSW &= ~(unsigned char) imm8 | 0x0000ff00

[Description]

Logically AND the inverse of the 8-bit value specified by imm8 with the 8 low-order bits in the PSW (bits 24–31) bitwise and write the result to the 8 low-order bits in the PSW bit by bit.

[EIT occurrence]

Privilege instruction exception(PIE)

[Encoding]

0111	0010	imm8

CLRPSW #imm8

CMP

compare instruction
Compare

CMP

[Mnemonic]

```
CMP Rsrc1,Rsrc2
```

[Function]

Compare

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

[Description]

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

[EIT occurrence]

None

[Encoding]

0000	src1	0100	src2	CMP Rsrc1,Rsrc2

CMPEQ

compare instruction
Compare equal to

CMPEQ

[Mnemonic]

```
CMPEQ Rsrc1,Rsrc2
```

[Function]

Compare

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

[Description]

When Rsrc1 and Rsrc2 are equal, the condition bit (C) is set to 1.

[EIT occurrence]

None

[Encoding]

0000	src1	0110	src2

CMPEQ Rsrc1,Rsrc2

CMPI

compare instruction
Compare immediate

CMPI

[Mnemonic]

CMPI Rsrc,#imm16

[Function]

Compare

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

[Description]

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

[EIT occurrence]

None

[Encoding]

CMPI Rsrc,#imm16

CMPU

compare instruction
Compare unsigned

CMPU

[Mnemonic]

```
CMPU Rsrc1,Rsrc2
```

[Function]

Compare

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

[Description]

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

[EIT occurrence]

None

[Encoding]

0000	src1	0101	src2

CMPU Rsrc1,Rsrc2

CMPUI

compare instruction
Compare unsigned immediate

CMPUI

[Mnemonic]

CMPUI Rsrc,#imm16

[Function]

Compare

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

[Description]

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

[EIT occurrence]

None

[Encoding]

1000	0000	0101	src

imm16

CMPUI Rsrc,#imm16

CMPZ

compare instruction
Compare equal to zero

CMPZ

[Mnemonic]

CMPZ Rsrc

[Function]

Compare

C = (Rsrc == 0) ? 1 : 0;

[Description]

The condition bit (C) is set when Rsrc is zero.

[EIT occurrence]

None

[Encoding]

0000	0000	0111	src

CMPZ Rsrc

DIV

multiply and divide instruction
Divide

DIV

[Mnemonic]

```
DIV Rdest,Rsrc
```

[Function]

Signed division

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

[Description]

DIV divides Rdest by Rsrc and puts the quotient in Rdest. The operands are treated as signed 32-bit values and the result is rounded toward zero.

The condition bit (C) dose not changed.

When Rsrc is zero, Rdest dose not changed.

[EIT occurrence]

None

[Encoding]

1001	dest	0000	src

0000	0000	0000	0000

```
DIV Rdest,Rsrc
```

DIVB

multiply and divide instruction
Divide byte

DIVB

[Mnemonic]

```
DIVB Rdest,Rsrc
```

[Function]

Signed division

Rdest =(signed char)Rdest / (signed)Rsrc ;

[Description]

DIVB divides Rdest by Rsrc and store the quotient in Rdest. Of the operands of this instruction, the dividend is handled as a signed 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as a signed 32-bit value, and the quotient is rounded toward zero.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0000	src

0000	0000	0001	1000

```
DIVB Rdest,Rsrc
```

DIVH

multiply and divide instruction
Divide Half-word

DIVH

[Mnemonic]

```
DIVH Rdest,Rsrc
```

[Function]

Signed division

Rdest = (signed short) Rdest / (signed) Rsrc ;

[Description]

DIVH divides Rdest by Rsrc and store the quotient in Rdest. Of the operands of this instruction, the dividend is handled as a signed 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as a signed 32-bit value, and the quotient is rounded toward zero.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0000	src

0000	0000	0001	0000

DIVH Rdest,Rsrc

DIVU

multiply and divide instruction
Divide unsigned

DIVU

[Mnemonic]

```
DIVU Rdest,Rsrc
```

[Function]

Unsigned division

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

[Description]

DIVU divides Rdest by Rsrc and puts the quotient in Rdest. The operands are treated as unsigned 32-bit values and the result is rounded toward zero.

The condition bit (C) dose not changed.

When Rsrc is zero, Rdest dose not changed.

[EIT occurrence]

None

[Encoding]

1001	dest	0001	src

0000	0000	0000	0000

```
DIVU Rdest,Rsrc
```

DIVUB

multiply and divide instruction
Divide unsigned byte

DIVUB

[Mnemonic]

DIVUB Rdest,Rsrc

[Function]

Unsigned division

Rdest =(unsigned char)Rdest / (unsigned)Rsrc ;

[Description]

DIVUB divides Rdest by Rsrc and stores the quotient in Rdest.

Of the operands of this instruction, the dividend is handled as an unsigned 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as an unsigned 32-bit value, and the quotient is rounded toward zero.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0001	src

0000	0000	0001	1000

DIVUB Rdest,Rsrc

DIVUH

multiply and divide instruction
Divide unsigned halfword

DIVUH

[Mnemonic]

DIVUH Rdest,Rsrc

[Function]

Unsigned division

Rdest =(unsigned short)Rdest / (unsigned)Rsrc ;

[Description]

DIVUH divides Rdest by Rsrc and stores the quotient in Rdest.

Of the operands of this instruction, the dividend is handled as an unsigned 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as an unsigned 32-bit value, and the quotient is rounded toward zero.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

DIVUH Rdest,Rsrc

JC

branch instruction
Jump on C-bit

JC

[Mnemonic]

```
JC Rsrc
```

[Function]

Jump

if (C ==1)PC =Rsrc & 0xfffffffc ;

[Description]

JC causes a jump to the address specified by Rsrc when the condition bit (C) = 1.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

0001	1100	1100	src

JC Rsrc

JL

branch instruction
Jump and link

JL

[Mnemonic]

```
JL Rsrc
```

[Function]

Subroutine call (register direct)

R14 = (PC & 0xfffffffc) + 4;

PC = Rsrc & 0xfffffffc;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	1110	1100	src

JL Rsrc

JMP

branch instruction

Jump

JMP

[Mnemonic]

```
JMP Rsrc
```

[Function]

Jump

PC = Rsrc & 0xfffffffc;

[Description]

JMP causes an unconditional jump to the address specified by Rsrc.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	1111	1100	src	JMP Rsrc

JNC

branch instruction
Jump on not C-bit

JNC

[Mnemonic]

```
JNC Rsrc
```

[Function]

Jump

if (C==0)PC =Rsrc & 0xfffffffc ;

[Description]

JNC causes a jump to the address specified by Rsrc when the condition bit (C) = 0.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

0001	1101	1100	src	JNC Rsrc

LD

load/store instruction

Load

LD

[Mnemonic]

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

[Function]

Load

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

[Description]

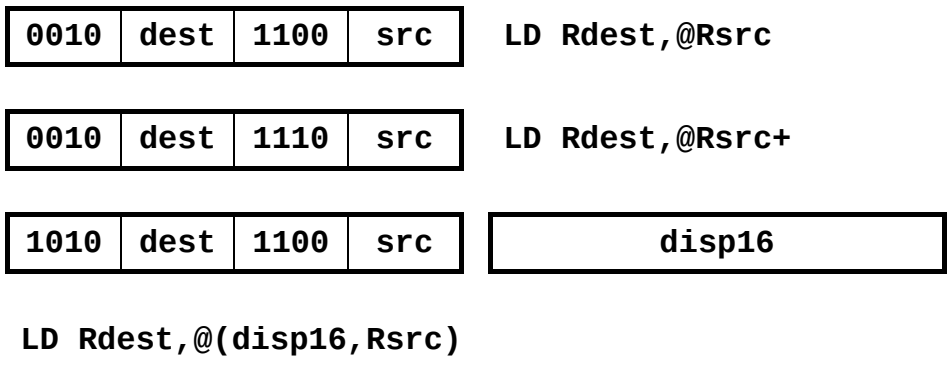
(1) The contents of the memory at the address specified by Rsrc are loaded into Rdest.
(2) The contents of the memory at the address specified by Rsrc are loaded into Rdest. Rsrc is post incremented by 4.
(3) The contents of the memory at the address specified by Rsrc combined with the 16-bit displacement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

The condition bit (C) dose not changed.

[EIT occurrence]

Address exception (AE)

[Encoding]



LD24

transfer 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) dose not changed.

[EIT occurrence]

None

[Encoding]


```
LD24 Rdest,#imm24
```

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.

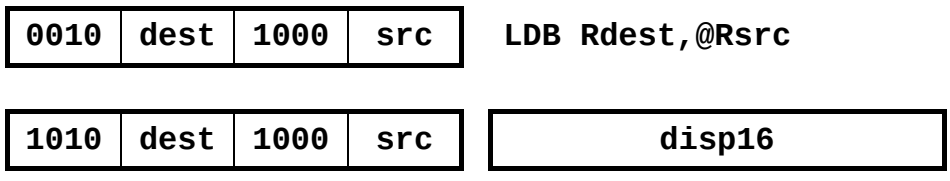
The displacement value is sign-extended to 32 bits before the address calculation.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]



LDH

load/store instruction
Load halfword

LDH

[Mnemonic]

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

[Function]

Load

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

[Description]

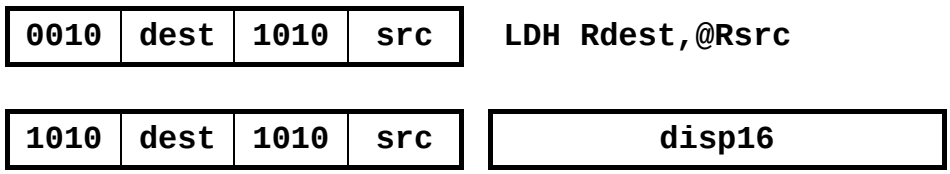
(1) LDH sign-extends the halfword data of the memory at the address specified by Rsrc and
(2) LDH sign-extends the halfword data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest. The displacement value is sign-extended to 32 bits before the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

The condition bit (C) dose not changed.

[EIT occurrence]

Address exception (AE)

[Encoding]



LDH Rdest,@(disp16,Rsrc)

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) dose not changed.

[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 = *(unsigned char *) Rsrc;

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

[Description]

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

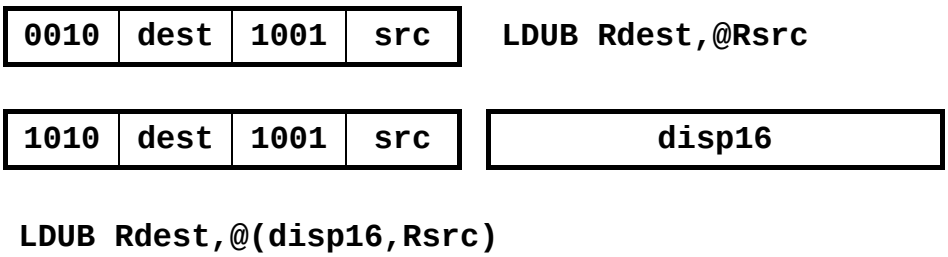
(2) LDUB zero-extends the byte data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest. The displacement value is sign-extended to 32 bits before address calculation.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]



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) dose not changed.

[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 = *(signed int *) Rsrc;

[Description]

The contents of the word at the memory location specified by Rsrc are loaded into Rdest.

The condition bit (C) dose not changed.

This instruction sets the LOCK bit in addition to simple loading. When the LOCK bit is 1, DMA transfer request or HOLD request 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

MACHI

DSP function instruction
Multiply-accumulate high-order halfword

MACHI

[Mnemonic]

MACHI Rsrc1,Rsrc2,Adest

[Function]

Multiply and add

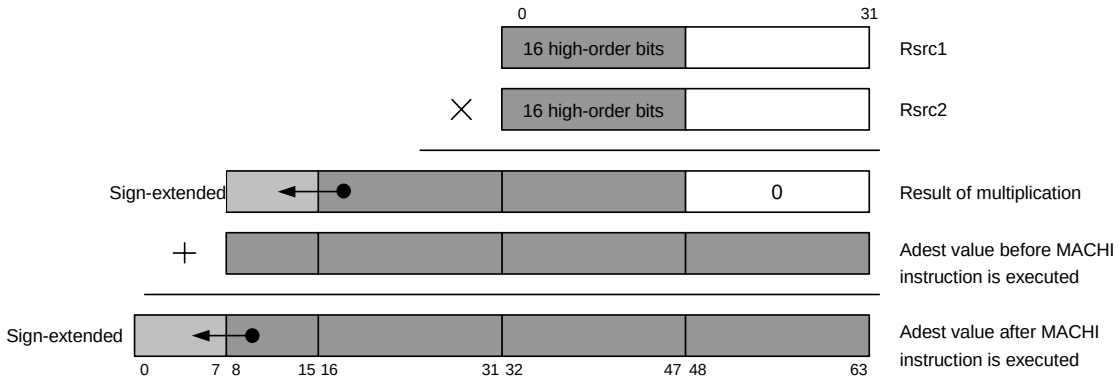
Adest += ((signed) (Rsrc1 & 0xffff0000) * (signed short) (Rsrc2 >> 16) ;

[Description]

MACHI multiplies the 16 high-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator Adest.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 8–15 of Adest are sign-extended before being added. The result of addition is stored in Adest. The 16 high-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 are handled as signed integers.

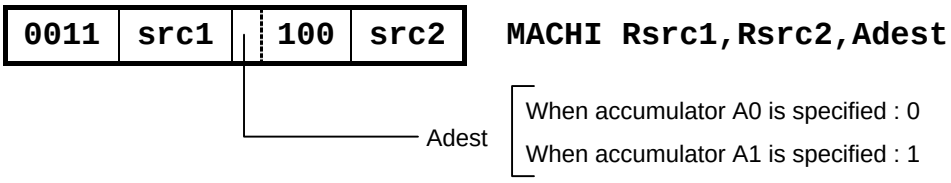
The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]



MACLH1

DSP function instruction
Multiply-accumulate low-order halfword and high-order halfword using accumulator 1

MACLH1

[Mnemonic]

MACLH1 Rsrc1,Rsrc2

[Function]

Multiply and Add

A1 += ((signed) (Rsrc1 << 16) * (signed short) (Rsrc2 >> 16));

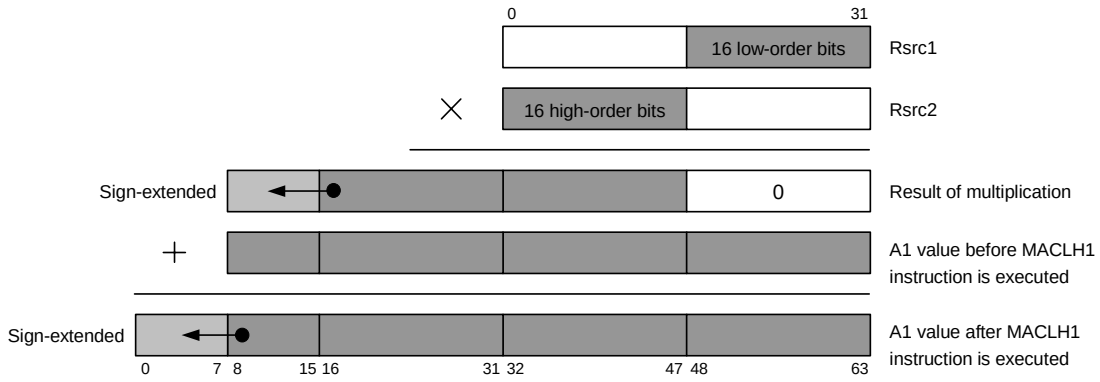
[Description]

MACWLH1 multiplies the 16 low-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator A1.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of A1 and those that correspond to bits 8–15 of A1 are sign-extended before being added. The result of addition is stored in A1. The 16 low-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 are handled as signed integers.

A0 does not change as a result of execution of this instruction.

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0101	src1	1100	src2

MACLH1 Rsrc1,Rsrc2

MACLO

DSP function instruction
Multiply-accumulate low-order halfword

MACLO

[Mnemonic]

MACLO Rsrc1,Rsrc2,Adest

[Function]

Multiply and Add

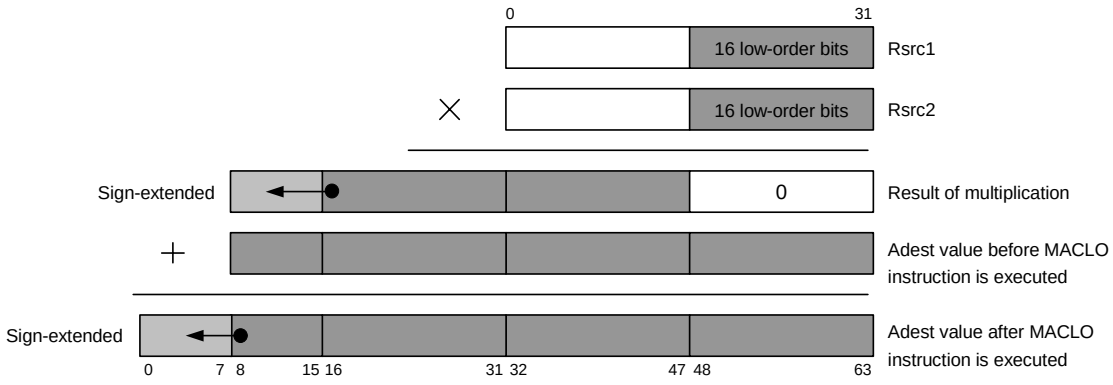
Adest += ((signed) (Rsrc1 << 16) * (signed short) Rsrc2) ;

[Description]

MACLO multiplies the 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator Adest.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 8–15 of Adest are sign-extended before being added. The result of addition is stored in Adest. The 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 are handled as signed integers.

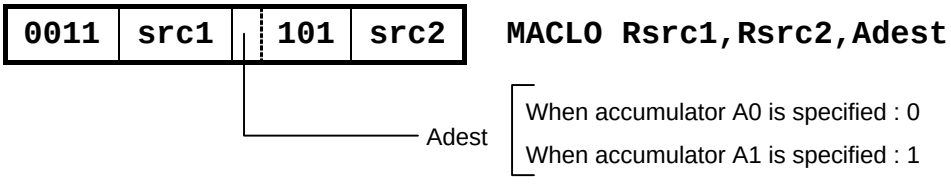
The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]



MACWHI

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

MACWHI

[Mnemonic]

MACWHI Rsrc1,Rsrc2

[Function]

Multiply and Add

A0 += ((signed) Rsrc1 * (signed short) (Rsrc2 >> 16));

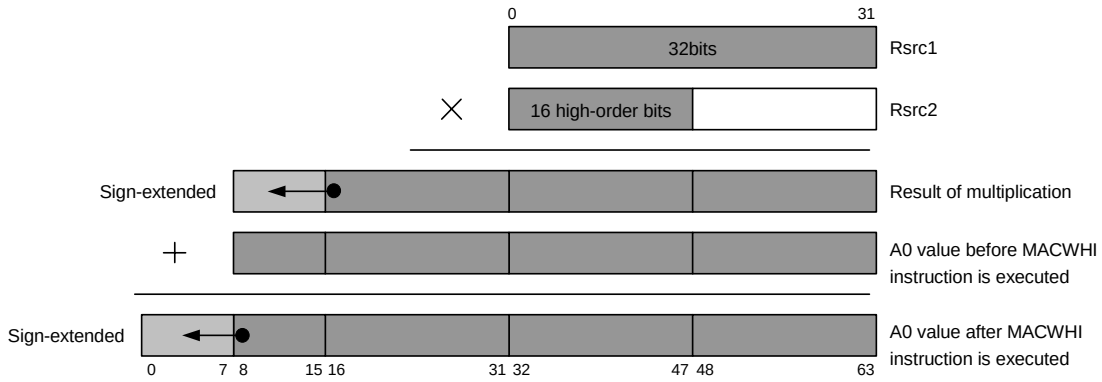
[Description]

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.

A1 does not change as a result of execution of this instruction.

The condition bit (C) dose not changed.



[EIT occurrence]

None

[Encoding]

0011	src1	0110	src2

MACWHI Rsrc1,Rsrc2

MACWLO

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

MACWLO

[Mnemonic]

MACWLO Rsrc1,Rsrc2

[Function]

Multiply and Add

A0 += ((signed) Rsrc1 * (signed short) Rsrc2) ;

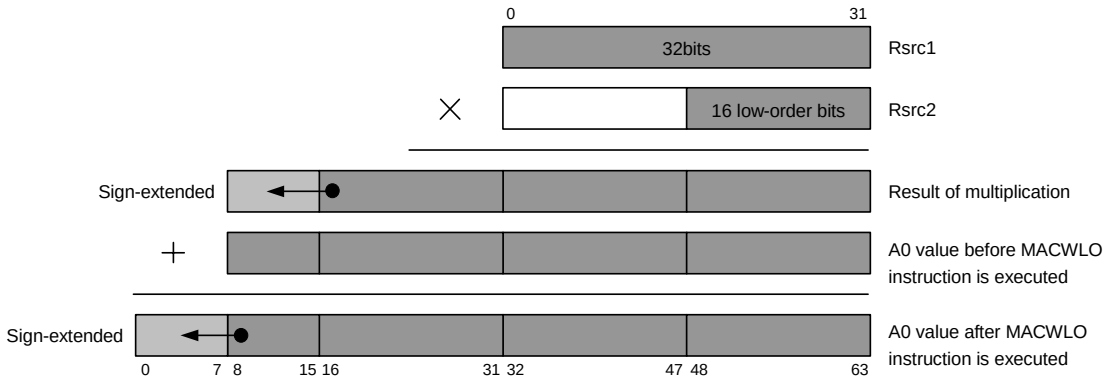
[Description]

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

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

A1 does not change as a result of execution of this instruction.

The condition bit (C) dose not changed.



[EIT occurrence]

None

[Encoding]

0011	src1	0111	src2

MACWLO Rsrc1,Rsrc2

MACWU1

DSP function instruction
Multiply-accumulate word and unsigned low-order halfword using accumulator 1

MACWU1

[Mnemonic]

MACWU1 Rsrc1,Rsrc2

[Function]

Multiply and Add

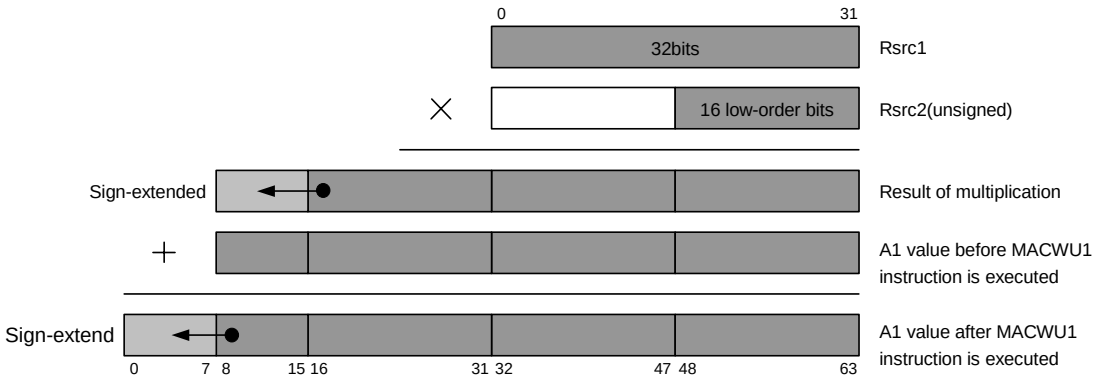
A1 += ((signed) Rsrc1 * (unsigned short) Rsrc2);

[Description]

MACWU1 multiplies the entire content (32 bits) of Rsrc1 and the 16 low-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator A1.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at the least significant bit of A1 and those that correspond to bits 8–15 of A1 are sign-extended before being added. The result of addition is stored in A1. The 32 bits of Rsrc1 are handled as a signed integer and the 16 low-order bits of Rsrc2 are handled as an unsigned integer

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0101	src1	1011	src2	MACWU1 Rsrc1,Rsrc2

MSBLO

DSP function instruction
Multiply low-order halfwords and subtract

MSBLO

[Mnemonic]

MSBLO Rsrc1,Rsrc2

[Function]

Multiply and Add

A0 -= ((signed) (Rsrc1 << 16) * (signed short) Rsrc2);

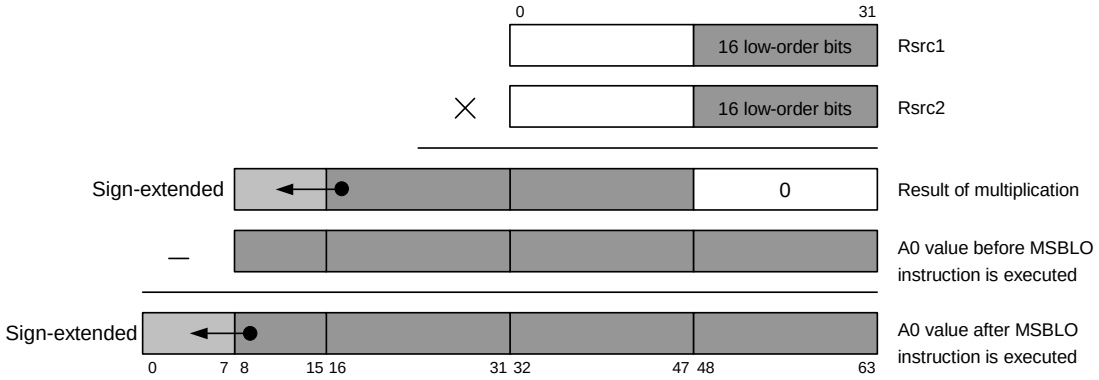
[Description]

Multiply the 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 together and subtract the result of multiplication from the 56 low-order bits of accumulator A0.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of A0 and those that correspond to bits 8–15 of A0 are sign-extended before subtraction. The result of subtraction is stored in A0. The 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 are handled as signed integers.

A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0101	src1	1101	src2	MSBLO Rsrc1,Rsrc2

MUL

multiply and divide instruction
Multiply

MUL

[Mnemonic]

MUL Rdest,Rsrc

[Function]

Multiply

```
{ signed64bit tmp;
  tmp = ( signed64bit ) Rdest * ( signed64bit ) Rsrc;
  Rdest = ( signed 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) dose not changed.

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,Adest

[Function]

Multiply

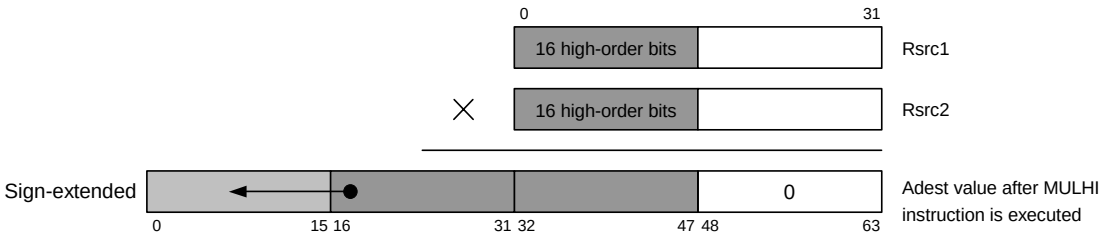
Adest = ((signed) (Rsrc1 & 0xffff0000) * (signed short) (Rsrc2 >> 16)) ;

[Description]

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

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 0–15 of Adest are sign-extended. Furthermore, the bits 48–63 of Adest are cleared to 0. The 16 high-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 are handled as signed integers.

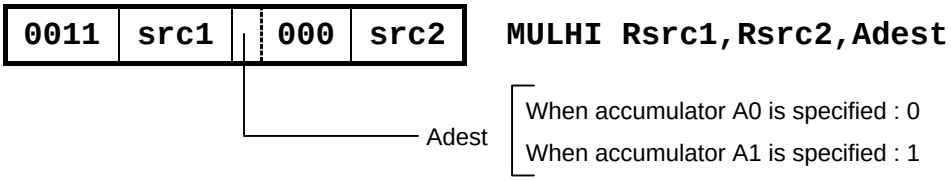
The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]



MULLO

DSP function instruction
Multiply low-order halfwords

MULLO

[Mnemonic]

MULLO Rsrc1,Rsrc2,Adest

[Function]

Multiply

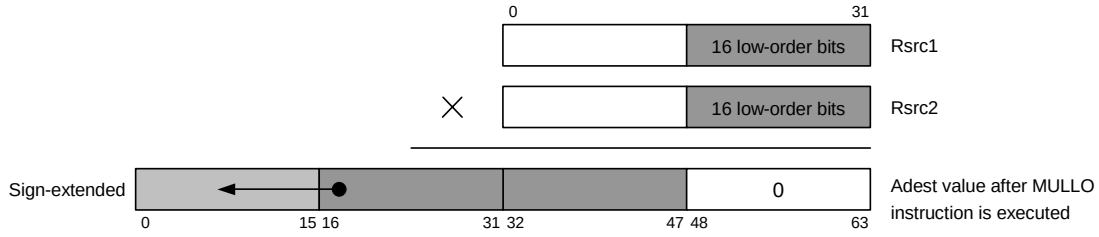
Adest = ((signed) (Rsrc1 << 16) * (signed short) Rsrc2) ;

[Description]

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

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 0–15 of Adest are sign-extended. Furthermore, the bits 48–63 of Adest are cleared to 0. The 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 are handled as signed integers.

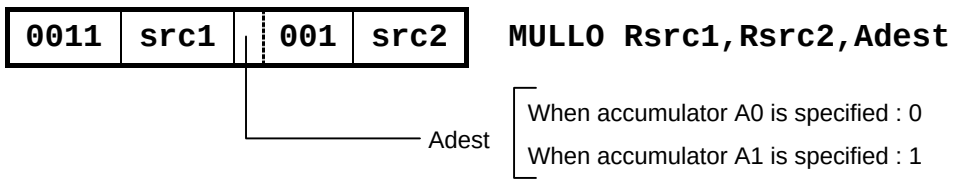
The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]



MULWHI

DSP function instruction
Multiply word and high-order halfword

MULWHI

[Mnemonic]

```
MULWHI Rsrc1,Rsrc2
```

[Function]

Multiply

A0 = ((signed) Rsrc1 * (signed short) (Rsrc2 >> 16));

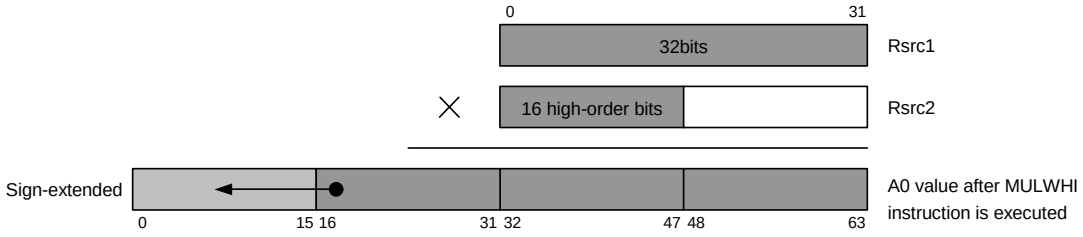
[Description]

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.

A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0011	src1	0010	src2

MULWHI Rsrc1,Rsrc2

MULWLO

DSP function instruction
Multiply word and low-order halfword

MULWLO

[Mnemonic]

```
MULWLO Rsrc1,Rsrc2
```

[Function]

Multiply

A0 = ((signed) Rsrc1 * (signed short) Rsrc2);

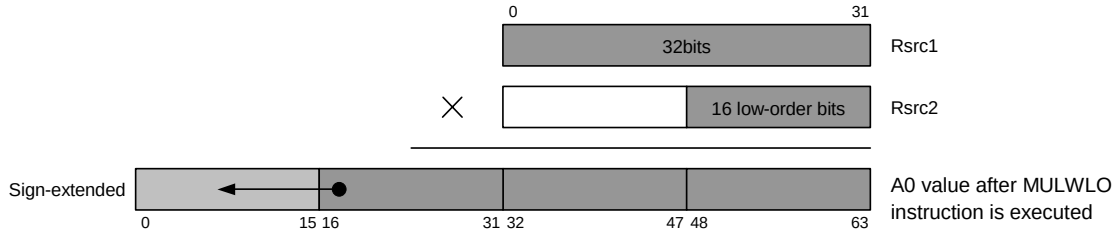
[Description]

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.

A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0011	src1	0011	src2	MULWLO Rsrc1,Rsrc2

MULWU1

DSP function instruction
Multiply word and unsigned low-order halfword
unsigned accumulator 1

MULWU1

[Mnemonic]

MULWU1 Rsrc1,Rsrc2

[Function]

Multiply

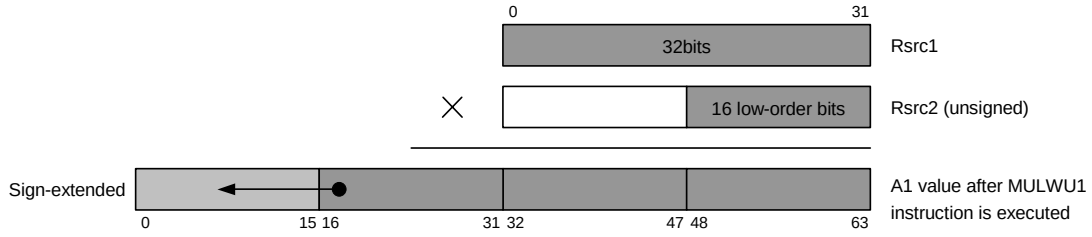
A1 = ((signed) Rsrc1 * (unsigned short) Rsrc2);

[Description]

MULWU1 multiplies the entire content (32 bits) of Rsrc1 and the 16 low-order bits of Rsrc2 together and stores the result in accumulator A1.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at the least significant bit of A1 and those that correspond to bits 0–15 of A1 are sign-extended. The 32 bits of Rsrc1 are handled as a signed integer and the 16 low-order bits of Rsrc2 are handled as an unsigned integer.

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0101	src1	1010	src2

MULWU1 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) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	**dest**	**1000**	**src**	**MV Rdest,Rsrc**

MVFACHI

DSP function instruction
Move from accumulator high-order word

MVFACHI

[Mnemonic]

```
MVFACHI Rdest,Asrc
```

[Function]

Transfer from accumulator to register

Rdest = (signed) (Asrc >> 32);

[Description]

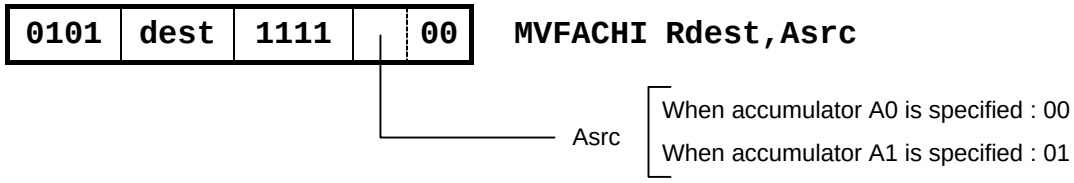
MVFACHI moves the high-order 32 bits of the accumulator Asrc to Rdest.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]



MVFACLO

DSP function instruction
Move from accumulator low-order word

MVFACLO

[Mnemonic]

```
MVFACLO Rdest,Asrc
```

[Function]

Transfer from accumulator to register

Rdest = (signed) Asrc;

[Description]

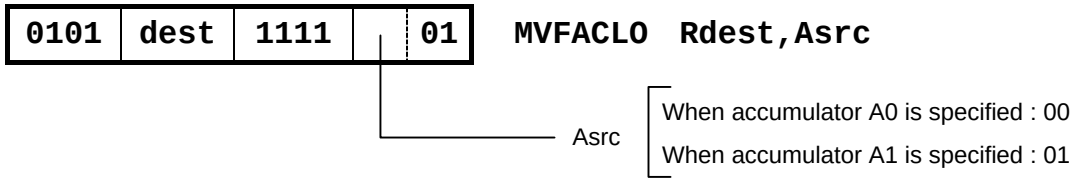
MVFACLO moves the low-order 32 bits of the accumulator Asrc to Rdest.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]



MVFACMI

DSP function instruction
Move middle-order word from accumulator

MVFACMI

[Mnemonic]

```
MVFACMI Rdest,Asrc
```

[Function]

Transfer from accumulator to register

Rdest = (signed) (Asrc >> 16);

[Description]

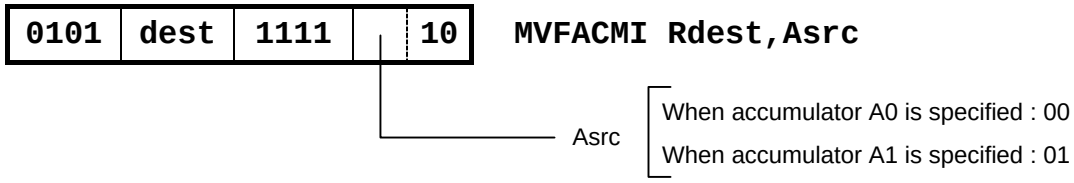
MVFACMI moves bits16 through 47 of the accumulator Asrc to Rdest.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]



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) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	dest	1001	src	MVFC Rdest,CRsrc

MVFCP

Coprocessor support instruction
Move from Coprocessor register

MVFCP

[Mnemonic]

MVFCP Rdest,CPRsrc,inst,cpid

[Function]

Transfer

Rdest = CPRsrc(num);

[Description]

MVFCP moves the content of CPRsrc register of the coprocessor specified by the coprocessor ID(cpid) to Rdest.

The bit field “inst” is provided for operation bits passed to the coprocessor. If additional processing needs to be performed while transferring data to the coprocessor, the necessary direction can be given to the coprocessor by setting this bit field.

The condition bit (C) does not change.

[EIT occurrence]

Coprocessor interrupt (CPI) or coprocessor disable exception (CDE)

[Encoding]

MVFCP Rdest,CPRsrc,inst,cpid

MVTACHI

DSP function instruction
Move high-order word to accumulator

MVTACHI

[Mnemonic]

```
MVTACHI Rsrc,Adest
```

[Function]

Transfer between accumulator and register

Adest[0 : 31] = Rsrc ;

[Description]

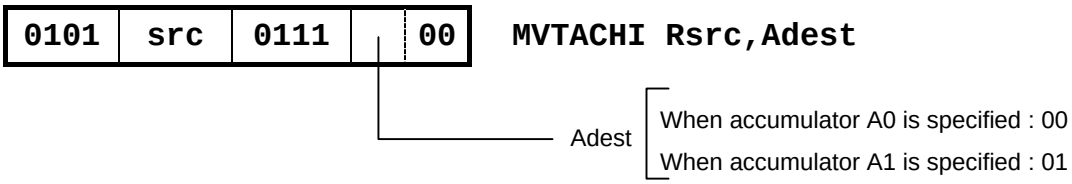
MVTACHI moves the content of Rsrc to the 32 high-order bits (bits 0–31) of accumulator Adest.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]



MVTACLO

DSP function instruction
Move low-order word to accumulator

MVTACLO

[Mnemonic]

MVTACLO Rsrc,Adest

[Function]

Transfer between accumulator and register

Adest [32 : 63] = Rsrc ;

[Description]

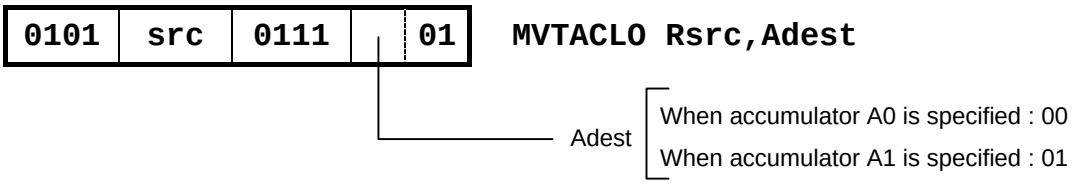
MVTACLO moves the content of Rsrc to the 32 low-order bits (bits 32–63) of accumulator Adest.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]



MVTC

transfer instruction
Move to control register

MVTC

[Mnemonic]

```
MVTC Rsrc,CRdest
```

[Function]

Transfer from register to control register

CRdest = Rsrc ;

[Description]

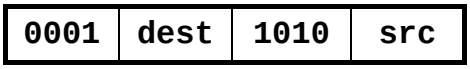
MVTC moves Rsrc to CRdest.

If PSW(CR0) is specified as CRdest, the condition bit (C) is changed; otherwise it dose not changed.

[EIT occurrence]

Privilege instruction exception(PIE)

[Encoding]



MVTC Rsrc,CRdest

MVTCP

Coprocessor support instruction
Move to Coprocessor register

MVTCP

[Mnemonic]

MVTCP Rsrc,CPRdest,inst,cpid

[Function]

Transfer

CPRdest(num) = Rsrc;

[Description]

MVTCP moves the content of Rsrc register to CPRdest register of the coprocessor specified by the coprocessor ID(cpid).

The bit field “inst” is provided for operation bits passed to the coprocessor. If additional processing needs to be performed while transferring data to the coprocessor, the necessary direction can be given to the coprocessor by setting this bit field.

The condition bit (C) does not change.

[EIT occurrence]

Coprocessor interrupt (CPI) or coprocessor disable exception (CDE)

[Encoding]

1101	src	0110	dest

cpid	0000	inst

MVTCP Rsrc,CPRdest,inst,cpid

NEG

arithmetic operation instruction
Negate

NEG

[Mnemonic]

```
NEG Rdest,Rsrc
```

[Function]

Negate

Rdest = 0 - (signed) 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) dose not changed.

[EIT occurrence]

None

[Encoding]

0000	dest	0011	src

NEG Rdest,Rsrc

NOP

branch instruction
No operation

NOP

[Mnemonic]

NOP

[Function]

No operation

/* */

[Description]

NOP performs no operation. The subsequent instruction then processed.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0111	0000	0000	0000	**NOP**

NOT

logic operation instruction
Logical NOT

NOT

[Mnemonic]

```
NOT Rdest,Rsrc
```

[Function]

Logical NOT

Rdest = ~Rsrc ;

[Description]

NOT inverts each of the bits of Rsrc and puts the result in Rdest.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0000	dest	1011	src

NOT Rdest,Rsrc

OR

logic operation instruction

OR

OR

[Mnemonic]

OR Rdest,Rsrc

[Function]

Logical OR

Rdest = Rdest | Rsrc ;

[Description]

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

The condition bit (C) dose not changed.

[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) dose not changed.

[EIT occurrence]

None

[Encoding]

1000	dest	1110	src	imm16

OR3 Rdest,Rsrc,#imm16

OPECP

Coprocessor support instruction
Operate Coprocessor

OPECP

[Mnemonic]

OPECP CPRdest,CPRsrc,inst,cpid

[Function]

Coprocessor operation
CPRdest = inst(CPRdest,CPRsrc)(cpid);

[Description]

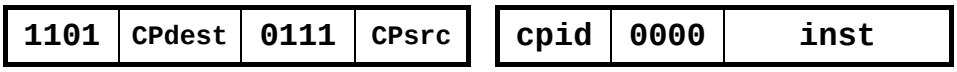
OPECP executes the coprocessor instruction specified by "inst" to the coprocessor is specified by coprocessor ID(cpid) and moves the result to CPRdest.

The condition bit (C) does not change.

[EIT occurrence]

Coprocessor interrupt (CPI) or coprocessor disable exception (CDE)

[Encoding]



OPECP CPRdest,CPRsrc,inst,cpid

PCMPBZ

compare instruction
Parallel compare byte to zero

PCMPBZ

[Mnemonic]

PCMPBZ Rsrc

[Function]

Compare

C =(((Rsrc[0:7] ==0) | | (Rsrc[8:15] ==0)| |(Rsrc[16:23] ==0)| | (Rsrc[24:31] ==0)) ? 1 :0)

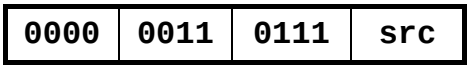
[Description]

Rsrc is assumed to be consisting of four packed 8-bit data. When one of these four 8-bit data = 0, the condition bit (C) is set to 1.

[EIT occurrence]

None

[Encoding]



PCMPBZ Rsrc

RAC

DSP function instruction
Round accumulator

RAC

[Mnemonic]

RAC Adest,Asrc,#imm1

[Function]

Round
{ signed64bit tmp;
  tmp = (signed64bit)Asrc << imm1;
  tmp = tmp + 0x0000 0000 0000 8000;
  if (tmp > (signed64bit) 0x0000 7fff ffff 0000)
    Adest = 0x0000 7fff ffff 0000;
  else if (tmp < (signed64bit) 0xffff 8000 0000 0000)
    Adest = 0xffff 8000 0000 0000;
  else
    Adest = tmp & 0xffff ffff ffff 0000;
}
(imm1 = 1, 2;)

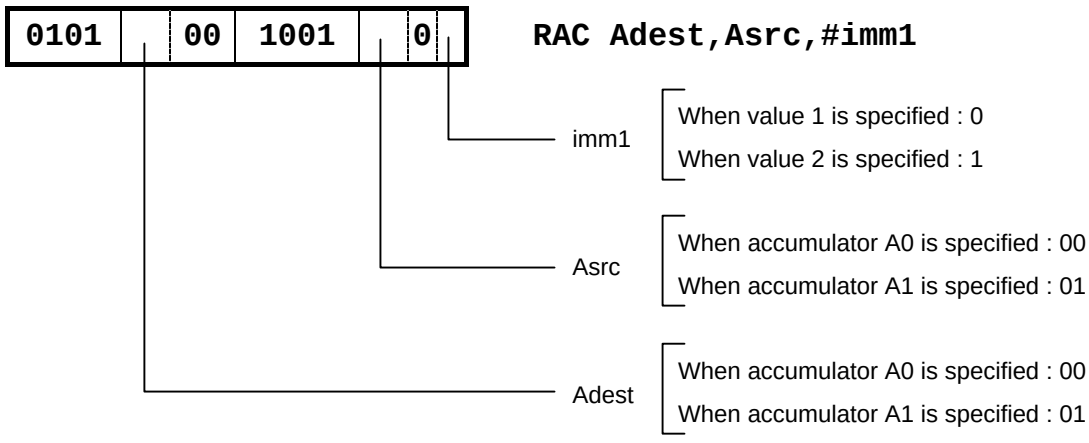
[Description]

RAC rounds the contents in the accumulator to word size and stores the result in the accumulator.
The condition bit (C) does not change.

[EIT occurrence]

None

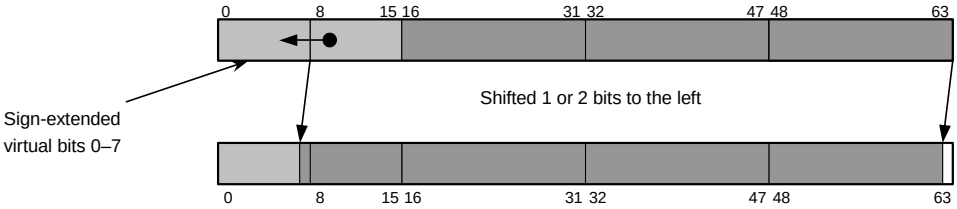
[Encoding]



[Supplementary explanation]

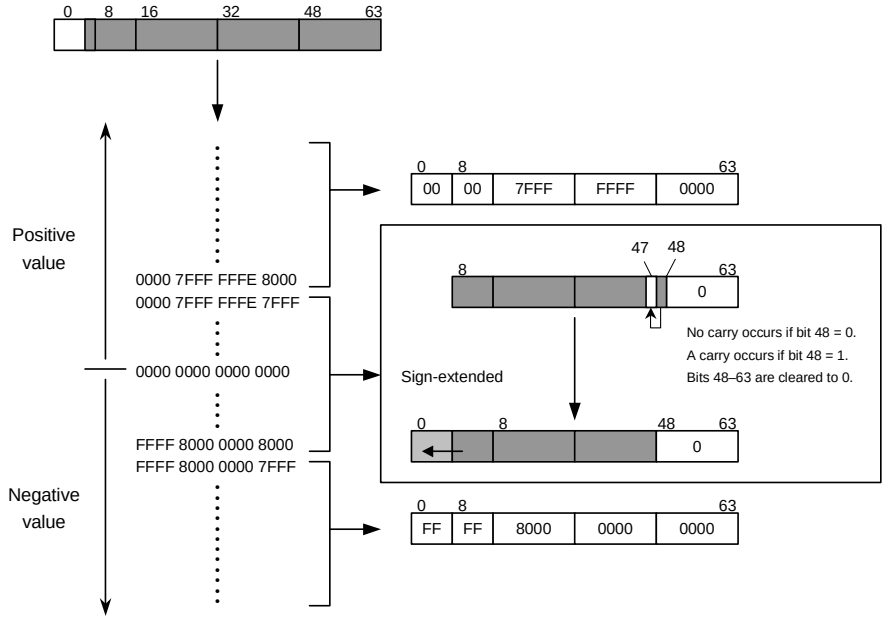
The RAC instruction is executed following a procedure similar to the one described below.

■ Procedure 1



■ Procedure 2

The accumulator value changes according to a 64-bit value consisting of virtual bits 0–7 in which the 1 or 2-bit shift is reflected plus the left-shifted bits 8–63.



RACH

DSP function instruction
Round accumulator halfword

RACH

[Mnemonic]

RACH Adest,Asrc,#imm1

[Function]

```
Round
{ signed64bit tmp;
   tmp = ( signed64bit )Asrc << imm1;
   tmp = tmp + 0x0000 0000 8000 0000;
   if( tmp > ( signed64bit ) 0x0000 7fff 0000 0000 )
      Adest = 0x0000 7fff 0000 0000;
   else if ( tmp < ( signed64bit) 0xffff 8000 0000 0000 )
      Adest = 0xffff 8000 0000 0000;
   else
      Adest = tmp & 0xffff ffff 0000 0000;
}
( imm1 = 1, 2; )
```

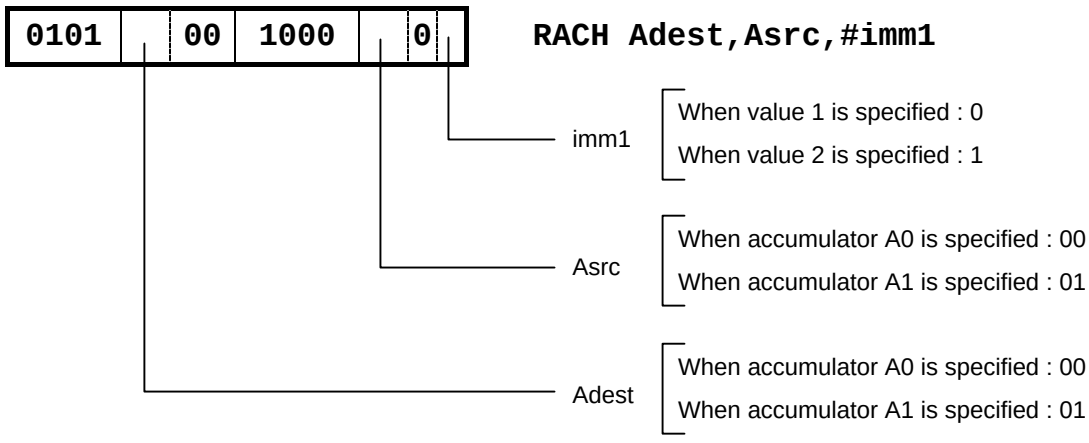
[Description]

RAC rounds the accumulator value to a halfword size and store the result in the accumulator.
The condition bit (C) does not change.

[EIT occurrence]

None

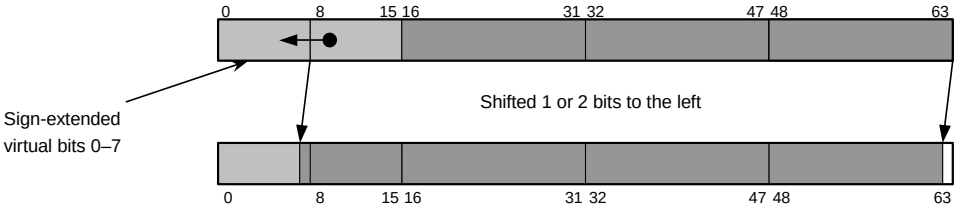
[Encoding]



[Supplementary explanation]

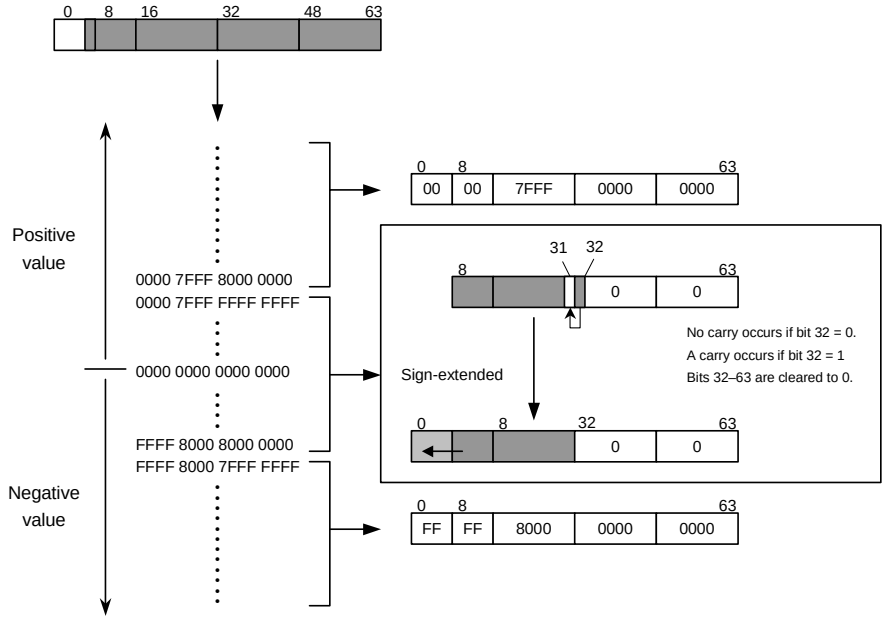
The RACH instruction is executed following a procedure similar to the one described below.

■ Procedure 1



■ Procedure 2

The accumulator value changes according to a 64-bit value consisting of the left-shifted bits 8–63 plus virtual bits 0–7 in which the value of 1 or 2 shifted out bits is reflected.



REM

multiply and divide instruction
Remainder

REM

[Mnemonic]

REM Rdest,Rsrc

[Function]

Signed remainder

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

[Description]

REM divides Rdest by Rsrc and stores the remainder in Rdest. The operands are treated as signed 32-bit values. The quotient is rounded toward zero and the remainder takes the same sign as the dividend.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0010	src

0000	0000	0000	0000

REM Rdest,Rsrc

REMB

multiply and divide instruction
Remainder byte

REMB

[Mnemonic]

REMB Rdest,Rsrc

[Function]

Signed remainder

Rdest =(signed char)Rdest % (signed)Rsrc ;

[Description]

REMB divides Rdest by Rsrc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as a signed 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as a signed 32-bit value. The quotient is rounded toward 0, and the remainder takes the same sign as the divisor.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0010	src

0000	0000	0001	1000

REMB Rdest,Rsrc

REMH

multiply and divide instruction
Remainder halfword

REMH

[Mnemonic]

REMH Rdest,Rsrc

[Function]

Signed remainder

Rdest =(signed short)Rdest % (signed)Rsrc ;

[Description]

REMH divides Rdest by Rsrc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as a signed 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as a signed 32-bit value. The quotient is rounded toward 0, and the remainder takes the same sign as the divisor.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0010	src

0000	0000	0001	0000

REMH Rdest,Rsrc

REMU

multiply and divide instruction
Remainder unsigned

REMU

[Mnemonic]

REMU Rdest,Rsrc

[Function]

Unsigned remainder

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

[Description]

REMU divides Rdest by Rsrc and stores the remainder in Rdest. The operands are treated as unsigned 32-bit values.

The condition bit (C) does not changed.

When Rsrc is zero, Rdest does not changed.

[EIT occurrence]

None

[Encoding]

1001	dest	0011	src

0000	0000	0000	0000

REMU Rdest,Rsrc

REMUB

multiply and divide instruction
Remainder unsigned byte

REMUB

[Mnemonic]

REMUB Rdest,Rsrc

[Function]

Unsigned remainder

Rdest =(unsigned char)Rdest % (unsigned)Rsrc ;

[Description]

REMUB divides Rdest by Rsrc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as an unsigned 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as an unsigned 32-bit value.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

1001	dest	0011	src

0000	0000	0001	1000

REMUB Rdest,Rsrc

REMUH

multiply and divide instruction
Remainder unsigned halfword

REMUH

[Mnemonic]

REMUH Rdest,Rsrc

[Function]

Unsigned remainder

Rdest =(unsigned short)Rdest % (unsigned)Rsrc ;

[Description]

REMUB divides Rdest by Rsrc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as an unsigned 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as an unsigned 32-bit value.

The condition bit (C) does not change.

When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

REMUH Rdest,Rsrc

RTE

EIT-related instruction
Return from EIT

RTE

[Mnemonic]

RTE

[Function]

Return from EIT handler

SM = BSM ;
IE = BIE ;
PM = BPM
CE = BCE
C = BC ;
PC = BPC & 0xfffffffe ;

[Description]

Restore the SM, IE, PM, CE and C bits of the PSW register from the respective backup bits BSM, BIE, BPM, BCE and C, and branch to the address indicated by BPC.

[EIT occurrence]

Privilege instruction exception(PIE)

[Encoding]

0001	0000	1101	0110	**RTE**

SADD

DSP function instruction
Add accumulators

SADD

[Mnemonic]

SADD

[Function]

Add

A0 = ((signed) A0 + (signed) ((signed) A1 >> 16));

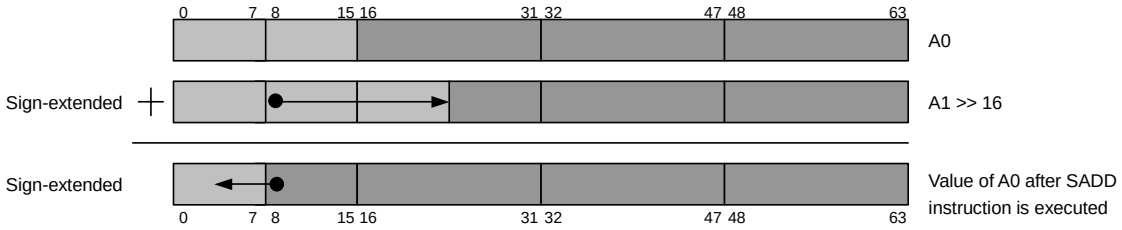
[Description]

SADD adds accumulator A0 and accumulator A1 that have been arithmetically shifted 16 bits right and stores the result in A0.

The values of A0 and A1 that have been shifted 16 bits right are handled as signed integers.

The accumulator A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.



[EIT occurrence]

None

[Encoding]

0101	0000	1110	0100	SADD

SATB

DSP function instruction
Saturate word into Byte

SATB

[Mnemonic]

SATB Rdest,Rsrc

[Function]

Saturation processing
{
if ((signed char) 0x7f <= (signed) Rsrc)
    Rdest = 0x0000007f;
else if ((signed) Rsrc <= (signed char) 0x80;)
    Rdest = 0xffffff80;
else
    Rdest = Rsrc;
};

[Description]

SATB rounds the value of Rsrc to a byte size (saturation processing) and stores the result in Rdest.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

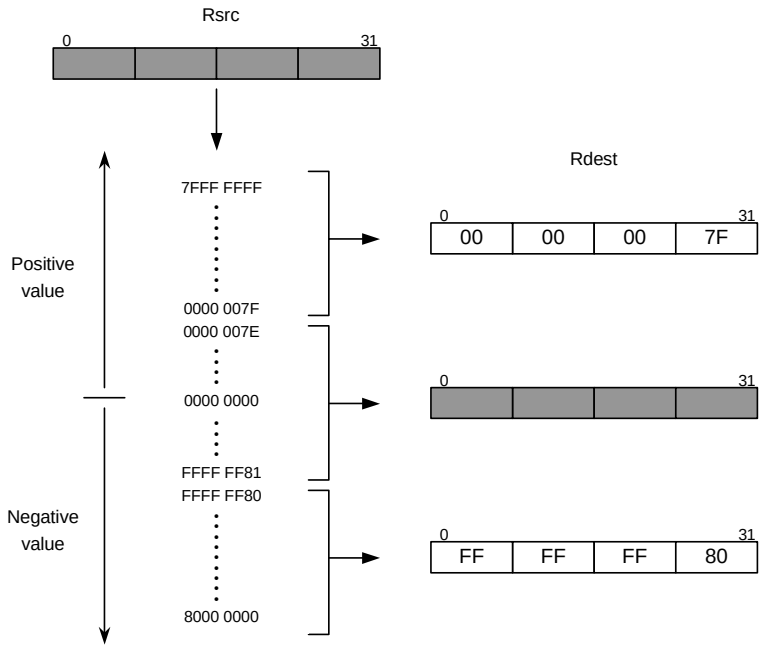
1000	dest	0110	src

0000	0011	0000	0000

SATB Rdest,Rsrc

[Supplementary explanation]

The value of Rdest changes according to the value of Rsrc.



SATH

DSP function instruction
Saturate word into Half-word

SATH

[Mnemonic]

SATH Rdest,Rsrc

[Function]

Saturation processing
{
if ((signed short) 0x7fff <= (signed) Rsrc)
    Rdest = 0x00007fff;
else if ((signed) Rsrc <= (signed short) 0x8000)
    Rdest = 0xffff8000;
else
    Rdest = Rsrc;
}

[Description]

SATH rounds the value of Rsrc to a halfword size (saturation processing) and stores the result in Rdest.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

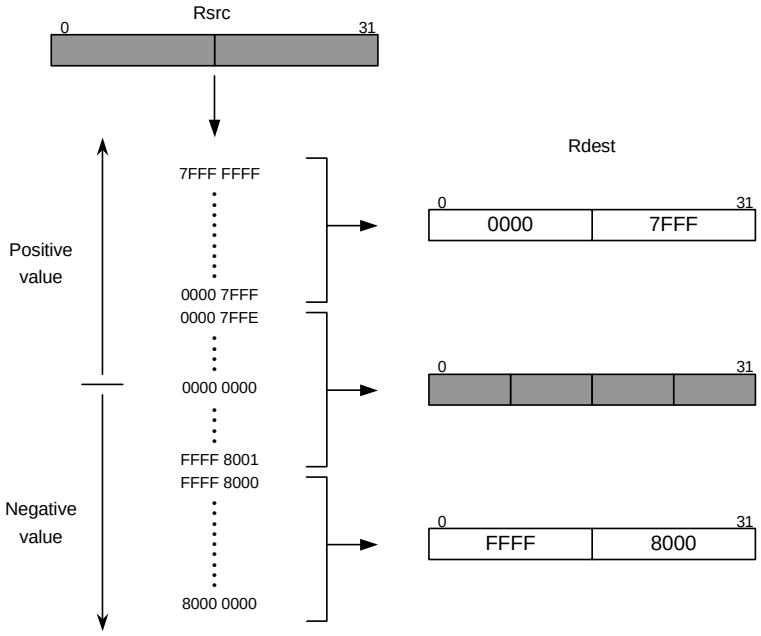
1000	dest	0110	src

0000	0010	0000	0000

SATH Rdest,Rsrc

[Supplementary explanation]

The value of Rdest changes according to the value of Rsrc.



SC

branch
Skip on C-bit

SC

[Mnemonic]

SC

[Function]

Conditional skip
if (C ==1)Cancel parallel execution of the next 16-bit instruction ;

[Description]

When the condition bit (C) = 1, cancel the 16-bit instruction to be executed at the same time and skip to the next instruction.

This instruction is used for conditional execution of another instruction to be executed in parallel with it (executed when C = 0), and is effective for only parallel instruction execution.

However, A combination of 16-bit instructions that can be executed simultaneously with the SC instruction is both-side instructions(OS) and right-side instructions(-S).

[EIT occurrence]

None

[Encoding]

0111	0100	0000	0001	**SC**

SETPSW

bit operation instruction
Set PSW

SETPSW

[Mnemonic]

SETPSW #imm8

[Function]

Set SM, IE, PM, CE or C bit in the PSW to 1.

PSW |= (unsigned char) imm8;

[Description]

Logically OR the 8-bit value specified by imm8 with the 8 low-order bits in the PSW (bits 24–31) bitwise and write the result to the 8 low-order bits in the PSW bit by bit.

Make sure that all of the unsupported PWS bits in the microcomputer used are set to 0.

[EIT occurrence]

Privilege instruction exception(PIE)

[Encoding]

0111	**0001**	**imm8**	**SETPSW #imm8**

SETH

transfer instruction
Set high-order 16-bit

SETH

[Mnemonic]

SETH Rdest,#imm16

[Function]

Transfer instruction

Rdest = (signed short) imm16 << 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) dose not changed.

[EIT occurrence]

None

[Encoding]

1101	dest	1100	0000	imm16

SETH Rdest,#imm16

SLL

shift instruction
Shift left logical

SLL

[Mnemonic]

```
SLL Rdest,Rsrc
```

[Function]

Logical left shift

Rdest = Rdest << (Rsrc & 31) ;

[Description]

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

Only the five least significant bits of Rsrc are used.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	dest	0100	src

SLL Rdest,Rsrc

SLL3

shift instruction
Shift left logical 3-operand

SLL3

[Mnemonic]

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

[Function]

Logical left shift

Rdest = Rsrc << (imm16 & 31) ;

[Description]

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

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1001	dest	1100	src	imm16

SLL3 Rdest,Rsrc,#imm16

SLLI

shift instruction
Shift left logical immediate

SLLI

[Mnemonic]

SLLI Rdest,#imm5

[Function]

Logical left shift

Rdest = Rdest << imm5 ;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0101	dest	010	imm5

SLLI Rdest,#imm5

SNC

branch

Skip on not C- bit

SNC

[Mnemonic]

SNC

[Function]

Conditional skip

if (C ==0)Cancel parallel execution of the next 16- bit instruction ;

[Description]

When the condition bit (C) = 0, cancel the 16-bit instruction to be executed at the same time and skip to the next instruction.

This instruction is used for conditional execution of another instruction to be executed in parallel with it (executed when C = 1), and is effective for only parallel instruction execution.

However, A combination of 16-bit instructions that can be executed simultaneously with the SNC instruction is both-side instructions(OS) and right-side instructions(-S).

[EIT occurrence]

None

[Encoding]

0111	0101	0000	0001	**SNC**

SRA

shift instruction
Shift right arithmetic

SRA

[Mnemonic]

SRA Rdest,Rsrc

[Function]

Arithmetic right shift

Rdest = (signed) Rdest >> (Rsrc & 31) ;

[Description]

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

Only the five least significant bits are used.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	dest	0010	src	SRA Rdest,Rsrc

SRA3

shift instruction
Shift right arithmetic 3-operand

SRA3

[Mnemonic]

SRA3 Rdest,Rsrc,#imm16

[Function]

Arithmetic right shift

Rdest = (signed) Rsrc >> (imm16 & 31) ;

[Description]

SRA3 right arithmetic-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, replicates the sign bit in Rsrc and puts the result in Rdest.

Only the five least significant bits are used.

The condition bit (C) dose not changed.

[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]

Logical right shift

Rdest = (signed) Rdest >> imm5 ;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0101	dest	001	imm5

SRAI Rdest,#imm5

SRL

shift instruction
Shift right logical

SRL

[Mnemonic]

SRA Rdest,Rsrc

[Function]

Logical right shift

Rdest = (unsigned) Rdest >> (Rsrc & 31) ;

[Description]

SRL right logical-shifts the contents of Rdest by the number specified by Rsrc, shifts zeroes into the most significant bits and puts the result in Rdest.

Only the five least significant bits of Rsrc are used.

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0001	dest	0000	src	SRL Rdest,Rsrc

SRL3

shift instruction
Shift right logical 3-operand

SRL3

[Mnemonic]

SRL3 Rdest,Rsrc,#imm16

[Function]

Logical right shift

Rdest = (unsigned) Rsrc >> (imm16 & 31) ;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1001	dest	1000	src	imm16

SRL3 Rdest,Rsrc,#imm16

SRLI

shift instruction
Shift right logical immediate

SRLI

[Mnemonic]

SRLI Rdest,#imm5

[Function]

Logical right shift

Rdest = (unsigned) Rdest >> (imm5 & 31) ;

[Description]

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

The condition bit (C) dose not changed.

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

[Description]

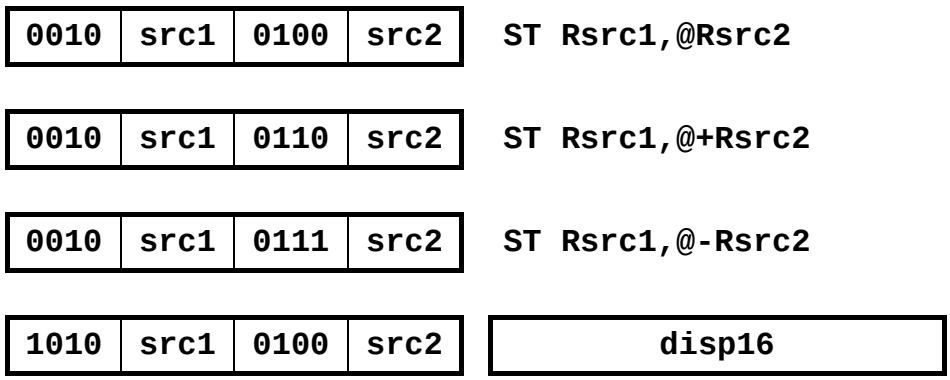
(1) ST stores Rsrc1 in the memory at the address specified by Rsrc2.
(2) ST increments Rsrc2 by 4 and stores Rsrc1 in the memory at the address specified by the resultant Rsrc2.
(3) ST decrements Rsrc2 by 4 and stores the contents of Rsrc1 in the memory at the address specified by the resultant Rsrc2.
(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) dose not changed.

[EIT occurrence]

Address exception (AE)

[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,@Rsrc2+**
(3) **STB Rsrc1,@(disp16,Rsrc2)**

[Function]

Store

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

(2) *(signed char *)Rsrc2 = Rsrc1, Rsrc2 += 1;

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

[Description]

(1) STB stores the byte data on the LSB side of Rsrc in a memory location whose address is specified by Rdest.

(2) STB stores the byte data on the LSB side of Rsrc in a memory location whose address is specified by Rdest, and then increment Rdest by 1.

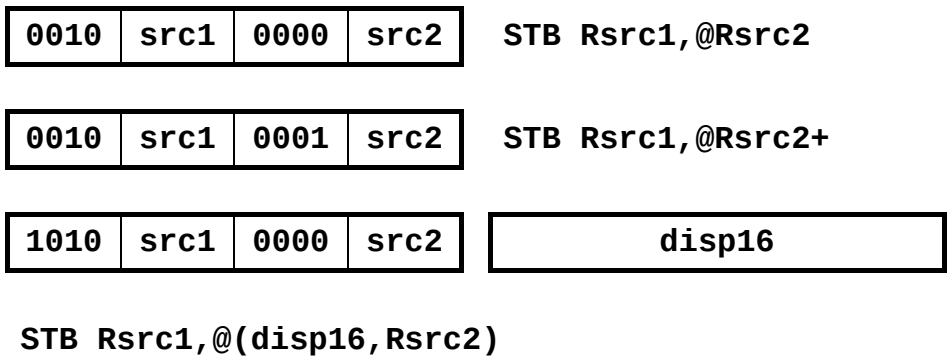
(3) STB stores the byte data on the LSB side of Rsrc1 in a memory location whose address is specified by Rdest and a 16-bit displacement. The displacement is sign-extended before address calculation.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]



STH

load/store instruction
Store halfword

STH

[Mnemonic]

(1) **STH Rsrc1,@Rsrc2**
(2) **STH Rsrc1,@Rsrc2+**
(3) **STH Rsrc1,@(disp16,Rsrc2)**

[Function]

Store

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

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

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

[Description]

(1) STH stores the halfword data on the LSB side of Rsrc in a memory location whose address is specified by Rdest.

(2) STH stores the halfword data on the LSB side of Rsrc in a memory location whose address is specified by Rdest, and then increment Rdest by 2.

(3) STH stores the halfword data on the LSB side of Rsrc in a memory location whose address is specified by Rdest and a 16-bit displacement. The displacement is sign-extended before address calculation.

The condition bit (C) does not change.

[EIT occurrence]

Address exception (AE)

[Encoding]

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

STH Rsrc1,@(disp16,Rsrc2)

SUB

arithmetic operation instruction
Subtract

SUB

[Mnemonic]

SUB Rdest,Rsrc

[Function]

Subtract

Rdest = Rdest - Rsrc;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

0000	dest	0010	src

SUB Rdest,Rsrc

SUBV

arithmetic operation instruction
Subtract with overflow checking

SUBV

[Mnemonic]

SUBV Rdest,Rsrc

[Function]

Subtract

Rdest = (signed) Rdest - (signed) Rsrc;

C = overflow ? 1 : 0;

[Description]

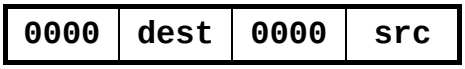
SUBV subtracts Rsrc from Rdest and puts the result in Rdest.

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

[EIT occurrence]

None

[Encoding]



SUBV Rdest,Rsrc

SUBX

arithmetic operation instruction
Subtract with borrow

SUBX

[Mnemonic]

SUBX Rdest,Rsrc

[Function]

Subtract

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

C = borrow ? 1 : 0;

[Description]

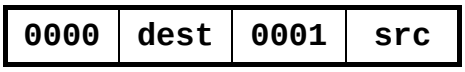
SUBX subtracts Rsrc and C from Rdest and puts the result in Rdest.

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

[EIT occurrence]

None

[Encoding]



SUBX Rdest,Rsrc

TRAP

EIT-related instruction
Trap

TRAP

[Mnemonic]

TRAP #imm4

[Function]

Generate TRAP
BPC = NextPC; (NextPC denotes the PC of the next instruction)
BSM = SM;
BIE = IE;
BPM = PM
BCE = CE
BC = C;
IE = 0;
PM =0;
CE = 0
C = 0;
call_trap_handler(imm4);

[Description]

Generate a trap of the specified number.
The values of the SM, IE, PM, CE and C bits in the PSW register are saved to the respective backup bits BSM, BIE, BPM, BCE and BC, and the IE, PM, CE and C bits each are updated 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) { * (signed 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) dose not changed. 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 accepts by using the LOCK and UNLOCK instructions.

[EIT occurrence]

Address exception (AE)

[Encoding]

0010	src1	0101	src2	**UNLOCK Rsrc1,@Rsrc2**

XOR

logic operation instruction
Exclusive OR

XOR

[Mnemonic]

```
XOR Rdest,Rsrc
```

[Function]

Exclusive OR

Rdest = Rdest ^ 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) dose not changed.

[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 = Rsrc ^ (unsigned short) imm16;

[Description]

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

The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

1000	dest	1101	src	imm16

XOR3 Rdest,Rsrc,#imm16

3.3 Notes about the BCL and BNCL Instructions

If the BCL or BNCL instruction is located at a word boundary and the 16-bit instruction in the latter half of the word boundary is a sequentially executed instruction, it depends on the value of the C bit whether the latter half 16-bit instruction is executed. Therefore, be especially careful when the BCL or BNCL instruction is located at a word boundary and the instruction is followed by a 16-bit instruction in the latter half of the word boundary.

If instruction codes are located in the manner shown below and BCL (or BNCL) branches off upon C bit = 1, the latter half 16-bit instruction is not executed. This is because the jump addresses in the OPSP-CPU should always be aligned with word boundaries (except when the RTE instruction is executed). If C bit = 0 and BCL (or BNCL) does not branch, the latter half 16-bit instruction is executed.

It depends on the value of the C bit whether the xxxx instruction is executed.

On the other hand, if the latter half 16-bit instruction (yyyy instruction) is a parallel executed instruction, the yyyy instruction is executed at the same time BCL (or BNCL) is executed.

The yyyy instruction is executed in parallel with BCL (or BNCL).

3.4 Exception and Trap Handling during Parallel Instruction Execution

During parallel instruction execution, exceptions and traps are handled in the manner shown below.

Table 3.4.1 Exception and trap handling during parallel instruction execution

O pipe(left side) instruction	S pipe (right side) instruction	Operation
RIE	Any instruction	RIE occurs. Instruction on neither the left side nor the right side is executed.
RIE	RIE	RIE occurs. Instruction on neither the left side nor the right side is executed.
Any instruction	RIE	RIE occurs. Instruction on neither the left side nor the right side is executed.
PIE	Any instruction	PIE occurs. Instruction on neither the left side nor the right side is executed.
PIE	RIE	RIE occurs. Instruction on neither the left side nor the right side is executed.
AE	Any instruction	AE occurs. Instruction on neither the left side nor the right side is executed.
AE	RIE	RIE occurs. Instruction on neither the left side nor the right side is executed.
TRAP	Any instruction	TRAP occurs. Instructions on both the left and right sides are executed.
TRAP	RIE	RIE occurs. Instruction on neither the left side nor the right side is executed.

This page is blank for reasons of layout.

APPENDICES

Appendix 1 Mechanism of Pipelined Instruction Processing

Appendix 1.1 Outline of Pipelined Instruction Processing

The OPSP CPU core has two pipelines (O pipe and S pipe). These two pipelines each consist of five pipeline stages. Parallel instruction execution by the OPSP CPU core is accomplished by using these two pipelines at the same time. (For details about combinatorial instructions that can be executed in parallel at the same time, refer to Section 2.5, “Parallel Instruction Execution.”)

■ Operation of the O pipeline and outline of each stage

(1) IF stage (instruction fetch stage)

This is the stage in which the CPU fetches instructions. Instructions are fetched from memory (cache).

The OPSP CPU has an instruction queue, so that it continues fetching instructions until the instruction queue is filled, regardless of whether decode processing in the D (decode) stage has finished.

(2) D stage (decode stage)

In the D stage, the CPU decodes instructions (DEC). At this time, the CPU reads out a register (RF) and if the result of the immediately preceding instruction needs to be referenced, it performs bypass processing (BYP). However, the bypass processing is performed only when the immediately preceding instruction is a register-to-register transfer or arithmetic operation instruction or a DSP function instruction.

(3) E stage (execution stage)

In this stage, the CPU performs arithmetic operation or address calculation (OP).

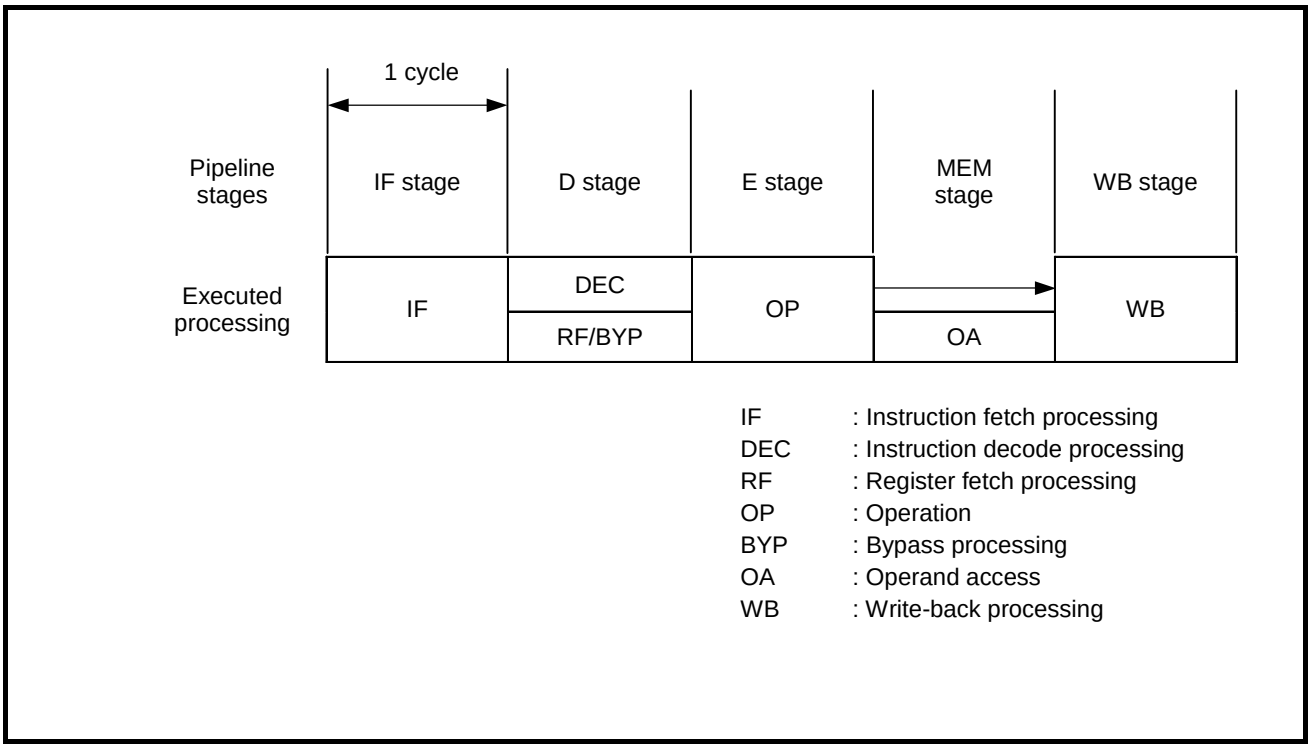
(4) MEM stage (memory access stage)

In this stage, the CPU accesses the operand (OA). This stage is used only when executing load/store instructions.

(5) WB stage (write-back stage)

In this stage, the CPU writes the operation result or fetched data to a register.

Appendix Figure 1.1 shows a structure of the O pipeline and the operation performed in it.



Appendix Figure 1.1 Structure of the O Pipeline and the Operation Performed in It

■ Operation of the S pipeline and outline of each stage

(1) IF stage (instruction fetch stage)

This is the stage in which the CPU fetches instructions. Instructions are fetched from memory (cache).

The OPSP CPU has an instruction queue, so that it continues fetching instructions until the instruction queue is filled, regardless of whether decode processing in the D (decode) stage has finished.

(2) D stage (decode stage)

In the D stage, the CPU decodes instructions (DEC). At this time, the CPU reads out a register (RF) and if the result of the immediately preceding instruction needs to be referenced, it performs bypass processing (BYP). However, the bypass processing is performed only when the immediately preceding instruction is a register-to-register transfer or arithmetic operation instruction or a DSP function instruction.

(3) E1 stage (execution stage 1)

In this stage, the CPU performs arithmetic operation or transfers data to registers or accumulators (OP1).

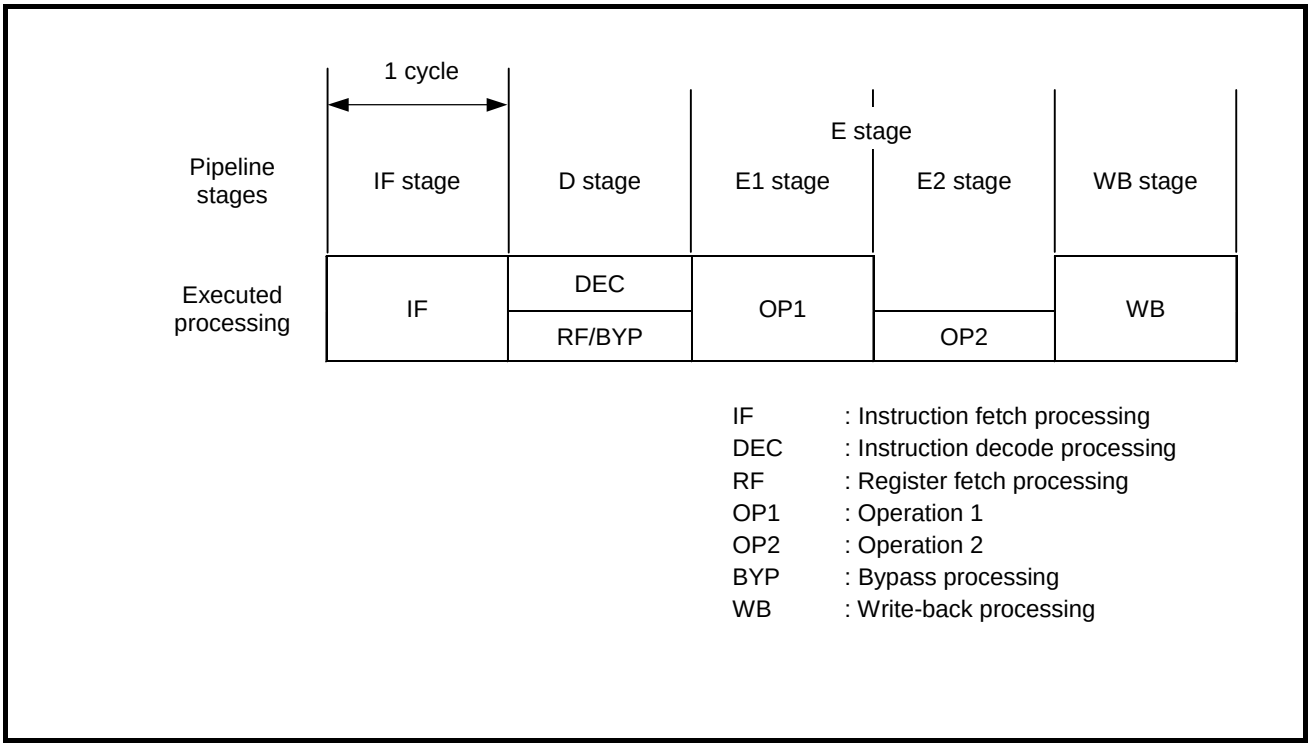
(4) E2 stage (execution stage 2)

This stage is used for DSP function instructions that write data to accumulators (OP2). In this case, the CPU requires two execution cycles because two execution stages (E1 and E2) are used. This stage is not used for other instructions that do not write operation results to accumulators. These instructions are sent directly to the next WB stage.

(5) WB stage (write-back stage)

Operation results are written to registers or accumulators.

Appendix Figure 1.2 shows a structure of the S pipeline and the operation performed in it.



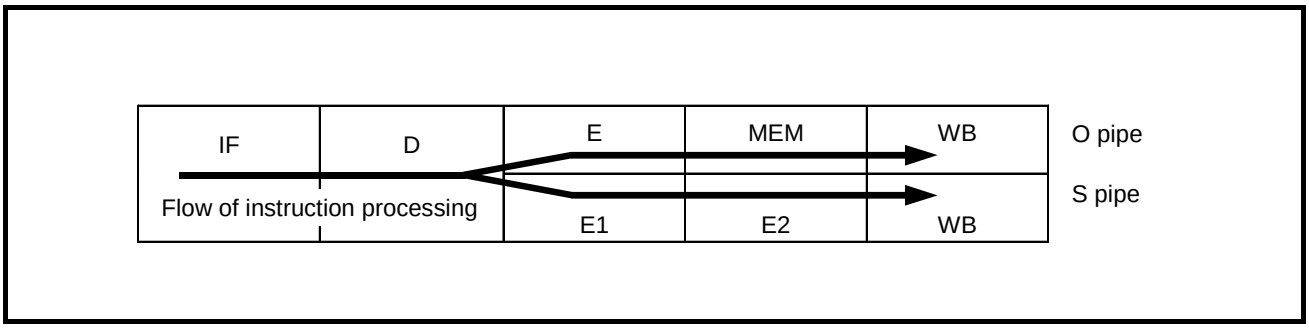
Appendix Figure 1.2 Structure of the S Pipeline and the Operation Performed in It

Appendix 1.2 Flow of Instruction Processing in the O and S Pipes

The IF and D stages in the O and S pipes are shared between the two pipelines, so that all instructions are processed in common at up to the D stage. Which pipeline each particular instruction should be forwarded to, is determined in the D stage.

Instructions to be executed in parallel are forwarded to both the O and S pipes. Other instructions are forwarded to either the O or the S pipe and processed separately at the E and subsequent stages.

A flow of instruction processing in the O and S pipes is shown below.

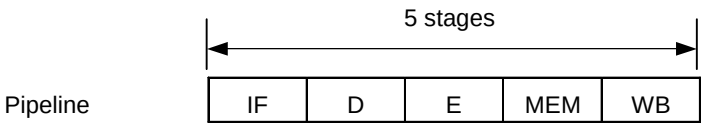


Appendix Figure 1.3 Flow of Instruction Processing in the O and S Pipes

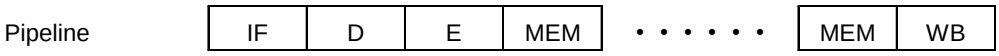
Appendix 1.3 Instructions and Pipelined Processing

The pipelines incorporated in the OPSP CPU each consist of five stages. Since the MEM stage is used for only load/store instructions, and the E2 stage is used for only DSP function instructions that write data to accumulators, all other instructions are processed at five pipeline stages.

■ For load/store instructions

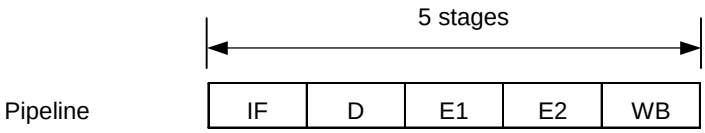


* If memory access is performed with zero wait states, the MEM stage is executed in one cycle. Otherwise, the MEM stage is executed in two or more cycles.

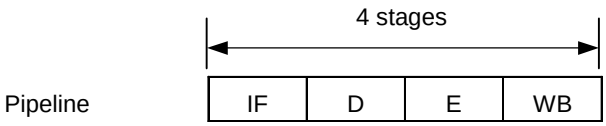


■ For DSP function instructions

(Instructions that write data to the accumulator)



■ For other instructions



* For multi-cycle instructions such as multiply or divide instructions, the E stage is executed in two or more cycles.

Appendix Figure 1.4 Instructions and Pipelined Processing

Appendix 1.4 Pipelined Processing of Parallel Instructions

The OPSP CPU uses two pipelines (O and S pipes) at the same time to accomplish parallel instruction processing.

A pair of 16-bit instructions to be executed in parallel are forwarded through pipeline stages at the same time from the IF to the D stage. After being forwarded to the E stage, they are processed independently in the O and S pipes. Example pipeline operations for parallel instruction execution are shown below.

<Case 1> Parallel execution of a left-side instruction (O–) and a right-side instruction (–S)

LD	**R1,@R2**	IF	D	E	MEM	WB	O pipe
MULHI	**R3,R4**	IF	D	E1	E2	WB	S pipe

<Case 2> Parallel execution of a left-side instruction (O–) and a both-side instruction (OS)

LD	**R1,@R2**	IF	D	E	MEM	WB	O pipe
ADD	**R3,R4**	IF	D	E1	WB		S pipe

<Case 3> Parallel execution of a both-side instruction (OS) and a right-side instruction (–S)

ADD	**R1,R2**	IF	D	E	WB		O pipe
MULHI	**R3,R4**	IF	D	E1	E2	WB	S pipe

<Case 4> Parallel execution of a both-side instruction (OS) and another both-side instruction (OS)

ADD	**R1,R2**	IF	D	E	WB		O pipe
ADD	**R3,R4**	IF	D	E1	WB		S pipe

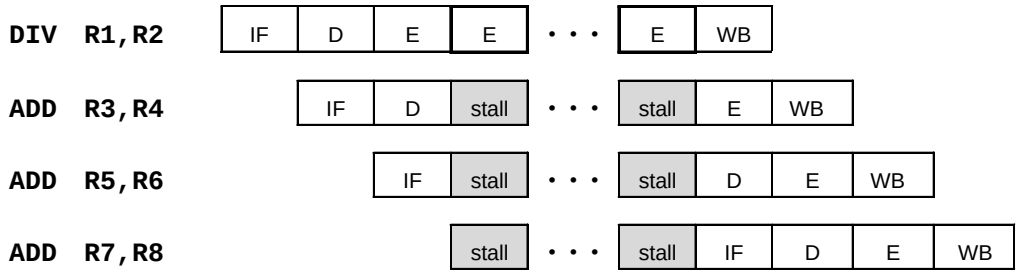
Appendix Figure 1.5 Pipelined Processing of Parallel Instructions

Appendix 1.5 Basic Pipeline Operation

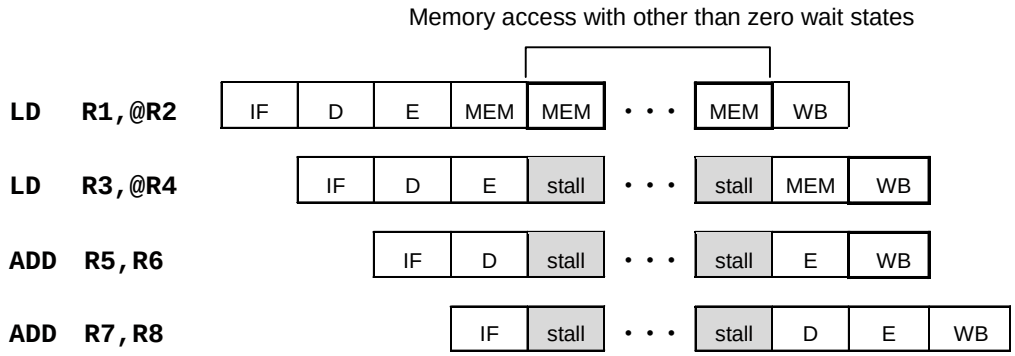
In ideal pipelined instruction processing, it can be expected that each stage is executed in one cycle. However, pipeline operation may be disturbed by processing at a particular stage or by execution of a branch instruction.

The following shows basic pipeline operation for several typical cases.

<Case 1>: When executing an instruction that requires two or more cycles for execution at the E stage



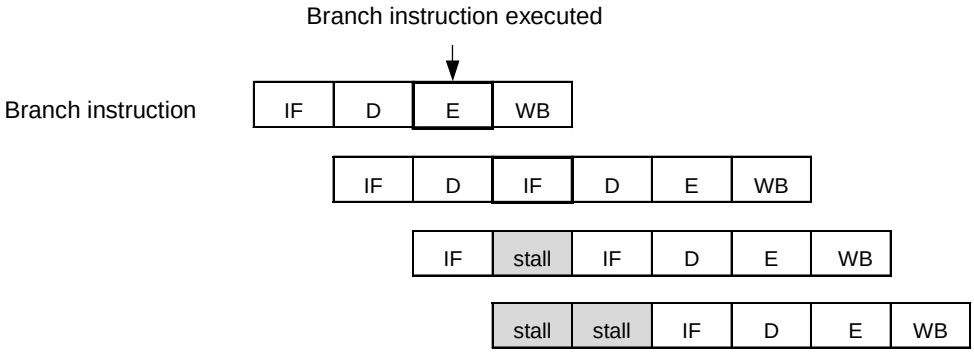
<Case 2>: When operand access cannot be finished in one cycle



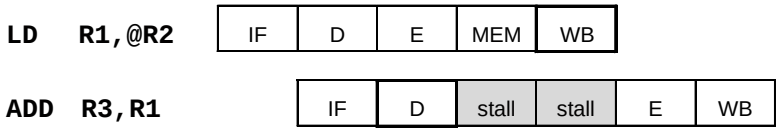
stall : Pipeline stall

Appendix Figure 1.6 Cases where Pipeline Operation is Disturbed - 1

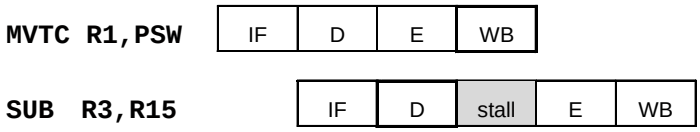
<Case 3>: When executing a branch instruction
(except for conditional branch instructions that did not cause control to branch off)



<Case 4>: Where the subsequent instruction uses the operand read from memory



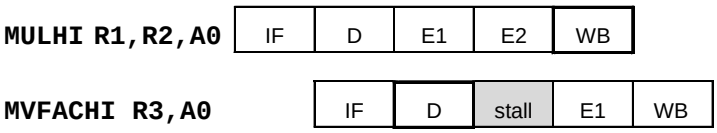
<Case 5>: Where after writing to the PSW register SM bit in MVTC instruction,the subsequent instruction reads out R15



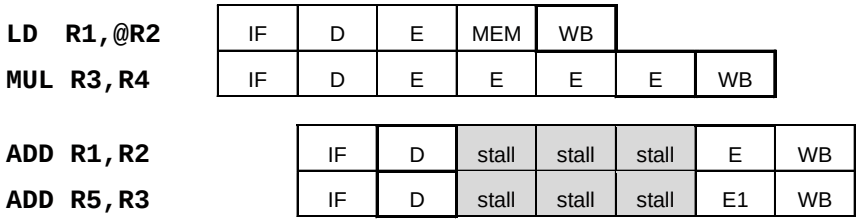
stall : Pipeline stall

Appendix Figure 1.7 Cases where Pipeline Operation is Disturbed - 2

<Case 6>: Where after executing an instruction that writes data to an accumulator
(e.g., DSP function instruction of MULHI), the same accumulator is read out in MVFAC instruction



<Case 7>: Where case 1 and case 4 occur at the same time in parallel instruction processing



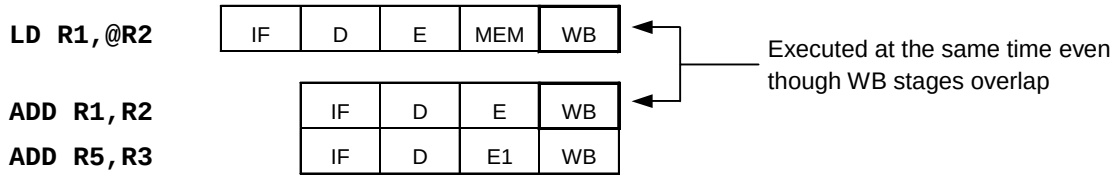
stall : Pipeline stall

Appendix Figure 1.8 Cases where Pipeline Operation is Disturbed - 3

Shown below are special cases where the pipeline operation is not disturbed.

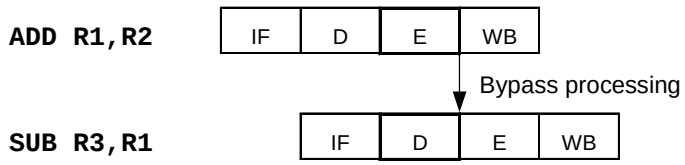
■ Where the WB stage of a load instruction and that of a non-load instruction overlap

(Pipeline processing is not disturbed because writes to registers or accumulators can be performed at the same time.)



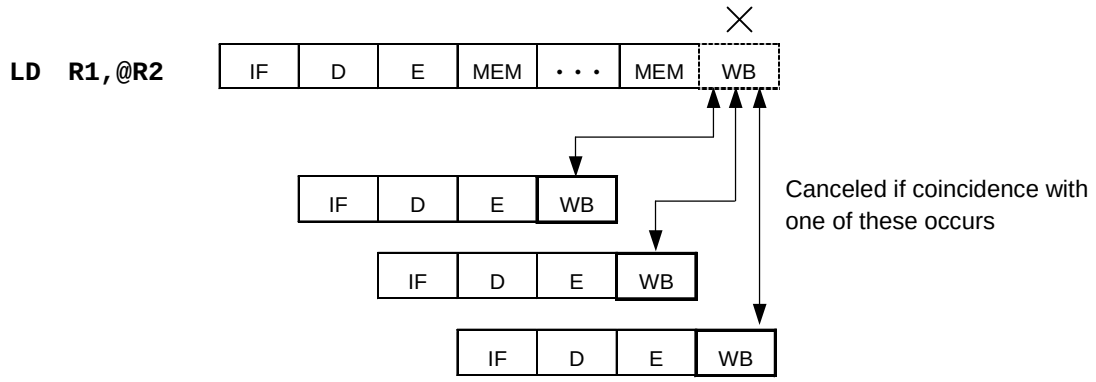
■ Where the subsequent instruction uses the register that was written to by the preceding instruction

(For register-to-register operations, pipeline processing is protected from becoming disturbed by a bypass mechanism.)



■ Where before a load instruction finishes, the subsequent instruction writes to the same register.

(Load instruction execution at the WB stage is canceled.)



Appendix Figure 1.9 Special Cases where Pipeline Operation is Not Disturbed

Appendix 2 Instruction Processing Time

The instruction processing time of the OPSP CPU normally is represented by the number of instruction execution cycles at the E stage. Depending on pipeline operation, however, this instruction processing time may be affected by instruction execution at other stages.

The following shows instruction processing time at each pipeline stage of the OPSP CPU.

Instruction	Number of Execution Cycles at Each Stage				
	IF	D	E Note3	MEM	WB
Load instructions (LD,LDB,LDUB,LDH,LDUH,LOCK)	R Note2	1	1	R Note2	1
Store instructions (ST,STB,STH,UNLOCK)	R Note2	1	1	W Note2	-
BSET and BCLR instructions	R Note2	1	1	R+W Note2	-
Multiplication instructions (MUL)	R Note2	1	4	-	1
Division/remainder instructions (DIVB,DIVUB,REMB,REMUB)	R Note2	1	13	-	1
Division/remainder instructions (DIVH,DIVUH,REMH,REMUH)	R Note2	1	21	-	1
Division/remainder instructions (DIV,DIVU,REM,REMU)	R Note2	1	37	-	1
DSP function instructions [1] (When writing data to accumulators)	R Note2	1	2 (1) Note4	-	1
DSP function instructions [2] (When not writing data to accumulators) Note1	R Note2	1	1	-	1
Instructions other than the above (including DSP function instructions and BTST, SETPSW and CLRPSW instructions)	R Note2	1	1	-	1

Note 1: The DSP function instructions that do not write data to accumulators include MVFACHI, MVFACLO, MVFACMI, SATB and SATH.

Note 2: The number of execution cycles indicated by R and W depends on the type of microcomputer used.

Note 3: For the instructions executed in the O pipe, this indicates the number of execution cycles at the E stage. For the instructions executed in the S pipe, this indicates a total number of execution cycles at both E1 and E2 stages.

Note 4: Although DSP function instructions [1] require two cycles for execution at the E stage, the actual throughput is one because of pipelined instruction processing.

RENESAS 32-BIT OPEN PLATFORM SYNTHESIZABLE PROCESSOR
Software Manual
OPSP

Publication Data : Rev.1.00 Mar 01,2004

Published by : Sales Strategic Planning Div.
Renesas Technology Corp.

OPSP
Software Manual

Renesas Technology Corp.
2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan