MF1097-02

EPSON

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER **E0C33 Family**

E0C33000 Core CPU Manual

ENERGY SAVING EPSON

SEIKO EPSON CORPORATION

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER **E0C33 Family**

E0C33000 CORE CPU MANUAL

This manual explains the functions and instructions of the E0C33000 32-bit RISC CPU which is used as the core of the E0C33 Family 32-bit single chip microcomputers.

Refer to the "Technical Manual " of each E0C33 Family model for details of the hardware including the on-chip peripheral circuits.

Conventions

This manual describes data sizes and numbers as follows:

Data size

8 bits:	Byte, B
16 bits:	Half word, H
32 bits:	Word, W

Numbers

Hexadecimal numbers:	0x0000000, 0xFF etc.
Binary numbers:	0b0000, 0b1111 etc.

Others are decimal numbers. However, "0b" may be omitted if the number can be distinguished as a binary number.

Instructions

Description of the instructions and examples uses small letters (a to z). Capital letters can be used for actual descriptions. See Section 4.1, "Symbol Meanings", for symbols used as operands of the instructions and used in the function descriptions.

CONTENTS

CHAPTER 1 OUTLINE

The E0C33000 is a Seiko Epson original 32-bit RISC-type core CPU for the E0C33 Family microprocessors. This CPU was developed for high-performance embedded applications such as peripheral equipment for personal computers, portable equipment and other products which need high-speed data processing with low power consumption.

The E0C33000 employs pipeline processing and load-store architecture that attains a MIPS value exceeding the operating frequency. The instruction set is optimized for developing in C language, and it is possible to generate compact object codes with the C compiler. Furthermore, the E0C33000 can implement a multiplier and has a multiplication and accumulation instruction (MAC) as an option, it makes it possible to realize on-chip DSP functions.

The E0C33 Family microcomputers consist of the E0C33000 as the core and on-chip peripheral circuits such as ROM, RAM and other high-performance circuits. The E0C33000 core CPU and E0C33 Family microprocessors can realize most user demand functions in one chip.

1.1 Features

CPU type:

- Seiko Epson original 32-bit RISC CPU
- 32-bit internal data processing

Operating frequency:

- DC to 33 MHz (differs depending on the E0C33 Family model)

Instruction set:

- Code size: 16 bits per instruction (fixed)
- Number of instructions: 105 instructions are available.
- Principal instructions can be executed in one cycle.
- An immediate extending instruction is available for immediate extension of instruction codes up to 32 bits.

Multiplication and accumulation instruction (option):

- 64-bit multiplication and accumulation operation (MAC instruction) is available. (16 bits × 16 bits + 64 bits)

Register set:

- Sixteen 32-bit general-purpose registers
- Three 32-bit special registers
- Two 32-bit arithmetic operation registers for multiplier (option)

Memory space and external bus:

- A linear space including code, data and I/O areas.
- A maximum 256MB (28 bits) memory space is accessible.
- Supports 8 and 16-bit external devices.
- Can output 19 area select signals that allow to not expand any glue logic circuit.
- DRAM and other types of memories can be driven directly (differs depending on the E0C33 Family model).
- Harvard architecture
- Little endian format

Interrupts:

- Supports Reset, NMI and 128 external interrupts.
- Four software exceptions and two execution error exceptions.
- The CPU can directly branch the program flow to the trap handler routine by reading the vector from the trap table.

Reset:

- Cold reset (for resetting all conditions)
- Hot reset (reset except for bus and port status)

Power down mode:

- Halt mode (core CPU stops)
- Sleep mode (core CPU and high-speed oscillation circuit stop)

1.2 Block Diagram

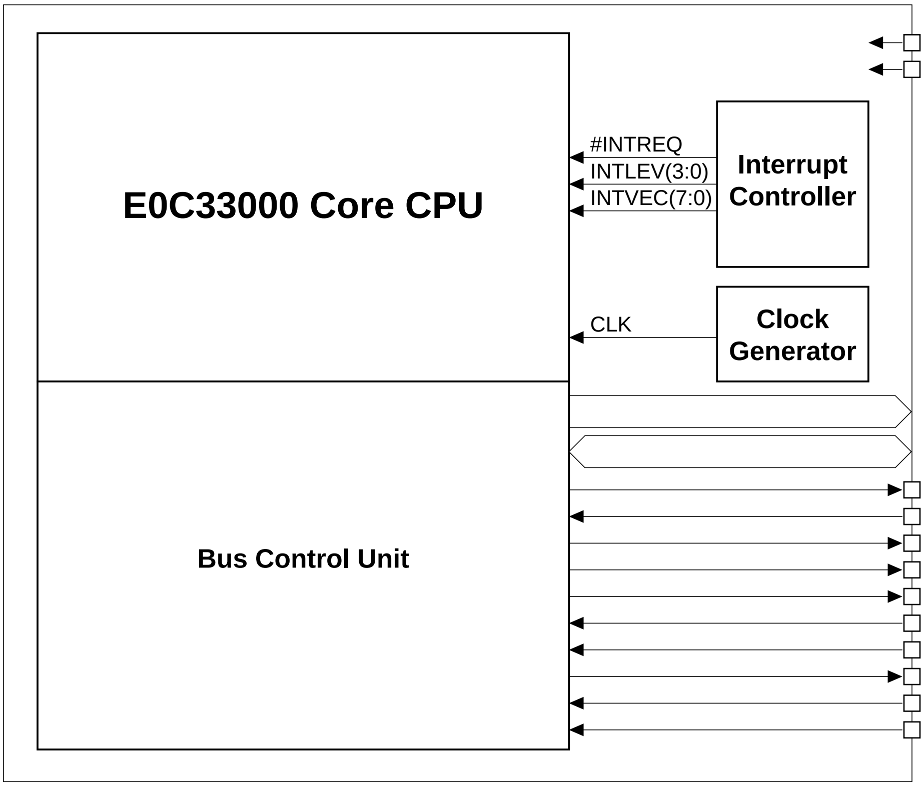


Fig. 1.2.1 E0C33000 block diagram

The diagram is an overview only for principal blocks and signals, it does not indicate the actual circuit configuration.
The actual E0C33 Family processors consist of the above blocks as the main unit and on-chip peripheral circuits.

1.3 I/O Signal Specification

Table 1.3.1 lists the principal input/output signals related to the operation of the E0C33000 core.

Table 1.3.1 E0C33000 I/O signals

Signal name	I/O	Description
VDD	I	Power supply + (supply voltage is different depending on the model)
VSS	I	Power supply - (GND)
CLK (Internal signal)	I	Input clock (clock frequency is different depending on the model)
BCLK	O	Bus clock A bus cycle clock is output.
D(15:0)	I/O	Data bus D[15:0] is a 16-bit bidirectional data bus.
A(27:0)	O	Address bus A[27:0] is a 28-bit address bus.
#WAIT	I	Wait cycle request signal This signal is output from low-speed devices to the CPU. The CPU extends the current bus cycle while this signal is active and waits until the device finishes the bus operation.
#RD	O	Read signal This signal is output when the CPU reads data from the data bus. The selected device outputs data to the data bus while this signal is active.
#WRL #WRH	O	Write signals This signal is output when the CPU writes data to the device connected to the data bus. The selected device inputs data from the data bus while this signal is active. #WRL is the low-order byte write signal and #WRH is the high-order byte write signal. The E0C33000 also supports bus strobe signals (#WR/#BSL/#BSH).
#CE(18:4)	O	Chip enable signals These are chip select signals corresponding to each of the 19 memory areas and are assigned when the CPU accesses the device of each area.
#RESET	I	Initial reset signal The CPU is reset when this signal goes low level. #RESET=0 & #NMI=1: Cold reset #RESET=0 & #NMI=0: Hot reset

BTA3	I	Boot address setting signal
		Specifies a boot address.
		BTA3=1: Booting from internal ROM (Area 3).
		BTA3=0: Booting from external ROM (Area 10).

#NMI	I	NMI request signal This is the non-maskable interrupt request signal. This signal puts the CPU in trap processing status. The signal is also used for specifying the initial reset condition.
#INTREQ (Internal signal)	I	Interrupt request signal This is the maskable interrupt request signal from external devices to the CPU. Usually, the on-chip interrupt controller outputs this signal in the E0C33 Family microprocessors. When this signal is assigned and interrupt conditions are met, the CPU goes into trap processing status.
INTLEV(3:0) (Internal signal)	I	Interrupt level The interrupt level of the peripheral circuit that has requested the interrupt is input. The contents of the signals are set to the IL field in the processor status register (PSR) when the CPU accepts the interrupt. After that, interrupts that have lower levels than the set level are disabled.
INTVEC(7:0) (Internal signal)	I	Interrupt vector number The vector number of the peripheral circuit that has requested the interrupt is input. The CPU reads the specified vector from the trap table to branch the program to the interrupt service routine when the CPU accepts the interrupt.
#BUSREQ	I	Bus request signal This is the bus request signal output from the external bus master devices.
#BUSACK	O	Bus acknowledge signal Indicates that the CPU has accepted the bus request by the external bus master. The CPU changes the bus status in high-impedance to release the bus to the external bus master while this signal is active. The bus control returns to the CPU when the external bus master finishes the bus operation and negates the #BUSREQ signal.

# prefixed the signal names indicate that the signal is low active.

Refer to the "Technical Manual" of each E0C33 Family model for the actual input/output signals and terminals.

CHAPTER 2 ARCHITECTURE

2.1 Register Set

The E0C33000 has sixteen 32-bit general-purpose registers and five 32-bit special registers.

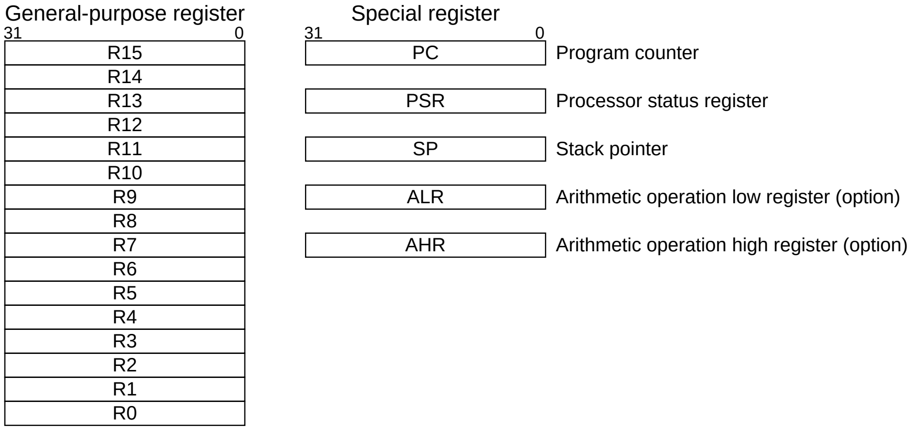


Fig. 2.1.1 Register set

2.1.1 General-purpose registers (R0 to R15)

16 registers R0 to R15 are 32-bit general-purpose registers that can be used for any purpose, such as data operations, data transfers and addressing memories. The register data is always handled as a 32-bit data or an address. Data less than 32 bits is sign-expanded or zero-expanded when it is loaded to the register. When using register data as an address, the high-order 4 bits are invalidated because the address bus is 28 bit size. However, effective address size differs depending on the memory configuration of each model. The general-purpose registers must be initialized before using if necessary, because the register data is undefined at initial reset.

2.1.2 Program counter (PC)

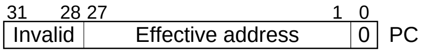


Fig. 2.1.2.1 PC

The program counter (hereinafter described as the PC) is a 32-bit counter that maintains the address of the instruction being executed. In the E0C33000 instruction set, all instructions are 16-bit fixed size. Therefore, the LSB (bit 0) of the PC is always fixed at 0. Furthermore, high-order 4 bits are invalidated because the address bus is 28-bit size. However, effective address size differs depending on the memory configuration of each model.
Programs cannot directly access the PC. Only the following cases change the PC.

(1) At initial reset

Initial reset loads the boot address to the PC and the program starts executing from the address. The boot address is stored in either 0x0080000 in the internal ROM or 0x0C00000 in the external ROM according to the BTA3 terminal setting.

(2) When an instruction is executed

The PC is incremented (+2) every time the CPU executes an instruction and always indicates the address being executed.

(3) When program branches

When the program branches the process flow such as a jump, subroutine call/return or trap processing for interrupts and exceptions, the CPU loads the destination address to the PC.
In subroutine calls and trap processing that need a return operation, the contents of the PC are saved in the stack and it returns to the PC when the return instruction is executed.

2.1.3 Processor status register (PSR)

The processor status register (hereinafter described as the PSR) is a 32-bit register that indicates the CPU status and the content changes according to the instruction executed. It can be read and written using the load instruction.

Since the PSR also affects program execution, when an interrupt or exception occurs, the contents of the PSR are saved into the stack before branching to the handler routine. The saved contents return to the PSR when the return (reti) instruction is executed.

At initial reset, each bit in the PSR is set to 0.

The following shows the function of each bit.

31	30		13	12	11	10	9	8	7	6	5	4	3	2	1	0
–	–	· · ·	–	–	IL(3:0)				MO	DS	–	IE	C	V	Z	N

Fig. 2.1.3.1 Processor status register

"-" indicates unused bit. Writing operation is invalid and 0 is always read.

N (bit 0): Negative flag

Indicates a sign: positive or negative. When a logic operation, arithmetic operation or a shift instruction is executed, the MSB (bit 31) of the result (loaded in the destination register) is copied to the N flag. When a step division is executed, the sign bit of the divisor is copied to the N flag and it affects the division.

Z (bit 1): Zero flag

Indicates that the operation result is zero. The Z flag is set to 1 when the operation result (loaded in the destination register) of a logic operation, arithmetic operation or a shift instruction is zero, and is reset to 0 when the result is not zero.

V (bit 2): Overflow flag

Indicates that an overflow or underflow has occurred. The V flag is set to 1 when an overflow or underflow occurs due to an execution of an addition or subtraction instruction that handles the values as signed 32-bit integers. It is reset to 0 when the addition/subtraction result is within the signed 32-bit data range. The following shows the conditions that set the V flag:

(1) The sign bit (MSB) of the result is 0 (positive) when a negative integer is added to a negative integer.

(2) The sign bit (MSB) of the result is 1 (negative) when a positive integer is added to a positive integer.

(3) The sign bit (MSB) of the result is 1 (negative) when a negative integer is subtracted from a positive integer.

(4) The sign bit (MSB) of the result is 0 (positive) when a positive integer is subtracted from a negative integer.

C (bit 3): Carry flag

Indicates a carry or a borrow. The C flag is set to 1 when the execution result of an addition or subtraction instruction that handles the values as unsigned 32-bit integers exceeds the unsigned 32-bit data range. It is reset to 0 when the addition/subtraction result is within the unsigned 32-bit data range. The following shows the conditions that set the V flag:

(1) When an addition instruction is executed as the result will be bigger than the unsigned 32-bit maximum value 0xFFFFFFFF.

(2) When a subtraction instruction is executed as the result will be smaller than the unsigned 32-bit maximum value 0x00000000.

IE (bit 4): Interrupt enable bit

Enables or disables accepting maskable external interrupts. When the IE bit is set to 1, the CPU can accept maskable external interrupts and when it is reset to 0 it cannot.

See Section 3.3.8, "Maskable external interrupts", for details of the IE bit.

DS (bit 6): Dividend sign flag

The step division copies the sign bit of the dividend to the DS flag. The DS flag affects the division.

MO (bit 7): MAC (Multiply and accumulate) overflow flag

Indicates that an overflow has occurred due to a multiply and accumulate operation. The MO flag is set to 1 when the temporary result of the multiply and accumulate (mac) operation exceeds the effective range of the signed 64-bit data. The operation continues at the last stage regardless of the overflow, therefore the MO flag should be read after the operation has finished to decide whether the result is valid or not. When the MO flag is set to 1, it is maintained until the MO flag is reset by program or initial reset.

IL (bit 8 to bit 11): Interrupt level

Indicates the acceptable interrupt level of the CPU. Maskable external interrupt requests are accepted only when the interrupt level is higher than the level set in the IL field. Furthermore, when an interrupt is accepted, the IL field is set to the accepted interrupt level. After that, interrupts that have the same or lower levels than the IL field are disabled until the program changes the IL field or the interrupt handler routine is terminated with the "reti" instruction.

2.1.4 Stack pointer

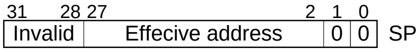


Fig. 2.1.4.1 SP

The stack pointer (hereinafter described as the SP) is a 32-bit register that maintains the stack beginning address.

The stack is an area allocable anywhere in the RAM and is extended toward to the low address from the address initially set in the SP according to the data number saved (pushed). When writing (pushing) data into the stack, the SP is decremented (-4; word units) before writing data to reserve the word area for the data. When getting (popping) data from the stack, word data is retrieved from the address specified by the SP, and then the SP is incremented (+4) to release the word area.

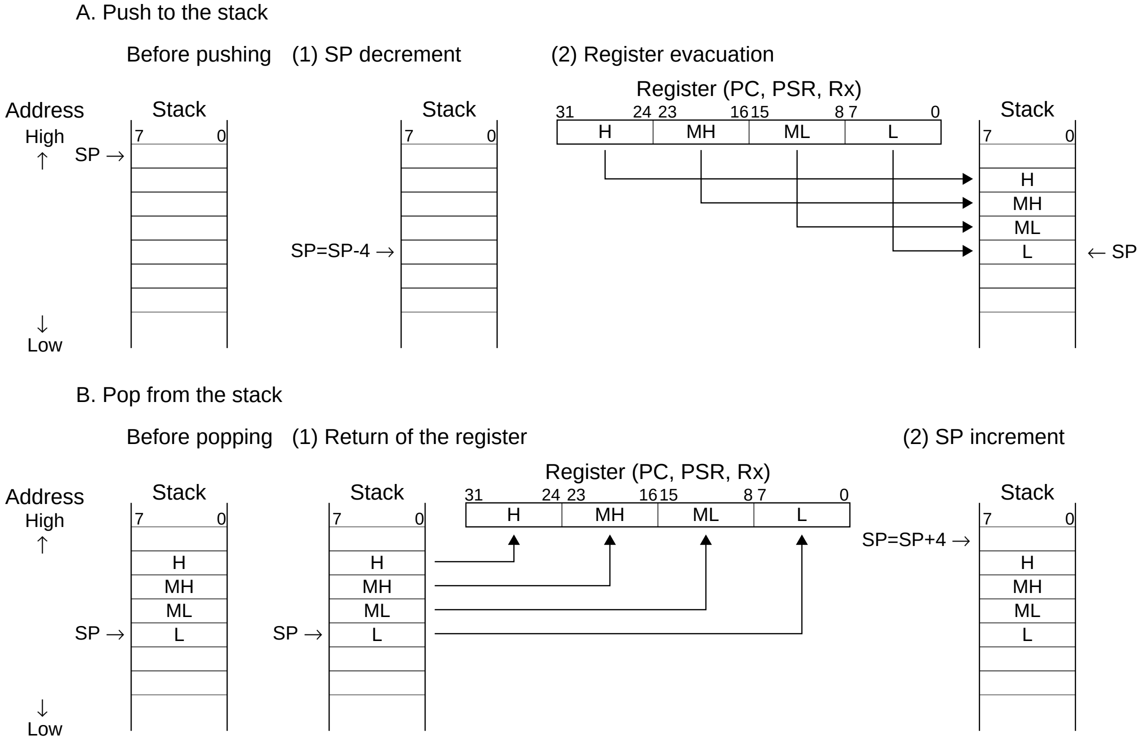


Fig. 2.1.4.2 SP and stack

Data that is pushed into the stack is only 32-bit internal register data, therefore the low-order 2 bits of the SP is fixed at 0 indicating a word boundary. Furthermore the high-order 4 bits are invalidated because the address bus is 28-bit size. However, effective address size differs depending on the memory configuration of each model.
Data push and pop from/to the stack is done in the following cases:

(1) When the call instruction is executed

"call" is the subroutine call instruction and uses 1 word from the stack area. The "call" instruction pushes the contents of the PC (return address; the next address of "call") into the stack before branching. The pushed address is loaded to the PC by the "ret" (return) instruction at the end of the subroutine and the program execution returns to the routine that called the subroutine.

(2) When an interrupt or exception occurs

When a trap such as an interrupt and software exception by the "int" instruction occurs, the CPU pushes the contents of the PC and the PSR into the stack before branching to the handler routine. This is because the trap processing changes these registers. The PC and PSR data is pushed into the stack as shown in Figure 2.1.4.3.
The "reti" instruction that returns the PC and PSR data should be used for return from handler routines.

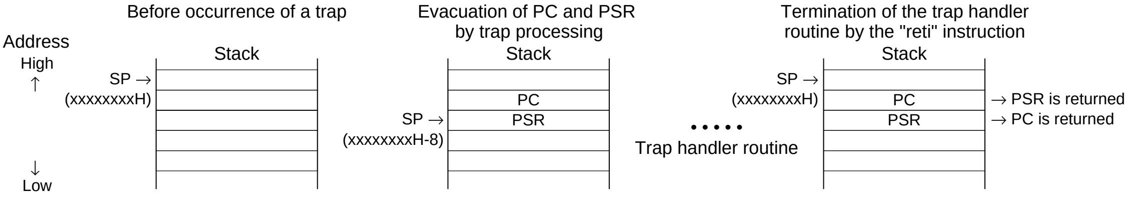


Fig. 2.1.4.3 Stack operation when an interrupt or exception occurs

(3) When the "pushn" or "popn" instruction is executed

The "pushn" instruction saves the contents of R0 to the specified general-purpose register. The "popn" instruction returns the saved data to each register.

The stack area size is restricted according to the RAM size and the area used for storing general data. Pay attention that both areas are not duplicated.
The SP is undefined at initial reset, therefore write an address (stack end address +4; low-order 2 bits are 0) at the head of the initial routine. The stack address can be written using the load instruction. When an interrupt or an exception occurs before setting the stack, the PC and PSR are saved to an undefined location. It cannot guarantee proper operation. Consequently, NMI that cannot be controlled by software is masked by the hardware until the SP is initialized.

2.1.5 Arithmetic operation register (ALR, AHR)

The arithmetic operation low register (hereinafter described as the ALR) and arithmetic operation high register (AHR) in the special registers are used for multiplication, division and multiplication and accumulation operations. These are 32-bit data registers and data can be transferred from/to general-purpose registers using the load instructions.
The multiplication instruction and the multiplication and accumulation instruction place the low-order 32 bits of the result to the ALR and the high-order 32 bits to the AHR.
The division instruction places the quotient to the ALR and the remainder to the AHR.
At initial reset, the ALR and AHR are undefined.
The ALR and the AHR can be used only in the models that have a built-in multiplier.

2.1.6 Register notation and register number

The following shows register notation and register numbers used in the E0C33000 instruction set. Register specification uses a 4-bit field in the instruction code. The specified register number is set in the field. In the mnemonics, "%" must be prefixed to register names.

(1) General-purpose registers

%rs rs is the metasymbol indicating a general-purpose register that contains source data for operation or transfer. Actually describe as %r0 to %r15.

%rd rd is the metasymbol indicating a general-purpose register used as destination (operated or data loaded). Actually describe as %r0 to %r15.

%rb rb is the metasymbol indicating a general-purpose register that contains the base address of the memory to be accessed. In this case, the register works as an index register.
Actually, enclose the register name to be specified with [ ] that indicate register indirect addressing like [%r0] to [%r15]. The E0C33000 allows a register indirect addressing with post increment function for sequential memory accessing. When using this function, postfix "+" like [%r0]+ to [%r15]+. In this case, the base address in the specified register is incremented according to the accessed data size after the memory has been accessed.
rb is also used in the "call" and "jp" instructions and indicates a register that contains a destination address for branching. In this case, [ ] are not necessary, just describe as %r0 to %r15.

The register number of the general-purpose registers is the same as the number in the register name. 0 to 15 (0b0000–0b1111) enters in the register bit field of the instruction code according to the register to be specified.

(2) Special registers

%ss ss is the metasymbol indicating a special register that contains source data to be transferred to a general-purpose register. This symbol is used only in the "ld.w %rd, %ss" instruction.

%sd sd is the metasymbol indicating a special register in which data is loaded from a general-purpose register. This symbol is used only in the "ld.w %sd, %rs" instruction.

Table 2.1.6.1 shows the special register number and the actual notation.

Table 2.1.6.1 Special register number and notation

Special register name	Register number	Notation
Processor status register	0	%psr
Stack pointer	1	%sp
Arithmetic operation low register	2	%alr
Arithmetic operation high register	3	%ahr

0b00 enters in the high-order 2 bits of the register bit field and a register number 0–3 (0b00–0b11) enters in the low-order 2 bits.

2.2 Data Type

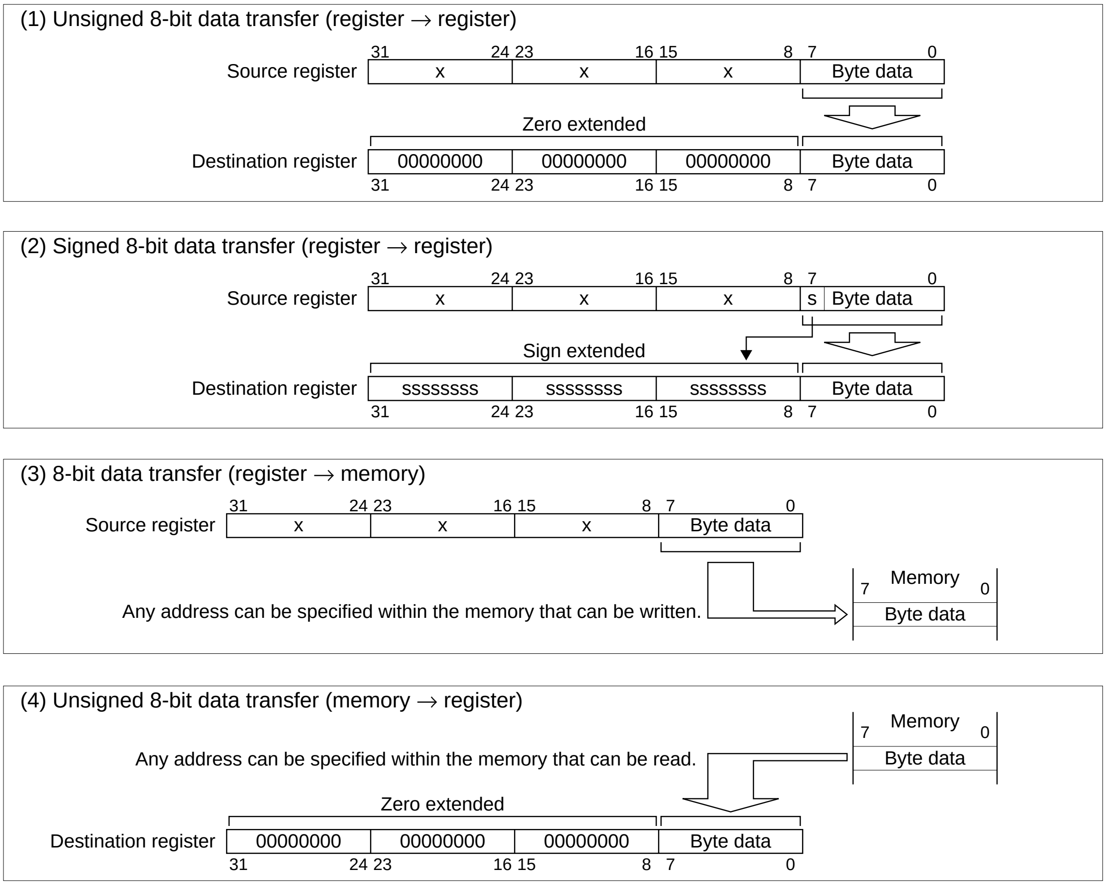
The E0C33000 can handle 8-bit, 16-bit and 32-bit data.
This manual describes each data size as follows:
8-bit data: **Byte** or **B**
16-bit data: **Half word** or **H**
32-bit data: **Word** or **W**

Note that some other manuals describe 16-bit data as Word and 32-bit data as Long word.

Data size can be selected only in data transfers (using a load instruction) between memory and a general-purpose register and between general-purpose registers.
Processing in the CPU core is performed in 32 bits. Consequently, in 16-bit data transfer and 8-bit data transfer to a general-purpose register, the transfer data is sign-extended or zero-extended into 32 bits when it is loaded to the register. The extension type, sign or zero, is decided according to the load instruction to be used.
In 16-bit data transfer or 8-bit data transfer from a general-purpose register, the low-order half word or the low-order byte is transferred, respectively.

Memory is accessed in byte, half word or word units with the little endian method. The address to be specified must be a half word boundary address (MSB is 0) for half word data accessing, and a word boundary address (low-order 2 bits are 0) for word data accessing, otherwise an address error exception will occur.

Figure 2.2.1 shows the types of data transfer.



(1) Unsigned 8-bit data transfer (register → register)

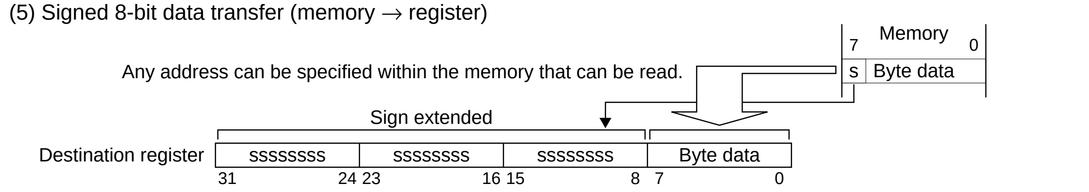
(2) Signed 8-bit data transfer (register → register)

(3) 8-bit data transfer (register → memory)

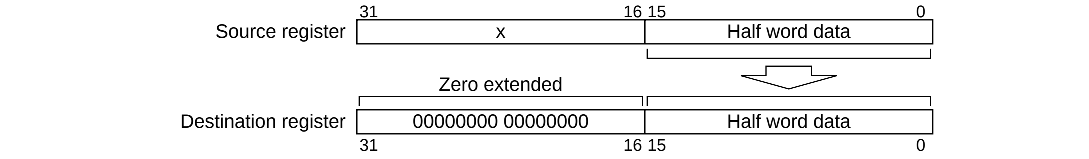
31 24 23 16 15 8 7 0
Source register
x
x
x
Byte data
Any address can be specified within the memory that can be written.
Memory
7 0
Byte data

(4) Unsigned 8-bit data transfer (memory → register)

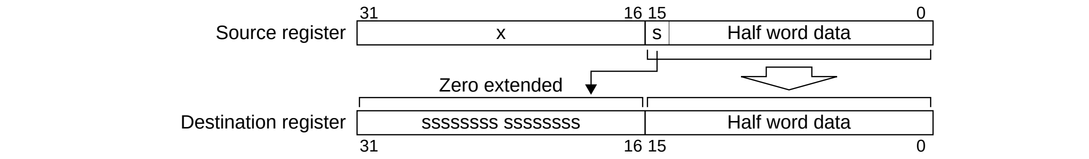
(5) Signed 8-bit data transfer (memory → register)



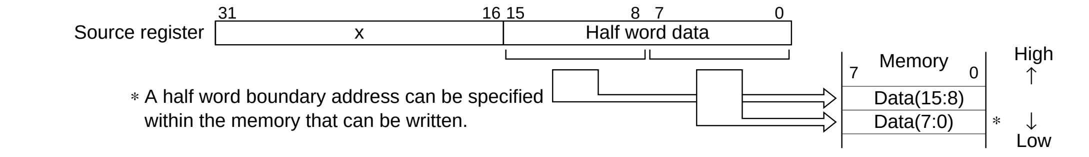
(6) Unsigned 16-bit data transfer (register → register)



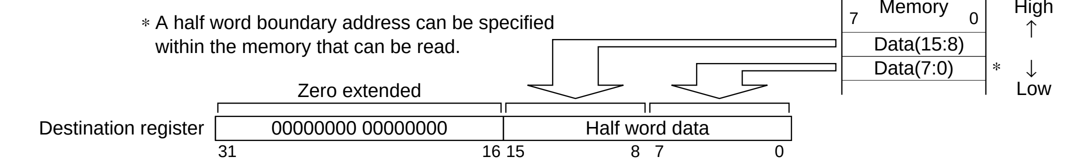
(7) Signed 16-bit data transfer (register → register)



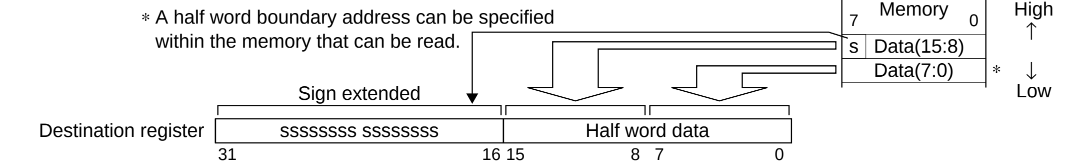
(8) 16-bit data transfer (register → memory)


31
16 15
8 7
0
Source register
x
Half word data
Memory
7
0
High
↑
∗ A half word boundary address can be specified within the memory that can be written.
Data(15:8)
Data(7:0)
∗
↓
Low

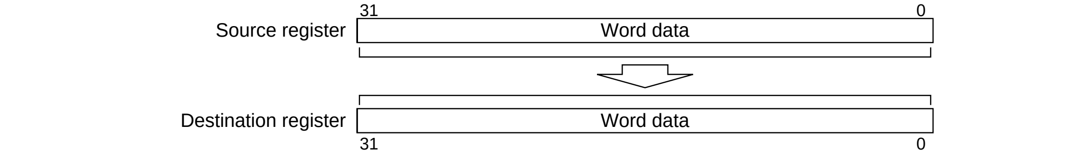
(9) Unsigned 16-bit data transfer (memory → register)



(10) Signed 16-bit data transfer (memory → register)



(11) 32-bit data transfer (register → register)



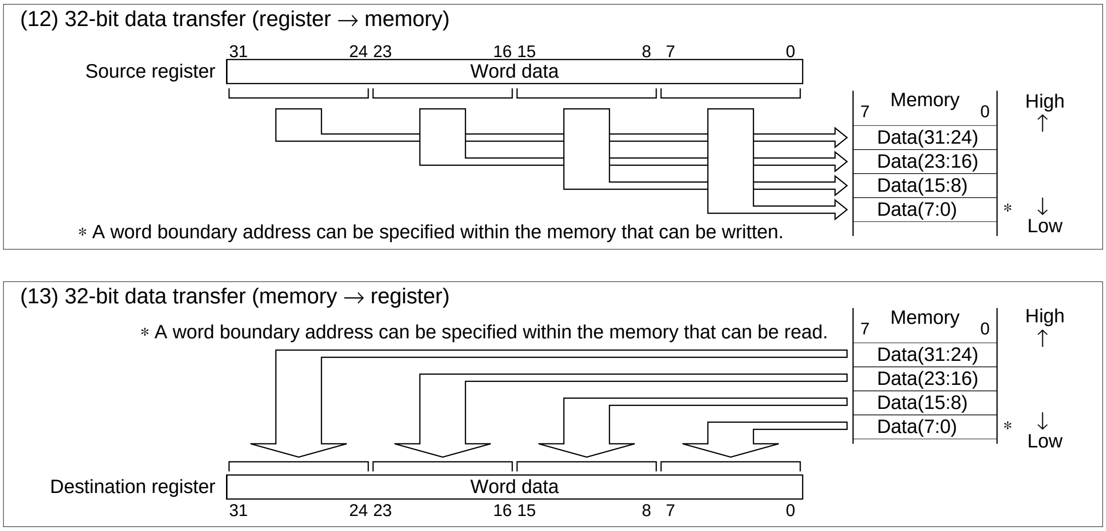

(12) 32-bit data transfer (register → memory)
31
24 23
16 15
8 7
0
Source register
Word data
Memory
7
0
High
↑
Data(31:24)
Data(23:16)
Data(15:8)
Data(7:0)
∗
↓
Low
∗ A word boundary address can be specified within the memory that can be written.
(13) 32-bit data transfer (memory → register)
∗ A word boundary address can be specified within the memory that can be read.
Memory
7
0
High
↑
Data(31:24)
Data(23:16)
Data(15:8)
Data(7:0)
∗
↓
Low
Destination register
Word data
31
24 23
16 15
8 7
0

Fig. 2.2.1 Data transfer type

2.3 Address Space

The E0C33000 has a 28-bit (256MB) address space.
Memories are all allocated within the space. Furthermore the E0C33000 employs a memory mapped I/O method, thus control registers of I/O modules are also allocated in this space and they can be accessed as well as general memories.
Figure 2.3.1 shows the basic memory map.

Area No.	Address		Area size
Area 18	0xFFFFFFF 0xC000000	External memory	64MB
Area 17	0xBFFFFFF 0x8000000	External memory	64MB
Area 16	0x7FFFFFF 0x6000000	External memory	32MB
Area 15	0x5FFFFFF 0x4000000	External memory	32MB
Area 14	0x3FFFFFF 0x3000000	External memory	16MB
Area 13	0x2FFFFFF 0x2000000	External memory	16MB
Area 12	0x1FFFFFF 0x1800000	External memory	8MB
Area 11	0x17FFFFF 0x1000000	External memory	8MB
Area 10	0x0FFFFFF 0x0C00000	External memory	4MB
Area 9	0x0BFFFFF 0x0800000	External memory	4MB
Area 8	0x07FFFFF 0x0600000	External memory	2MB
Area 7	0x05FFFFF 0x0400000	External memory	2MB
Area 6	0x03FFFFF 0x0300000	External I/O	1MB
Area 5	0x02FFFFF 0x0200000	External memory	1MB
Area 4	0x01FFFFF 0x0100000	External memory	1MB
Area 3	0x00FFFFF 0x0080000	Internal ROM	512KB
Area 2	0x007FFFF 0x0060000	Reserved area for ICE	128KB
Area 1	0x005FFFF 0x0040000	Internal peripheral circuit	128KB
Area 0	0x003FFFF 0x0000000	Internal RAM	256KB

Fig. 2.3.1 Memory map

As shown in the figure, the E0C33000 manages the address space by dividing it into 19 areas. The type of modules that can be connected are predefined in each area. Area 0 is for the internal RAM in the E0C33 Family, Area 1 is for internal peripheral circuits and Area 3 is for the internal ROM.
Area 10 can be used as an external ROM area including a boot address.
Area 2 is an internal area, but do not use it because Area 2 is reserved for ICE software (See Section 3.6, "Debugging Mode").

Each area for external modules can specify the device type to be used, data size and number of wait cycles. The specifiable items differ depending on the E0C33 Family model.

The E0C33000 has a built-in address decoder, it makes it possible to output 19 select signals corresponding to the 19 areas. Thus the system that follows the basic memory map does not need any external glue logic, and external devices can be directly connected.

The internal memory capacity, I/O memory size and address bus size differ depending on the E0C33 Family model. Therefore, the memory map shown in Figure 2.3.1 does not apply to all models. Refer to the "Technical Manual" of each model for the actual memory map.

2.4 Boot Address

In the E0C33000, the trap table location can be selected from either Area 3 (internal ROM) or Area 10 (external ROM) by the BTA3 terminal setting. The trap table begins from the head of the area and the reset vector for booting is placed at the head of the table, so the boot address is placed at the beginning address of the selected area.

Table 2.4.1 Boot address setting

Terminal level	Area selected	Boot address
BTA3=1 (High)	Area 3 (internal ROM)	0x0080000
BTA3=0 (Low)	Area 10 (external ROM)	0x0C00000

General models of the E0C33 Family have a built-in ROM and can boot from both areas.
Models that have no built-in ROM can only boot from the external ROM.
Refer to the "Technical Manual" of each model for boot address settings.

2.5 Instruction Set

The E0C33000 instruction set contains 61 basic instructions (105 instructions in all). The instruction codes are all fixed at the 16-bit size. The CPU can execute the principal instructions in 1 cycle with pipeline processing and load-store type architecture. The instruction set has an optimized code system that can generate compact object codes even if developing in C language.
This section explains the function overview of the E0C33000 instruction set.
See Chapter 4, "Detailed Explanation of Instructions", for details of each instruction.

2.5.1 Type of instructions

Table 2.5.1.1 lists the instructions.

Table 2.5.1.1 Instruction list

Classification	Mnemonic		Function	
Logic operation	and	%rd, %rs	AND between general-purpose registers	
		%rd, sign6	AND between general-purpose register and immediate data (with sign extension)	
	or	%rd, %rs	OR between general-purpose registers	
		%rd, sign6	OR between general-purpose register and immediate data (with sign extension)	
	xor	%rd, %rs	XOR between general-purpose registers	
		%rd, sign6	XOR between general-purpose register and immediate data (with sign extension)	
	not	%rd, %rs	NOT for general-purpose registers	
		%rd, sign6	NOT for immediate data (with sign extension)	
Arithmetic operation	add	%rd, %rs	Addition between general-purpose registers	
		%rd, imm6	Addition of immediate data to general-purpose registers (with zero extension)	
		%sp, imm10	Addition of immediate data to SP (with zero extension)	
	adc	%rd, %rs	Addition with carry between general-purpose registers	
	sub	%rd, %rs	Subtraction between general-purpose registers	
		%rd, imm6	Subtraction of immediate data from general-purpose register (with zero extension)	
		%sp, imm10	Subtraction of immediate data from SP (with zero extension)	
	sbc	%rd, %rs	Subtraction with borrow between general-purpose registers	
	cmp	%rd, %rs	Comparison between general-purpose registers	
		%rd, sign6	Comparison between general-purpose register and immediate data (with sign extension)	
	mlt.h	%rd, %rs	Multiplication for signed integers (16 bits × 16 bits = 32 bits)	<option>
	mltu.h	%rd, %rs	Multiplication for unsigned integers (16 bits × 16 bits = 32 bits)	<option>
	mlt.w	%rd, %rs	Multiplication for signed integers (32 bits × 32 bits = 64 bits)	<option>
	mltu.w	%rd, %rs	Multiplication for unsigned integers (32 bits × 32 bits = 64 bits)	<option>
	div0s	%rs	Signed division 1st step	<option>
	div0u	%rs	Unsigned division 1st step	<option>
	div1	%rs	Step division execution	<option>
	div2s	%rs	Data correction 1 for signed division result	<option>
	div3s		Data correction 2 for signed division result	<option>
Shift & Rotate	srl	%rd, %rs	Logical shift to right (shift count is specified with register)	
		%rd, imm4	Logical shift to right (shift count is specified with immediate data)	
	sll	%rd, %rs	Logical shift to left (shift count is specified with register)	
		%rd, imm4	Logical shift to left (shift count is specified with immediate data)	
	sra	%rd, %rs	Arithmetic shift to right (shift count is specified with register)	
		%rd, imm4	Arithmetic shift to right (shift count is specified with immediate data)	
	sla	%rd, %rs	Arithmetic shift to left (shift count is specified with register)	
		%rd, imm4	Arithmetic shift to left (shift count is specified with immediate data)	
	rr	%rd, %rs	Rotation to right (shift count is specified with register)	
		%rd, imm4	Rotation to right (shift count is specified with immediate data)	
	rl	%rd, %rs	Rotation to left (shift count is specified with register)	
		%rd, imm4	Rotation to left (shift count is specified with immediate data)	
Branch	jrgt jrgt.d	sign8	PC relative conditional jump; Branch condition: !Z & !(N ^ V) (".d" allows delayed branch.)	
	jrge jrge.d	sign8	PC relative conditional jump; Branch condition: !(N ^ V) (".d" allows delayed branch.)	
	jrlt jrlt.d	sign8	PC relative conditional jump; Branch condition: N ^ V (".d" allows delayed branch.)	
	jrle jrle.d	sign8	PC relative conditional jump; Branch condition: Z | N ^ V (".d" allows delayed branch.)	
	jrugt jrugt.d	sign8	PC relative conditional jump; Branch condition: !Z & !C (".d" allows delayed branch.)	
	jruge jruge.d	sign8	PC relative conditional jump; Branch condition: !C (".d" allows delayed branch.)	

Classification	Mnemonic		Function
Branch	jrult	sign8	PC relative conditional jump; Branch condition: C
	jrult.d		(".d" allows delayed branch.)
	jrule	sign8	PC relative conditional jump; Branch condition: Z | C
	jrule.d		(".d" allows delayed branch.)
	jreq	sign8	PC relative conditional jump; Branch condition: Z
	jreq.d		(".d" allows delayed branch.)
	jrne	sign8	PC relative conditional jump; Branch condition: !Z
	jrne.d		(".d" allows delayed branch.)
	jp	sign8	PC relative jump (".d" allows delayed branch.)
	jp.d	%rb	Absolute jump (".d" allows delayed branch.)
	call	sign8	PC relative call (".d" allows delayed branch.)
	call.d	%rb	Absolute call (".d" allows delayed branch.)
	ret		Return from subroutine
	ret.d		(".d" allows delayed branch.)
	reti		Return from interrupt/exception handler routine
	retd		Return from debugging routine
	int	imm2	Software exception
	brk		Debugging exception
Data transfer	ld.b	%rd, %rs	General-purpose register (byte) → General-purpose register (with sign extension)
		%rd, [%rb]	Memory (byte) → General-purpose register (with sign extension)
		%rd, [%rb]+	"+" is specification for address post-increment function.
		%rd,[%sp+imm6]	Stack (byte) → General-purpose register (with sign extension)
		[%rb], %rs	General-purpose register (byte) → Memory
		[%rb]+, %rs	"+" is specification for address post-increment function.
		[%sp+imm6],%rs	General-purpose register (byte) → Stack
	ld.ub	%rd, %rs	General-purpose register (byte) → General-purpose register (with zero extension)
		%rd, [%rb]	Memory (byte) → General-purpose register (with zero extension)
		%rd, [%rb]+	"+" is specification for address post-increment function.
		%rd,[%sp+imm6]	Stack (byte) → General-purpose register (with zero extension)
	ld.h	%rd, %rs	General-purpose register (half word) → General-purpose register (with sign extension)
		%rd, [%rb]	Memory (half word) → General-purpose register (with sign extension)
		%rd, [%rb]+	"+" is specification for address post-increment function.
		%rd,[%sp+imm6]	Stack (half word) → General-purpose register (with sign extension)
		[%rb], %rs	General-purpose register (half word) → Memory
		[%rb]+, %rs	"+" is specification for address post-increment function.
		[%sp+imm6],%rs	General-purpose register (half word) → Stack
	ld.uh	%rd, %rs	General-purpose register (half word) → General-purpose register (with zero extension)
		%rd, [%rb]	Memory (half word) → General-purpose register (with zero extension)
		%rd, [%rb]+	"+" is specification for address post-increment function.
		%rd,[%sp+imm6]	Stack (half word) → General-purpose register (with zero extension)
	ld.w	%rd, %rs	General-purpose register (word) → General-purpose register
		%rd, %ss	Special register (word) → General-purpose register
		%sd, %rs	General-purpose register (word) → Special register
		%rd, sign6	Immediate data → General-purpose register (with sign extension)
		%rd, [%rb]	Memory (word) → General-purpose register
		%rd, [%rb]+	"+" is specification for address post-increment function.
		%rd,[%sp+imm6]	Stack (word) → General-purpose register
		[%rb], %rs	General-purpose register (word) → Memory
		[%rb]+, %rs	"+" is specification for address post-increment function.
		[%sp+imm6],%rs	General-purpose register (word) → Stack
System control	nop		No operation
	halt		Sets CPU to HALT mode
	slp		Sets CPU to SLEEP mode
Immediate extension	ext	imm13	Extends the operand (immediate data) of the following instruction.
Bit operation	btst	[%rb], imm3	Tests the specified bit in the memory data (byte)
	bclr	[%rb], imm3	Clears the specified bit in the memory data (byte)
	bset	[%rb], imm3	Sets the specified bit in the memory data (byte)
	bnot	[%rb], imm3	Reverses the specified bit in the memory data (byte)
Others	scan0	%rd, %rs	"0" bit search
	scan1	%rd, %rs	"1" bit search
	swap	%rd, %rs	Swap of the byte data order in word data (upper byte ↔ lower byte)
	mirror	%rd, %rs	Swap of the bit order in each byte of word data (upper bit ↔ lower bit)
	mac	%rs	Multiplication and accumulation (16 bits × 16 bits + 64 bits → 64 bits) <option>
	pushn	%rs	Pushes %rs–%r0 register data into stack.
	popn	%rd	Pops %r0–%rd register data from stack.

2.5.2 Addressing mode

The E0C33000 instruction set has six addressing modes. The CPU accesses data according to the addressing mode specified by the operand in each instruction.

(1) Immediate addressing

This mode uses an immediate data in the instruction code such as immX (unsigned immediate data) and signX (signed immediate data) as the source data. This mode can be used in the logic operation (and, or, xor, not), arithmetic operation (add, sub, cmp), immediate data load ("ld.w %rd, sign6"), shift & rotate (srl, sll, sra, sla, rr, rl), bit operation (btst, bclr, bset, bnot) and immediate extension (ext) instructions.

The number in the immediate symbols indicates the usable immediate data size (e.g. imm4 = unsigned 4-bit data, sign6 = signed 6-bit data).

Immediate data except for shift & rotate operations can be extended using the "ext" instruction (see the next section).

(2) Register direct addressing

This mode uses the contents of the specified register as source data. When a register is specified as the destination of the instruction, the operation result or transfer data is loaded to the register. The instructions that have an operand below are executed in this mode.

%rs rs is the metasymbol indicating a general-purpose register that contains source data for operation or transfer. Actually describe as %r0 to %r15.

%rd rd is the metasymbol indicating a general-purpose register used as destination. Actually describe as %r0 to %r15. It may be used as a source data.

%ss ss is the metasymbol indicating a special register that contains source data to be transferred to a general-purpose register.

%sd sd is the metasymbol indicating a special register in which data is loaded from a general-purpose register.

The special register names should actually be described as follows:

Processor status register	%psr
Stack pointer	%sp
Arithmetic operation low register	%alr
Arithmetic operation high register	%ahr

"%" must be prefixed to the register names in order to distinguish from symbol names.

(3) Register indirect addressing

This mode accesses a memory indirectly using the register that contains an address. It is applied to only the load instructions that have [%rb] as an operand. The register name should be enclosed with [ ] in actual specification as [%r0] to [%r15].

The CPU transfers data in data type according to the load instruction using the contents of the specified register as the base address of the memory to be accessed.

In half word data transfers and word data transfers, the base address to be set in the register must be pointed at a half word boundary (LSB is 0) and a word boundary (low-order 2 bits are 0), respectively. If not, an address error exception will occur.

(4) Register indirect addressing with post-increment

The general-purpose register specifies a memory to be accessed the same as register indirect addressing. When the data transfer has finished, this mode increments the base address in the specified register according to the transferred data size*. Thus continuous reading/writing from/to the memory can be done by setting the beginning address only.

* **Increment size**

Byte transfer (ld.b, ld.ub):	rb←rb+1
Half word transfer (ld.h, ld.uh):	rb←rb+2
Word transfer (ld.w):	rb←rb+4

This mode should be specified by enclosing the register name with [ ] and postfixing "+". Actually describe as [%r0]+ to [%r15]+.

(5) Register indirect addressing with displacement

This mode accesses the memory specified with a register as the base address and an immediate data as the displacement (the displacement is added to the base address). This mode is applied only to the load instructions that have [%sp+imm6] as an operand excluding the case of the "ext" instruction.

Example:

```
ld.b    %r0,[%sp+0x10]   ; Loads the byte data stored in the address that is specified by the
                           contents of the SP + 0x10 to the R0 register. The 6-bit immediate
                           data is directly added as a displacement in the byte data transfer.
ld.h    %r0,[%sp+0x10]   ; Loads the half word data stored from the address that is specified
                           by the contents of the SP + 0x20 to the R0 register. In half word
                           data transfer, the doubled 6-bit immediate data (LSB is always 0)
                           is added as a displacement to specify a half word boundary.
ld.w    %r0,[%sp+0x10]   ; Loads the word data stored from the address that is specified by
                           the contents of the SP + 0x40 to the R0 register. In word data
                           transfer, the quadrupled 6-bit immediate data (low-order 2 bits
                           are always 0) is added as a displacement to specify a word
                           boundary.
```

The "ext" instruction (explained in the next section) changes the following register indirect addressing instruction ([%rb]) to this mode using the immediate data specified in the "ext" instruction as the displacement.

Example:

```
ext     imm13
ld.b    %rd,[%rb]        ; Functions as "ld.b  %rd, [%rb+imm13]".
```

(6) Signed PC relative addressing

This mode is applied to the branch instructions (jr*, jp, call) that have a signed 8-bit immediate data (sign8) as the operand. Those instructions branch the program flow to the address specified by the current PC + sign8 × 2.

The displacement (sign8) can be extended using the "ext" instruction (see the next section).

2.5.3 Immediate extension (EXT) instruction

All the instruction codes are 16-bit size, so it limits the immediate size included in the code. The "ext" instruction is mainly used to extend the immediate size.
The "ext" instruction should be described prior to the target instruction (to extend the immediate data). The "ext" instruction can specify a 13-bit immediate data and up to two "ext" instructions can be used at a time for more extension. The "ext" instruction is valid only if the instruction that follows the "ext" instruction can be extended. It is invalid for all other instructions. If three or more "ext" instructions are described consecutively, only the two instructions at the first and the last (prior to the target instruction) are validated. The middle "ext" instructions are ignored.
The following shows the functions of the "ext" instruction.

Note: Examples of the "ext" instruction use imm13 for the immediate data of the first "ext" instruction and imm13' for the second "ext" instruction.

(1) Immediate extension in immediate addressing instructions

- **Extension of imm6**

Target instructions: "add %rd, imm6", "sub %rd, imm6"
The above instructions can use a 6-bit immediate data by itself.
The immediate data can be extended into 19-bit size or 32-bit size by describing the "ext" instruction prior to these instructions.

<u>When one "ext" instruction is used:</u>

```
ext     imm13
add     %rd,imm6        ; Executed as "add  %rd, imm19".
```

The "ext" instruction extends the imm6 (6 bits) into imm19 (19 bits). The imm13 in the "ext" instruction becomes the high-order 13 bits of the imm19. The imm19 is zero-extended into 32 bits and operation to the rd register is done in 32-bit size.

<u>When two "ext" instructions are used:</u>

```
ext     imm13
ext     imm13'
sub     %rd,imm6        ; Executed as "sub  %rd, imm32".
```

The "ext" instructions extend the imm6 (6 bits) into imm32 (32 bits). The imm32 is configured in the order of imm13, imm13' and imm6 from the high-order side.

- **Extension of sign6**

Target instructions: "and %rd, sign6", "or %rd, sign6", "xor %rd, sign6", "not %rd, sign6",
"cmp %rd, sign6", "ld.w %rd, sign6"
The above instructions can use a signed 6-bit immediate data by itself.
The immediate data can be extended into signed 19 bits or signed 32 bits by describing the "ext" instruction prior to these instructions.

<u>When one "ext" instruction is used:</u>

```
ext     imm13
and     %rd,sign6       ; Executed as "and  %rd, sign19".
```

The "ext" instruction extends the sign6 (signed 6-bit data) into sign19 (signed 19-bit data). The imm13 in the "ext" instruction becomes the high-order 13 bits of the sign19. The sign19 is sign-extended into 32 bits using the MSB as the sign bit (0=+, 1=-) and operation to the rd register is done in signed 32-bit size.

<u>When two "ext" instructions are used:</u>

```
ext     imm13
ext     imm13'
cmp     %rd,sign6       ; Executed as "cmp  %rd, sign32".
```

The "ext" instructions extend the imm6 (signed 6-bit data) into sign32 (signed 32-bit data). The sign32 is configured in the order of imm13, imm13' and sign6 from the high-order side. The MSB of the 1st sign13 becomes the sign bit of the sign32.

(2) Displacement extension in register indirect addressing

• Adding a displacement to [%rb]

Target instructions: ld. %rd, [%rb]" (ld.*: ld.b, ld.ub, ld.h, ld.uh, ld.w), "ld.* [%rb], %rs" (ld.*: ld.b, ld.h, ld.w), "btst [%rb], imm3", "bclr [%rb], imm3", "bset [%rb], imm3", "bnot [%rb], imm3"*

The above instructions access memories in register indirect addressing mode using the contents of the rb register as the base address.
The addressing mode changes into register indirect addressing with displacement by describing the "ext" instruction prior to these instructions.

When one "ext" instruction is used:

```
ext     imm13
ld.b    %rd,[%rb]         ; Executed as "ld.b  %rd, [%rb+imm13]".
```

The extended instruction accesses the memory specified by adding the 13-bit displacement (imm13) to the base address stored in the rb register. The imm13 is zero-extended at the address operation.

When two "ext" instructions are used:

```
ext     imm13
ext     imm13'
btst    [%rd],imm3        ; Executed as "btst  [%rb+imm26], imm3".
```

The extended instruction accesses the memory specified by adding the 26-bit displacement (imm26) to the base address stored in the rb register. The imm26 is configured in the order of imm13 and imm13' from the high-order side. The imm26 is zero-extended at the address operation.

This extension is not applied to the instructions for register indirect addressing with post increment ([%rb]+).

• Extending the displacement of [%sp+imm6]

Target instructions: "ld. %rd, [%sp+imm6]" (ld.*: ld.b, ld.ub, ld.h, ld.uh, ld.w)*
"ld. [%sp+imm6], %rs" (ld.*: ld.b, ld.h, ld.w)*

The above instructions access memories in register indirect addressing with displacement using the contents of the rb register as the base address and the immediate data (imm6) in the code as the 6-bit, 7-bit or 8-bit displacement.

Byte data transfer (ld.b, ld.ub):	6-bit displacement = imm6 = {imm6}
Half word data transfer (ld.h, ld.uh):	7-bit displacement = imm6 × 2 = {imm6, 0}
Word data transfer (ld.w):	8-bit displacement = imm6 × 4 = {imm6, 00}

The displacement size can be extended into 19 bits or 32 bits by describing the "ext" instruction prior to these instructions.

When one "ext" instruction is used:

```
ext     imm13
ld.b    %rd,[%sp+imm6]; Executed as "ld.b  %rd, [%sp+imm19]".
```

The extended instruction accesses the memory specified by adding the 19-bit displacement (imm19) to the stack beginning address stored in the SP. The imm13 in the "ext" instruction is placed at the high-order 13 bits of the imm19 and the imm6 in the load instruction is used for the low-order 6 bits. However in half word data transfer and word data transfer, the imm6 is used as below to prevent the occurrence of an address error exception.

Byte data transfer (ld.b, ld.ub):	imm19 = {imm13, imm6)
Half word data transfer (ld.h, ld.uh):	imm19 = {imm13, imm6(5:1), 0}
Word data transfer (ld.w):	imm19 = {1mm13, imm6(5:2), 00}

The imm19 is zero-extended at the address operation.

When two "ext" instructions are used:

```
ext     imm13
ext     imm13'
ld.w    [%sp+imm6],%rs; Executed as "ld.w  [%sp+imm32], %rs".
```

The extended instruction accesses the memory specified by adding the 32-bit displacement (imm32) to the stack beginning address stored in the SP. The imm32 is configured in the order of imm13, imm13' and imm6 from the high-order side. However in half word data transfer and word data transfer, the imm6 is used as below to prevent the occurrence of an address error exception.

Byte data transfer (ld.b, ld.ub):	imm32 = {imm13, imm13', imm6)
Half word data transfer (ld.h, ld.uh):	imm32 = {imm13, imm13', imm6(5:1), 0}
Word data transfer (ld.w):	imm32 = {1mm13, imm13', imm6(5:2), 00}

The imm32 is handled as an unsigned 32-bit data for the address operation. If the value after adding the displacement exceeds the effective address range (28 bits max.), the exceeded part is invalidated.

(3) Extending the instructions between registers operation into 3 operands instruction

Target instructions: "add %rd, %rs", "sub %rd, %rs", "cmp %rd, %rs", "and %rd, %rs", "or %rd, %rs", "xor %rd, %rs"

The above instructions operate with the contents of the rd and rs registers, and then stores the results into the rd register.

When the "ext" instruction is described prior to the instructions, they operate with the rs register and the immediate data in the "ext" instruction and then the results are stored into the rd register. The contents of the rd register do not affect the operation.

When one "ext" instruction is used:

```
ext     imm13
add     %rd,%rs          ; Executed as "rd ← rs + imm13".
```

The imm13 is zero-extended into 32 bits because the operation is performed in 32-bit size.

When two "ext" instructions are used:

```
ext     imm13
ext     imm13'
sub     %rd,%rs          ; Executed as "rd ← rs - imm26".
```

The imm26 is configured in order of imm13 and imm13' from the high-order side.

The imm26 is zero-extended into 32 bits because the operation is performed in 32-bit size.

(4) Displacement extension for the PC relative branch instructions

The PC relative branch instructions that have a sign8 (signed 8-bit immediate data) as the operand branch the program flow to the address specified by the current PC address + doubled sign8 (9-bit displacement). The "ext" instruction extends the displacement into 22 bits (when one "ext" is used) or 32 bits (when two "ext" are used). See Section 2.5.12, "Branch instructions and delayed instructions" for more information.

2.5.4 Data transfer instructions

The E0C33000 instruction set supports data transfers between registers and between a register and memory. Transfer data size and data extension type can be specified by the instruction code. The classifications on the mnemonic notation are as follows:

ld.b	Signed byte data transfer
ld.ub	Unsigned byte data transfer
ld.h	Signed half word data transfer
ld.uh	Unsigned half word data transfer
ld.w	Word data transfer

In a signed byte/half word transfer to a register, the source data is sign-extended into 32 bits. In an unsigned byte/half word transfer, the source data is zero-extended into 32 bits.
In a data transfer that specifies a register as the source, the specified size of low-order bits in the register is transferred.

2.5.5 Logic operation instructions

Four types of logic operation instructions are available in the E0C33000 instruction set.

and	Logical product
or	Logical sum
xor	Exclusive OR
not	Negation

All the logic operations use a general-purpose register (R0–R15) as the destination. Two types of sources can be used: 32-bit data in a general-purpose register or signed immediate data (6, 19 or 32 bits).

2.5.6 Arithmetic operation instructions

The E0C33000 instruction set supports addition, subtraction, comparison, multiplication and division for arithmetic operation (see the next section for the multiplication/division instructions).

add	Addition
adc	Addition with carry
sub	Subtraction
sbc	Subtraction with borrow
cmp	Comparison

The arithmetic operations are performed between general-purpose registers (R0–R15) or between a general-purpose register and an immediate data. Furthermore the "add" and "sub" instructions supports an operation between the SP and an immediate data. The immediate data other than word size is zero-extended at the operation excluding the "cmp" instruction.
The "cmp" instruction compares two operands and sets/resets the flags according to the comparison results. Generally it is used to set a condition for the conditional jump instruction. When an immediate data other than word size is specified for the source, it is sign-extended at comparison.

2.5.7 Multiplication and division instructions

Multiplication and division functions have been implemented in the E0C33000 instruction set. However, they can be used only in the models which have a built-in multiplier by option. Refer to the "Technical Manual" of each model for confirming whether the model has the multiplier or not.

(1) Multiplication instructions

The E0C33000 instruction set has contained four multiplication instructions.

mlt.h	16 bits × 16 bits → 32 bits (signed multiplication)
mltu.h	16 bits × 16 bits → 32 bits (unsigned multiplication)
mlt.w	32 bits × 32 bits → 64 bits (signed multiplication)
mltu.w	32 bits × 32 bits → 64 bits (unsigned multiplication)

These instructions use data in the specified general-purpose registers (R0–R15) for the multiplier and the multiplicand. In 16-bit multiplication, the low-order 16 bits in the specified registers are used. The signed multiplication instructions handle the MSBs of the multiplier and multiplicand as the sign bits. 16 bits × 16 bits of multiplication stores the result into the ALR. 32 bits × 32 bits of multiplication stores the high-order 32 bits of the result into the AHR and the low-order 32 bits into the ALR.
The E0C33000 executes a 16 bits × 16 bits multiplication in one cycle and a 32 bits × 32 bits in five cycles.

(2) Division instructions

The signed and unsigned step division functions have been implemented in the E0C33000.

Instructions used for signed step divisions:	div0s, div1, div2s, div3s
Instructions used for unsigned step divisions:	div0u, div1

The following shows the executing procedure and functions of the step division:

1 Pre-process of the step division (div0s, div0u)

Prepare a dividend in the ALR and a divisor in an rs register (general-purpose register R0–R15) before starting a step division, then execute the "div0s" (for signed division) or "div0u" (for unsigned division) instruction.
These instructions operate as follows:

<u>div0s (pre-process for signed step division)</u>

- Extends the dividend in the ALR into 64 bits with a sign and sets it in {AHR, ALR}.
  When the dividend is a positive number, the AHR is set to 0x00000000.
  When the dividend is a negative number, the AHR is set to 0xFFFFFFFF.
- Sets the sign bit of the dividend (MSB of ALR) to the DS flag in the PSR.
  When the dividend is a positive number, the DS flag is reset to 0.
  When the dividend is a negative number, the DS flag is reset to 1.
- Sets the sign bit of the divisor (MSB of the rs register) to the N flag in the PSR.
  When the divisor is a positive number, the N flag is reset to 0.
  When the divisor is a negative number, the N flag is reset to 1.

<u>div0u (pre-process for unsigned step division)</u>

- Clears the AHR to 0x00000000.
- Resets the DS flag in the PSR to 0.
- Resets the N flag in the PSR to 0.

2 Executing the step division

Execute the "div1" instruction for the necessary steps. For example, in 32 bits ÷ 32 bits division, the "div1" instruction should be executed 32 times.
The "div1" instruction is commonly used for signed and unsigned division.
One "div1" instruction step performs the following process:

1) Shifts the 64-bit data (dividend) in {AHR, ALR} 1 bit to the left (to upper side). (ALR(0) = 0)

2) Adds rs to the AHR or subtracts rs from the AHR and modifies the AHR and the ALR according to the results.
The addition/subtraction uses the 33-bit data created by extending the contents of the AHR with the DS flag as the sign bit and the 33-bit data created by extending the contents of the rs register with the N flag as the sign bit.
The process varies according to the DS and N flags in the PSR as shown below. "tmp(32)" in the explanation indicates the bit-33 value of the addition/subtraction results.

In the case of DS = 0 (dividend is positive) and N = 0 (divisor is positive):
2-1) Executes tmp = {0, AHR} - {0, rs}
2-2) If tmp(32) = 1, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 0, terminates without changing the AHR and ALR.

In the case of DS = 1 (dividend is negative) and N = 0 (divisor is positive):
2-1) Executes tmp = {1, AHR} + {0, rs}
2-2) If tmp(32) = 0, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 1, terminates without changing the AHR and ALR.

In the case of DS = 0 (dividend is positive) and N = 1 (divisor is negative):
2-1) Executes tmp = {0, AHR} + {1, rs}
2-2) If tmp(32) = 1, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 0, terminates without changing the AHR and ALR.

In the case of DS = 1 (dividend is negative) and N = 1 (divisor is negative):
2-1) Executes tmp = {1, AHR} - {1, rs}
2-2) If tmp(32) = 0, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 1, terminates without changing the AHR and ALR.

In unsigned division, the results are obtained from the following registers by executing the necessary "div1" instruction steps.

The results of unsigned division: ALR = Quotient, AHR = Remainder

In signed division, the results should be corrected as shown below.

3 Correcting the results of signed division

In signed division, execute the "div2s" and "div3s" instructions sequentially to correct the results after the necessary steps of the "div1" instruction are executed.
Unsigned division does not need to execute the "div2s" and "div3s" instructions. If executed, they function the same as the "nop" instruction and do not affect the operation results.

The following shows the functions of the "div2s" and "div3s" instructions:

div2s (correction stage 1 for the results of signed step division)

When the dividend is a negative number and zero results in a division step (execution of div1), the remainder (AHR) after completing all the steps may be the same as the divisor and the quotient (AHR) may be 1 short from the actual absolute value. The "div2s" instruction corrects such a result.

In the case of DS = 0 (dividend is positive):

This problem does not occur when the dividend is a positive number, so the "div2s" instruction terminates without any execution (same as the "nop" instruction).

In the case of DS = 1 (dividend is negative):

1) If N = 0 (divisor is positive), executes tmp = AHR + rs
   If N = 1 (divisor is negative), executes tmp = AHR - rs

2) According to the result of step 1).
   If tmp is zero, executes AHR = tmp(31:0) and ALR = ALR + 1 and then terminates.
   If tmp is not zero, terminates without changing the AHR and ALR.

<u>div3s (correction stage 2 for the result of signed step division)</u>

Step division always stores a positive number of quotient into the ALR. When the signs of the dividend and divisor are different, the result must be a negative number. The "div3s" instruction corrects the sign in such cases.

In the case of DS = N (dividend and divisor have the same sign):

This problem does not occur, so the "div3s" instruction terminates without any execution (same as the "nop" instruction).

In the case of DS = !N (dividend and divisor have different signs):

Reverses the sign bit of the ALR (quotient).

In signed division, the results are obtained from the following registers after executing the "div2s" and "div3s" instructions.

The results of unsigned division: ALR = Quotient, AHR = Remainder

Execution examples of division

(1) Signed division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w   %alr,%r0    ; Set the dividend to the ALR
div0s  %r1         ; Initialization for signed division
div1   %r1         ; Step division
 :      :
div1   %r1         ; Executing div1 32 times
div2s  %r1         ; Correction 1
div3s              ; Correction 2
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR. This example completes execution in 36 cycles.

In signed division, the remainder has the same sign as the dividend.

Examples: (-8) ÷ 5 = -1 remainder = -3
          8 ÷ (-5) = -1 remainder = 3

(2) Unsigned division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w   %alr,%r0    ; Set the dividend to the ALR
div0u  %r1         ; Initialization for signed division
div1   %r1         ; Step division
 :      :
div1   %r1         ; Executing div1 32 times
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR. This example completes execution in 34 cycles.

2.5.8 Multiplication and accumulation instruction

The E0C33000 supports a multiplication and accumulation function that executes "64 bits + 16 bits × 16 bits" the specified number of times. This function realizes on-chip digital signal processing without an external DSP chip. However, this function is only available in the models which have a built-in multiplier by option. Refer to the "Technical Manual" of each model for confirming whether the model has the multiplier or not.
The multiplication and accumulation operation is executed by the "mac" instruction.
The "mac %rs" instruction repeats execution of the "{AHR, ALR} ← {AHR, ALR} + H[<rs+1>]+ × H[<rs+2>]+" operation for the count number specified by the rs register.

The repeat count should be set in the rs register before starting multiplication and accumulation operation. The rs register is used as a counter and is decremented by each operation. The "mac" instruction terminates operation when the rs register becomes 0. Thus it is possible to repeat operation up to 2^{32}-1 (4,294,967,295) times. When the "mac" instruction is executed by setting the rs register to 0, the "mac" instruction does not perform a multiplication and accumulation operation and does not change the AHR and the ALR. The rs register is not decremented as it is 0.

<rs+1> and <rs+2> are the general-purpose registers which follow the rs register.
Example: When the R0 register is specified for rs: <rs+1>=R1 register, <rs+2>=R2 register
When the R15 register is specified for rs: <rs+1>=R0 register, <rs+2>=R1 register

H[<rs+1>]+ and H[<rs+2>]+ indicate the half word data stored from the base address specified by the register.
The "mac" instruction multiplies these data as signed 16-bit data, and adds the results to the {AHR, ALR} register pair. "+" indicates that the base address (contents of the <rs+1> and <rs+2> registers) is incremented (+2) every time the operation step is finished.

Example: When the "mac %r0" is executed after setting R0=16, R1=0x100, R2=0x120, AHR=ALR=0:

1) {AHR, ALR} = 0 + H[0x100] × H[0x120]
2) {AHR, ALR} = {AHR, ALR} + H[0x102] × H[0x122]
3) {AHR, ALR} = {AHR, ALR} + H[0x104] × H[0x124]
: :
16) {AHR, ALR} = {AHR, ALR} + H[0x11E] × H[0x13E]

The operation result is obtained as a 64-bit data from the AHR for the high-order 32 bits and the ALR for the low-order 32 bits.
The register values are changed as R0 = 0, R1 = 0x120 and R2 = 0x140.

Overflow during multiplication and accumulation operation

When the temporary result overflows the signed 64-bit range during multiplication and accumulation operation, the MO flag in the PSR is set to 1. However, the operation continues until the repeat count that is set in the rs register goes to 0. Since the MO flag stays 1 until it is reset by software, it is possible to check whether the result is valid or not by reading the MO flag after completing execution of the "mac" instruction.

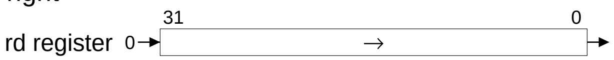
Interrupts during multiplication and accumulation operation

Interrupts are accepted even if the "mac" instruction is executing halfway through the repeat count. The trap processing saves the address of the "mac" instruction into the stack as the return address before branching to the interrupt handler routine. Thus when the interrupt handler routine is finished by the "reti" instruction, the suspended "mac" instruction resumes execution. The content of the rs register at that point is used as the remaining repeat count, therefore if the interrupt handler routine has modified the rs register the "mac" instruction cannot obtain the expected results. Similarly, when the <rs+1> and/or <rs+2> registers have been modified in the interrupt handler routine, the resumed "mac" instruction cannot be executed properly.

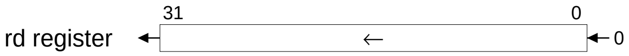
2.5.9 Shift and rotation instructions

The E0C33000 instruction set has shift and rotation instructions for register data.

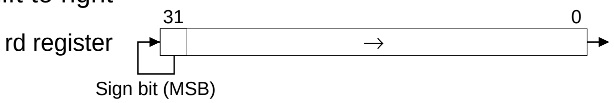
srl Logical shift to right



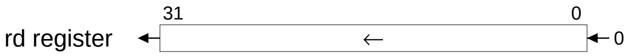
sll Logical shift to left



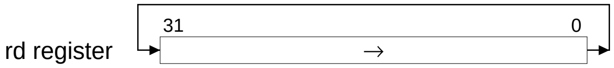
sra Arithmetical shift to right



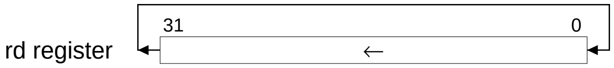
sla Arithmetical shift to left



rr Rotation to right



rl Rotation to left



These instructions shift the contents of the specified general-purpose registers as shown in each figure. The shift count can be specified from 0 to 8 bits using a general-purpose register or an immediate data.

Instruction %rd, %rs Shifts/rotates the content of the rd register by the shift count specified with the rs register.
Bits 0 to 3 of the rs register are effective for the shift count (0 to 8).

Instruction %rd, imm4 Shifts/rotates the content of the rd register by the shift count specified with the unsigned 4-bit immediate data (imm4).

The rs register and imm4 specify the shift count as follows:

rs(3:0)/imm4	Shift count	
1xxx	8 bits	(x: 1 or 0)
0111	7 bits	
0110	6 bits	
0101	5 bits	
0100	4 bits	
0011	3 bits	
0010	2 bits	
0001	1 bit	
0000	0 bit	

2.5.10 Bit operation instructions

The following four instructions are available for handling memory data in bit units. These instructions allow direct modification of display memory bits and I/O control bits.

btst	[%rb], imm3	Sets Z flag if the specified bit is 0.
bclr	[%rb], imm3	Clears the specified bit to 0.
bset	[%rb], imm3	Sets the specified bit to 1.
bnot	[%rb], imm3	Reverses the specified bit (1 ↔ 0).

The bit operation is performed for the memory address specified by the rb (general-purpose) register. The imm3 specifies the bit number (bit 0 to bit 7) of the byte data stored in the address.
These instructions (excluding "btest") change the specified bit only, however, the specified address is rewritten since the memory access is performed in byte units. Therefore, pay attention to the operation of the address that contains an I/O control bit affected by writing.

2.5.11 Push and pop instructions

The push and pop instructions are used to evacuate and return the contents of the general-purpose registers from/to the stack.

Push instruction pushn %rs

Saves the contents of the rs to the R0 registers sequentially into the stack.

Pop instruction popn %rd

Loads the stack data to the R0 to the rd registers sequentially.

Example:

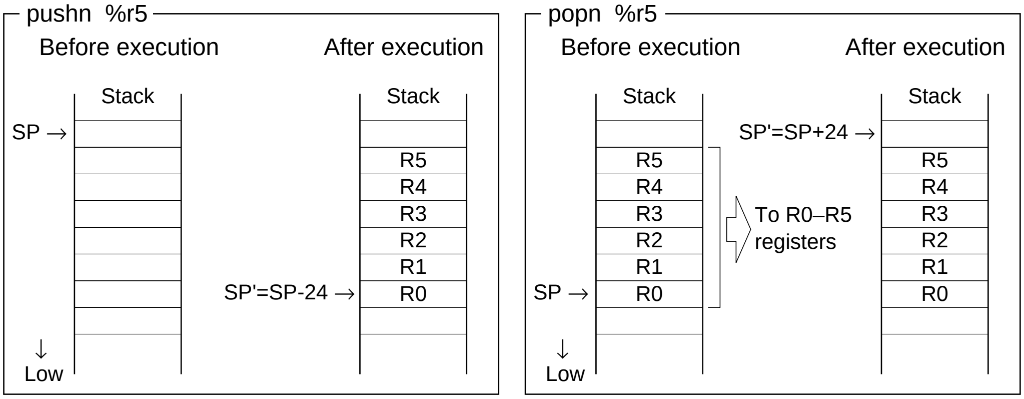


Fig. 2.5.11.1 Evacuation and return of general-purpose registers

The "pushn" and "popn" instructions should be used as a pair that specify the same registers.
These instructions modify the SP according to the register count to be evacuated/returned.

Besides the push and pop instructions, some load instructions that execute in register indirect addressing with displacement mode ([%sp+imm6]) are provided. They can load/store register data individually from/to the stack using the SP as the base address. However in this case, the SP is not modified.

2.5.12 Branch instructions and delayed instructions

Classification of branch instructions

(1) PC relative jump instructions ("jr* sign8", "jp sign8")

The PC relative jump instruction adds the signed displacement in its operand to the current PC address (address of the branch instruction) for branching the program flow to the address. It allows relocatable programming.
Since all the instruction size is fixed at 16 bits, the sign8 specifies a half word address in 16-bit units. Consequently, the displacement that is added to the PC becomes a signed 9-bit data (LSB is always 0) by doubling the sign8, and it always specifies an even address. When the PC value exceeds the 28-bit address space after adding the displacement, the exceeded part (high-order 4 bits) is invalidated.
The displacement can be extended using the "ext" instruction as shown below.

Independent use of the branch instruction:

```
jp      sign8              ; Executed as "jp  sign9". (sign9 = {sign8, 0})
```

When using a branch instruction independently, a signed 8-bit displacement (sign8) can be specified. Since the sign8 is a relative value in 16-bit units, the specifiable branch range is [PC - 256 to PC + 254].

When one "ext" instruction is used:

```
ext     imm13
jp      sign8              ; Executed as "jp  sign22". (sign22 = {imm13, sign8, 0})
```

The sign8 is extended into a sign22 using the imm13 of the "ext" instruction as the high-order 13 bits. The specifiable branch range is [PC - 2,097,152 to PC + 2,097,150].

When two "ext" instructions are used:

```
ext     imm13
ext     imm13'
jp      sign8              ; Executed as "jp  sign32".
```

The imm13 of the first "ext" instruction is used as the high-order 10 bits of the sign32, therefore only 10 bits from Bit 12 to Bit3 are effective (the low-order 3 bits are ignored). The sign32 is configured as follows:

sign32 = {imm13(12:3), imm13', sign8, 0}

The specifiable branch range is [PC - 2,147,483,648 to PC + 2,147,483,646].

The branch ranges above are just a logical value. Actually it is limited to the memory range of the model to be used.

Branch conditions

The "jp" instruction is an unconditional branch instruction that always branches the program.
The instructions that begin with "jr" are conditional branch instructions. Each instruction has a branch condition specified with a combination of the flags, and branches the program flow only when the condition has been met. If not, it does not branch.
Usually the conditional branch instructions are used to judge the results of the "cmp" instruction that compares two values. For this purpose, each instruction name contains the letters that indicate the relation.
Table 2.5.12.1 lists the conditional branch instructions and their conditions.

Table 2.5.12.1 Conditional branch instructions and conditions

Instruction	Flag condition	Result of "cmp A, B"	Remarks
jrgt (Greater Than)	!Z & !(N ^ V)	A > B	for signed data comparison
jrge (Greater or Equal)	!(N ^ V)	A ≥ B	
jrlt (Less Than)	N ^ V	A < B	
jrle (Less or Equal)	Z | (N ^ V)	A ≤ B	
jrugt (Unsigned, Greater Than)	!Z & !C	A > B	for unsigned data comparison
jruge (Unsigned, Greater or Equal)	!C	A ≥ B	
jrult (Unsigned, Less Than)	C	A< B	
jrule (Unsigned, Less or Equal)	Z | C = 1	A ≤ B	
jreq (Equal)	Z	A = B	for signed and unsigned comparison
jrne (Not equal)	!Z	A ≠ B	

The program branches if the logic equation of the flags are true (1). (!: NOT, |: OR, &: AND, ^: XOR)

(2) Absolute jump instruction ("jp %rb")

The absolute jump instruction "jp %rb" unconditionally branches the program flow to the absolute address specified by the rb register.
The LSB of the rb register goes to 0 when the register data is loaded to the PC, and the high-order 4 bits that are out of the address range are also invalidated.

(3) PC relative call instruction ("call sign8")

The PC relative call instruction adds the signed displacement in its operand to the current PC address (address of the branch instruction) to unconditionally branch to the subroutine that begins from the address. It allows relocatable programming.
The address of the following instruction (or address of the second from the call instruction in delayed branch) is saved into the stack as the return address before branching. Executing the "ret" instruction at the end of the subroutine loads the saved address to the PC, and the program returns from the subroutine.

Since all the instruction size is fixed at 16 bits, the sign8 specifies a half word address in 16-bit units. Consequently, the displacement that is added to the PC becomes a signed 9-bit data (LSB is always 0) by doubling the sign8, and it always specifies an even address. When the PC value exceeds the 28-bit address space after adding the displacement, the exceeded part (high-order 4 bits) is invalidated.
The displacement can be extended using the "ext" instruction the same as the PC relative jump instruction. See "PC relative jump instructions" on the previous page for the displacement extension.

(4) Absolute call instruction ("call %rb")

The absolute call instruction "call %rb" unconditionally calls a subroutine that begins from the absolute address specified by the rb register.
The LSB of the rb register goes to 0 when the register data is loaded to the PC, and the high-order 4 bits that are out of the address range are also invalidated.

(5) Software exception ("int imm2")

The software exception instruction "int imm2" issues a software exception to execute the specified trap handler routine. Up to four handler routines can be created and the imm2 specifies the vector number of the handler routine to be executed. When a software exception occurs, the CPU saves the PSR and the address of the instruction that follows the "int" instruction into the stack and then reads the specified vector from the trap table to execute the trap handler routine. Therefore, the "reti" instruction that returns the saved PSR must be used for returning from the trap handler routine.
See Section 3.3, "Trap (Interrupts and Exceptions)", for details of the software exceptions.

(6) Return instructions ("ret", "reti")

The "ret" instruction is the return instruction that corresponds to the "call" instruction. It ends the subroutine by loading the return address saved in the stack to the PC. The SP must contain the same value (that points the return address) as the beginning of the subroutine when the "ret" instruction is executed.

The "reti" instruction is the return instruction for exclusive use of trap handler routines. The trap processing of the CPU saves a return address and the PSR into the stack, therefore the "reti" instruction must be used for returning the contents of the PSR. As well as the "ret" instruction, the SP must contain the same value (that points the return address) as the beginning of the trap handler routine when the "reti" instruction is executed.

(7) Debugging exceptions ("brk", "retd")
The "brk" and "retd" instructions are used for calling a debugging routine and return. Since these instructions are provided for the ICE software, do not use them in the application program.
See Section 3.6, "Debugging Mode", for the functions of these instructions.

Delayed branch function

The E0C33000 executes an instruction and fetches an instruction simultaneously by pipe-line processing. When executing a branch instruction, the following instruction has been fetched by the CPU. By executing the fetched instruction before branching, the execution cycles of the branch instruction can be reduced for 1 cycle. This is the delayed branch function and the following instruction that is executed before branching is called a delayed instruction.
The instructions below can use the delayed branch function. In the mnemonic notation, ".d" should be postfixed to the branch instruction.

Delayed branch instructions

jrgt.d jrge.d jrlt.d jrle.d jrugt.d jruge.d jrult.d jrule.d jreq.d jrne.d call.d jp.d ret.d

Delayed instructions

The delayed instruction must meet all the following conditions:

- 1 cycle instruction
- Does not access memories
- Not extended with the "ext" instruction

The following instructions can be used as a delayed instruction:

ld.w	%rd, %rs	ld.w	%rd, sign6				
add	%rd, %rs	add	%rd, imm6	add	%sp, imm10	adc	%rd, %rs
sub	%rd, %rs	sub	%rd, imm6	sub	%sp, imm10	sbc	%rd, %rs
mlt.h	%rd, %rs	mltu.h	%rd, %rs				
cmp	%rd, %rs	cmp	%rd, sign6				
and	%rd, %rs	and	%rd, sign6				
or	%rd, %rs	or	%rd, sign6				
xor	%rd, %rs	xor	%rd, sign6				
not	%rd, %rs	not	%rd, sign6				
srl	%rd, %rs	srl	%rd, imm4	sll	%rd, %rs	sll	%rd, imm4
sra	%rd, %rs	sra	%rd, imm4	sla	%rd, %rs	sla	%rd, imm4
rr	%rd, %rs	rr	%rd, imm4	rl	%rd, %rs	rl	%rd, imm4
scan0	%rd, %rs	scan1	%rd, %rs				
swap	%rd, %rs	mirror	%rd, %rs				

Note: Do not use instructions that do not meet the conditions of a delayed instruction, if used the operation cannot be guaranteed.

The delayed instruction is executed regardless of the delayed instruction type (conditional or unconditional branch) and whether the program flow is branched or not.

When a branch instruction without a delayed function (that has no ".d") is executed, the instruction at the next address will not be executed if the program flow branches. If the branch instruction is a conditional branch instruction and the program flow does not branch, the instruction at the next address is executed following the branch instruction.

The "call.d" instruction saves the address of the instruction that follows the delayed instruction into the stack as the return address. The delayed instruction is not executed when returning from the subroutine.

Traps such as interrupts and exceptions do not occur between a delayed branch instruction and the delayed instruction because the hardware masks traps.

2.5.13 System control instructions

The following three instructions are used for controlling the system and do not affect the registers and memories:

nop	No operation (increments PC only)
halt	Sets the CPU to HALT mode.
slp	Sets the CPU to SLEEP mode.

See Section 3.4, "Power Down Mode", for HALT and SLEEP modes.

2.5.14 Scan instructions

The scan instruction scans 0 or 1 bit within the high-order 8 bits of the specified general-purpose register from the MSB, and returns the first found bit position.

scan0 %rd, %rs

Scans the high-order 8 bits of the rs register from the MSB. When a bit of 0 is found, the bit position (offset from the MSB) is loaded to the rd register. Bit 31 to Bit 4 of the rd register are all set to 0. If there is no 0, 0x00000008 is loaded to the rd register and the C flag is set to 1.

Example:

High-order 8 bits of rs	Low-order 8 bits of rd	PSR			
		C	V	Z	N
0xxx xxxx	0000 0000	0	0	1	0
10xx xxxx	0000 0001	0	0	0	0
110x xxxx	0000 0010	0	0	0	0
1110 xxxx	0000 0011	0	0	0	0
1111 0xxx	0000 0100	0	0	0	0
1111 10xx	0000 0101	0	0	0	0
1111 110x	0000 0110	0	0	0	0
1111 1110	0000 0111	0	0	0	0
1111 1111	0000 1000	1	0	0	0

scan1 %rd, %rs

Scans the high-order 8 bits of the rs register from the MSB. When a bit of 1 is found, the bit position (offset from the MSB) is loaded to the rd register. Bit 31 to Bit 4 of the rd register are all set to 0. If there is no 1, 0x00000008 is loaded to the rd register and the C flag is set to 1.

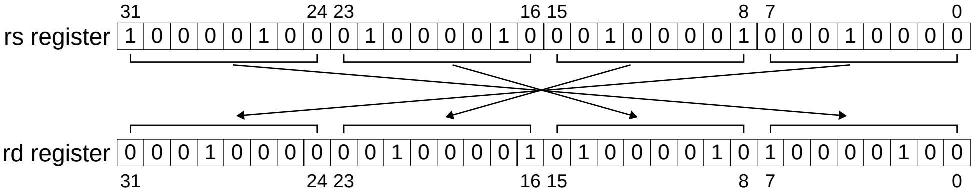
Example:

High-order 8 bits of rs	Low-order 8 bits of rd	PSR			
		C	V	Z	N
1xxx xxxx	0000 0000	0	0	1	0
01xx xxxx	0000 0001	0	0	0	0
001x xxxx	0000 0010	0	0	0	0
0001 xxxx	0000 0011	0	0	0	0
0000 1xxx	0000 0100	0	0	0	0
0000 01xx	0000 0101	0	0	0	0
0000 001x	0000 0110	0	0	0	0
0000 0001	0000 0111	0	0	0	0
0000 0000	0000 1000	1	0	0	0

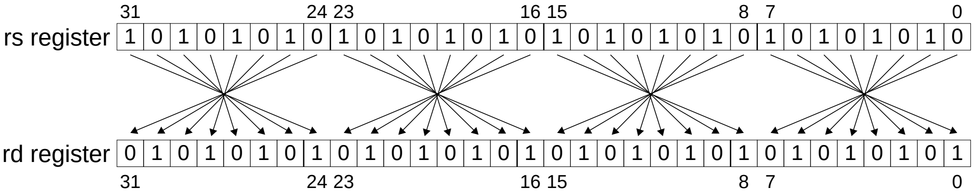
2.5.15 Swap and mirror instructions

The swap and mirror instructions replace the bit order of a general-purpose register as shown below.

Swap instruction: swap %rd, %rs


31 24 23 16 15 8 7 0
rs register 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0
rd register 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0
31 24 23 16 15 8 7 0

Mirror instruction: mirror %rd, %rs


31 24 23 16 15 8 7 0
rs register 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
rd register 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
31 24 23 16 15 8 7 0

CHAPTER 3 CPU OPERATION AND PROCESSING STATUS

This chapter describes the outline of the CPU processing status and operations. Refer to the "Technical Manual" of each E0C33 Family model for more information.

3.1 Processing Status of CPU

Figure 3.1.1 shows the status transition of the E0C33000.

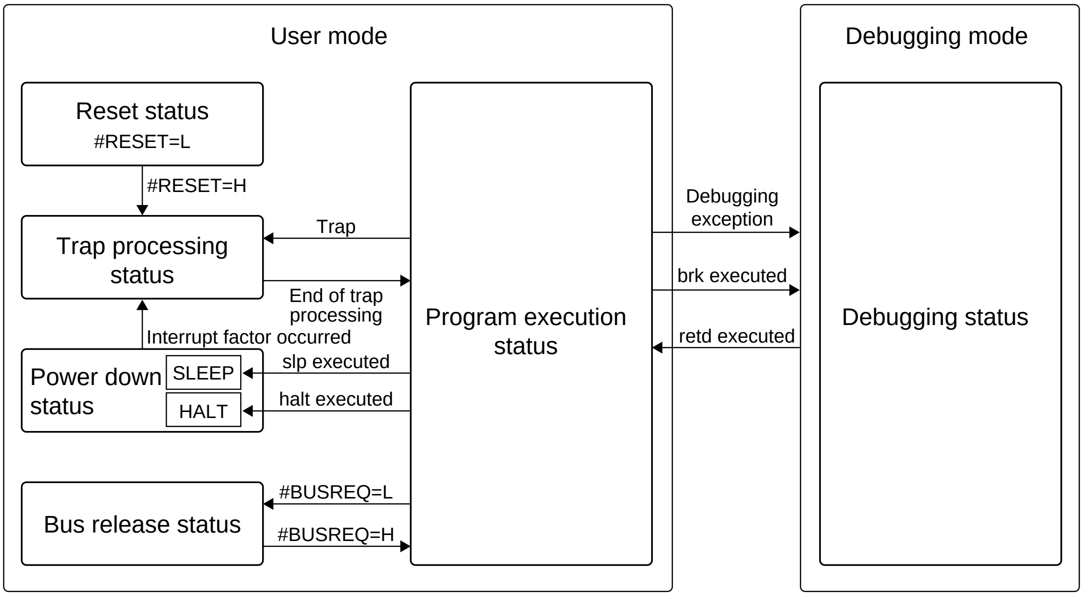


Fig. 3.1.1 Status transition diagram

User mode

The E0C33000 executes the application program in the user mode.
At initial reset, the E0C33000 is set to this mode. In this mode, the E0C33000 is placed in one of the following five processing statuses:

(1) Reset status
In the reset status, the CPU initializes the internal circuits and stops operation.

(2) Program execution status
In this status, the CPU executes the user program sequentially.

(3) Trap processing status
This is a transition period after an interrupt or exception occurs. The CPU branches the program to the handler routine for the trap.

(4) Power down status
In this status, the CPU stops operation to reduce current consumption.

(5) Bus release status
In this status, the CPU releases the bus and waits until the external bus master finishes the bus operation.

Debugging mode

The E0C33000 has the debugging support functions for efficient development. Those functions can be used only in the debugging mode. The "brk" instruction and debugging exceptions switch the CPU from the user mode to this mode. Usually, the CPU does not enter this mode.

3.2 Program Execution Status

Usually the CPU operates in this status, and executes the user program in the ROM/RAM sequentially. The PC (program counter) maintains the address being executed and is incremented every time an instruction is executed. When a branch instruction is executed, the branch destination address is loaded to the PC and the program branches to the address.

The program execution status is suspended by the occurrence of a trap, execution of the "halt" or "slp" instruction or a bus request from a peripheral circuit, then the CPU enters the processing status according to the factor that has occurred.

3.2.1 Fetching and executing program

The E0C33000 performs three stages of pipe-line processing that executes an instruction and fetches an instruction expected to execute simultaneously in order to increase the processing speed. Further the CPU can access the internal ROM (program memory) and the internal RAM (data memory) at the same time with the Harvard architecture.

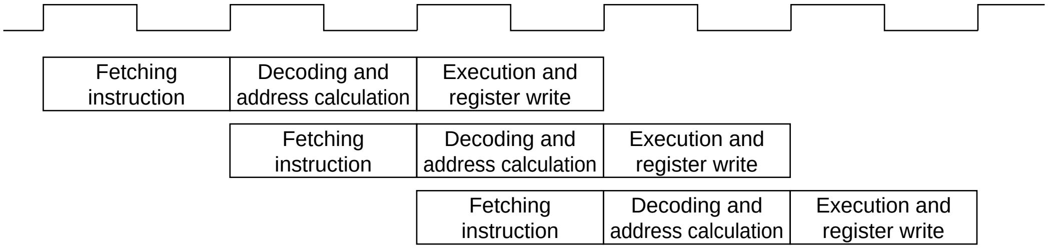

Fetching instruction
Decoding and address calculation
Execution and register write
Fetching instruction
Decoding and address calculation
Execution and register write
Fetching instruction
Decoding and address calculation
Execution and register write

Fig. 3.2.1.1 Fetch and execution of program

3.2.2 Number of instruction execution cycles

The E0C33000 can execute the principle instructions in 1 cycle. See the instruction list in the Appendix for the number of execution cycles of each instruction. Note that this manual describes the execution cycles only when the program in the internal ROM and data in the internal ROM are accessed. The following supplements the execution cycles when external memory/devices are used for reference when calculating execution times. However, the following indicates simplified calculation methods. Actual execution cycles may vary due to the combination of instructions and memory map settings.

(1) When fetching instructions from an external memory area, the execution time will be prolonged for [wait cycle count + 1] cycles. (The wait cycle count varies depending on the device of each area.)

(2) When reading/writing data from/to an area other than the internal RAM using a load instruction, the execution time will be prolonged for [wait cycle count + 1] cycles.

(3) When accessing the internal RAM for both fetching instructions and writing/reading data, the execution time will be prolonged for 1 cycle per one data accessing.

(4) Fetching instructions and writing/reading data execute 1, 2 or 4 bus operations according to the transfer data size and the connected device size. The execution time will be prolonged according to the bus operation count. Further wait cycles will be added to each bus operation. For example, when fetching an instruction from an 8-bit external ROM without a wait cycle, 2 bus operations will be executed and the execution time will be prolonged for 3 cycles.

(5) Besides the above factors, the following factors among the external bus conditions that have been set in the BCU (bus control unit) affect the execution cycle count:
- Output disable cycles set for the device on the external bus
- RAS cycles, pre-charge cycles and refresh cycles for the DRAM
- Wait cycles using the external #WAIT terminal

(6) The instructions below access data several times. Therefore the execution time will be prolonged for [wait cycle count + 1] cycles per one data access.

• Bit operation instructions (btst)	1 (data access count)
• Bit operation instructions (bset, bclr, bnot)	2
• Push and pop instructions (pushn, popn)	n
• Multiplication and accumulation instruction (mac)	2n
• Software exception (int)	3
• Return from trap handler routine (reti)	2
• Debugging exception (brk)	3
• Return from debugging routine (retd)	2

(7) Delay by interlock

When using the destination register (%rd) of the previous load instruction that transferred memory data to the general-purpose register as the operation source of the next instruction (when the %rs or %rd is the same as the previous %rd), the execution time will be prolonged for 1 cycle to eliminate the interlock.

Refer to the "Technical Manual" of each E0C33 Family model for the BCU and external bus conditions such as the wait cycle.

3.3 Trap (Interrupts and Exceptions)

The CPU goes to the trap processing status when a trap factor (interrupt or exception) occurs during program execution. The trap processing status is a transition period until the CPU branches the program flow to the user handler routine corresponding to the interrupt/exception factor that has occurred. The CPU returns to the program execution status after branching.

3.3.1 Trap table

Table 3.3.1.1 lists the traps of the E0C33000.

Table 3.3.1.1 Trap list

Trap name	Sync./Async.	Classification	Vector address	Priority	Interrupt level after trapping
Reset	Async.	Interrupt	base+0	Highest	Level 0
Reserved			base+4~12	↑	Unchanged
Zero division	Sync.	Exception	base+16		Unchanged
Reserved			base+20		Unchanged
Address error exception	Sync.	Exception	base+24		Unchanged
Debugging exception (brk, others)	Sync.	Exception	0x0 or 0x60000		Unchanged
NMI	Async.	Interrupt	base+28		Unchanged
Reserved			base+32~44		Unchanged
Software exception 0 : Software exception 3	Sync. : Sync.	Exception : Exception	base+48 : base+60		Unchanged : Unchanged
Maskable external interrupt 0 : Maskable external interrupt 215	Async. : Async.	Interrupt : Interrupt	base+64 : base+924	↓ Lowest	Interrupt level (Level 0 to 15) of the peripheral circuit that requested the interrupt.

The E0C33000 has seven trap factors listed in the Trap name column (details are described later).

"Sync./Async." indicates either the trap factor will occur in synchronization with program execution or asynchronously. This manual classifies the trap factors into two types: "Exception" that will occur in synchronization with program execution and "Interrupt" that will occur asynchronously. However, this manual uses "Trap Processing" for all trap processing of the CPU.

The vector address stores the vector (branch destination address) of the user handler routine that is executed when each trap occurs. The vector addresses are arranged at a word boundary address because they store an address. The memory area for vector storage is called a trap table. The "base" in the vector address column indicates the trap table beginning address.

The E0C33000 allows the base (starting) address of the trap table to be set by the TTBR register.
TTBR0 =D(9:0)/0x48134: Trap table base address (9:0) ... fixed at 0
TTBR1 =D(F:A)/0x48134: Trap table base address (15:10)
TTBR2 =D(B:0)/0x48136: Trap table base address (27:16)
TTBR3 =D(F:C)/0x48136: Trap table base address (31:28) ... fixed at 0

After a cold start (see Section 3.3.3), the TTBR register is set to the boot address determined by the BTA3 pin status.

Table 3.3.1.2 Trap table location

BTA3 terminal	Trap table location
High	Area 3 (Top of the internal ROM; base=0x0080000)
Low	Area 10 (Top of the external ROM; base=0x0C00000)

Therefore, even when the trap table position is changed, it is necessary that at least the reset vector be written to the above address for cold starting. A hot start does not change the TTBR setting.
TTBR0 and TTBR3 are read-only registers which are fixed at “0”. Therefore, the trap table starting address always begins with a 1KB boundary address.
The TTBR registers are normally write-protected to prevent them from being inadvertently rewritten. To remove this write protection function, another register, the TBRP register (D(7:0)/0x4812D), is provided. A write to the TTBR register is enabled by writing "0x59" to the TBRP register and is disabled back again by a write to the most significant byte of the TTBR register (0x48137). Consequently, a write to the TTBR register needs to begin with the low-order half-word first. However, since an occurrence of NMI or the like between writes of the low-order and high-order half-words would cause a malfunction, it is recommended that the register be written in words.

The accessible memory space differs depending on the model. A word sized area is reserved for each vector, however the lower effective bit size only is actually used for the vector. Furthermore the LSB of the vector is handled as 0 because the vector is an address in the program memory.
The trap table size is decided by the number of the maskable interrupts of each model.

The priority indicates which trap is accepted first when two or more traps occur at the same time. Exceptions do not occur at the same time because they occur when an instruction is executed. The reset factor is accepted taking priority over all other processing. The priority of maskable interrupts are also managed by the interrupt levels (described later). Therefore the priority of the maskable interrupts shown in Table 3.3.1.1 assumes that all interrupts have same priority.

See Section 3.3.8, "Maskable external interrupts", for the interrupt level after trapping.

3.3.2 Trap processing

The CPU executes the trap processing shown below when a trap except for reset and debugging exceptions occurs. However the following processing does not apply to the reset processing. It is explained in the next section. The debugging exception is explained in Section 3.6.

(1) Terminates or cancels the instruction being executed.
(2) Saves the contents of the PC and the PSR sequentially into the stack.
(3) Resets the IE (interrupt enable) bit in the PSR to disable maskable interrupts after this point.
Modifies the IL (interrupt level) field in the PSR to the occurred interrupt level if the trap is a maskable interrupt.
(4) Reads the vector corresponding to the trap factor from the trap table and loads it to the PC. It branches the processing to the user handler routine.

The above sequence is the trap processing of the CPU.

When the "reti" instruction is executed at the end of the user handler routine, the contents of the PSR and the PC that have been saved into the stack return to each register and the processing that is suspended by the trap resumes execution.
The "ret" instruction cannot be used for return from trap handler routines because the instruction does not return the PSR.

The CPU masks traps in the following cases, and traps except for reset are not accepted until the masking factors are canceled:

(1) When the "ext" instruction is executed:
When the "ext" instruction is executed, traps are masked until finishing execution of the following target instruction. However address error exception is excluded.

(2) When a delayed branch instruction is executed:
When a delayed branch instruction (.d) is executed, traps are masked until starting execution of the following delayed instruction.

(3) NMI before setting SP
When the CPU is reset, the NMI is masked until data is written to the SP (stack address is set) in order to prevent program runaways.
Exceptions are not masked because they can be predicted. Maskable interrupts are also not masked because they have been masked by the IE bit in the PSR after reset.

3.3.3 Reset

The CPU is reset when a low pulse is input to the #RESET terminal. The initial reset clears all the bits in the PSR and makes other registers undefined.
The CPU starts operating at the rising edge of the #RESET pulse and executes the reset processing. The reset processing reads the reset vector from the top of the trap table and sets it to the PC. It starts executing the user initial routine.
The reset processing has priority over all other processing.

The E0C33000 supports two reset methods: Hot start and Cold start. The #NMI terminal is used with the #RESET terminal to set this condition.

Cold start (#RESET = L, #NMI = H)
The E0C33 Family MPU cold-starts when it is reset by setting the #RESET terminal to low and the #NMI terminal to high. Since cold start initializes all the on-chip peripheral circuits as well as the CPU, it is useful as a power-on reset.

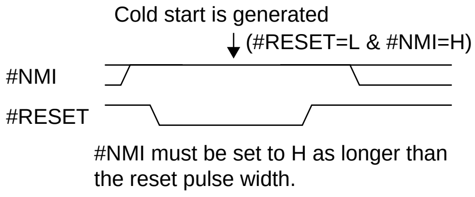


Fig. 3.3.3.1 Cold start timing

Hot start (#RESET = L, #NMI = L)
The E0C33 Family MPU hot-starts when it is reset by setting the #RESET and #NMI terminals to low. Hot start initializes the CPU but does not initialize some peripheral circuits such as the external bus control unit and the input/output ports. It is useful as a reset that maintains the external memory and external input/output statuses.

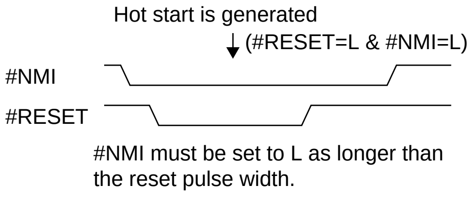


Fig. 3.3.3.2 Hot start timing

Refer to the "Technical Manual" of each E0C33 Family model for the reset timing and the initialization for the peripheral circuits.

3.3.4 Zero division exception

A zero division exception will occur if the divisor is 0 when the division instruction is executed.
This exception may occur with the "div0s" or "div0u" instruction for preprocessing of division. If the divisor is 0, the CPU executes the trap processing after finishing execution of the instruction. The trap processing saves the next instruction address (usually "div1") into the stack as the return address.
However, the exception may occur at the next instruction due to the pipe line processing.

3.3.5 Address error exception

The load instructions for accessing a memory or I/O area have a predefined transfer data size. The address to be specified must be a boundary address according to the data size.

Instruction	Transfer data size	Address
ld.b/ld.ub	Byte (8 bits)	Byte boundary (any address can be specified within the usable area)
ld.h/ld.uh	Half word (16 bits)	Half word boundary (LSB of the address must always be 0)
ld.w	Word (32 bits)	Word boundary (low-order 2 bits must always be 0)

If the specified address of a load instruction does not meet the condition, the CPU regards it as an address error and executes the trap processing. In this case, the CPU does not execute the load instruction and saves the load instruction address into the stack as the return address.
Normally, traps are masked when the "ext" instruction is executed until the next instruction is executed. However only the address error exception is not masked. Therefore if an address error exception occurs in a load instruction that follows the "ext" instruction (the load instruction has to be executed in register indirect addressing with displacement), the CPU enters in the trap processing before executing the load instruction. Be aware that it may be a problem if return from the trap handler routine is done by simply executing the "reti" instruction. In this case, the load instruction is executed independently in register indirect addressing mode without displacement.

The address error exception may also occur by the multiplication and accumulation (mac) instruction because it handles half word data. The trap processing saves the "mac" instruction address into the stack as the return address, so the "mac" instruction will resume the remaining multiplication and accumulation after returning from the trap handler routine.

The load instructions that use the SP for specifying the base address do not issue an address error exception because the address is adjusted at the boundary according to the transfer data size.

In the branch instructions ("call %rb", "jp %rb"), this exception does not occur because the LSB of the PC is always fixed at 0. It is the same for trap processing vectors.

3.3.6 NMI (Non-maskable interrupt)

When the #NMI signal (low) is assigned to the CPU, an NMI occurs at the falling edge.
When an NMI occurs, the CPU executes the trap processing after finishing the instruction being executed. The trap processing saves the next instruction address into the stack as the return address.
The NMI cannot be masked. However, when the CPU is reset (both cold start and hot start), the #NMI input is masked by the hardware until the SP is set by the "ld.w %sp, %rs" instruction in order to prevent program runaways due to undefined SP.

3.3.7 Software exception

A software exception occurs when the "int imm2" instruction is executed. The trap processing saves the address of the instruction that follows the "int" instruction into the stack as the return address. The imm2 in the "int" instruction specifies a vector address among four software exceptions. The CPU reads the vector from the address calculated by adding 4 × imm2 to base + 48 (vector address for software exception 0) for branching to the handler routine.

3.3.8 Maskable external interrupts

The E0C33000 can accept up to 128 maskable external interrupts (except for the NMI).
Maskable interrupts are accepted to the CPU only when the IE (interrupt enable) bit in the PSR has been set. Further, the IL (interrupt level) field in the PSR also affects the acceptance. The IL field contains an interrupt level number (0 to 15) that indicates the acceptable interrupt level. The CPU can only accept interrupts that have an interrupt level higher than the IL value.
The IE bit and the IL field can be set by software. Furthermore, when a trap occurs, the IE bit is reset to 0 (interrupt is disabled) after saving the PSR into the stack. Therefore maskable interrupts are disabled until the IE bit is set in the handler routine or the handler routine is terminated by the "reti" instruction that returns the PSR.
The IL field is also set to the interrupt level that has occurred. To enable multiple interrupt processing, set the IE flag in the interrupt handler routine. It allows acceptance of interrupts that have higher levels than the currently processed interrupt.
Resetting the CPU initializes the PSR to 0, therefore maskable interrupts are disabled and the interrupt level is set to 0 (levels 1 to 15 are enabled).

All the E0C33 Family models have an on-chip interrupt controller, and the controller manages the interrupt request to the CPU.
The following shows the interrupt request procedure of the on-chip interrupt controller and the trap processing of the CPU:

(1) The on-chip interrupt controller requests an interrupt by setting the #INTREQ terminal to low. At the same time, it delivers the interrupt level to the INTLEV(3:0) terminals and the vector number to the INTVEC(7:0) terminals.
(2) When the CPU accepts the interrupt request, it saves the PC and the PSR into the stack, then resets the IE bit in the PSR and sets the IL field to the level according to the INTLEV signal.
(3) The CPU reads the vector from the vector address specified by the INTVEC signal and sets it to the PC for branching to the interrupt handler routine.

Refer to the "Technical Manual" of each model for use of the interrupt controller.

3.4 Power Down Mode

The CPU can stop operating in order to reduce current consumption when program execution is not necessary, in particular standby status awaiting a key entry. For this purpose, the E0C33000 has two power down modes: HALT mode and SLEEP mode.
The internal registers maintain the contents in the power down mode.

3.4.1 HALT mode

When the CPU executes the "halt" instruction, it suspends the program execution and goes into the HALT mode.
In the HALT mode, the CPU stops operating. The on-chip peripheral circuits keep operating since the clocks are supplied.
The HALT mode is canceled by initial reset or an interrupt including NMI. The CPU transits to program execution status through trap processing for the trap factor. When an interrupt cancels the HALT mode, the trap processing saves the address of the instruction that follows the "halt" instruction into the stack. Therefore, when the interrupt handler routine finishes by the "reti" instruction, the program flow returns to the instruction that follows the "halt" instruction.

3.4.2 SLEEP mode

When the CPU executes the "slp" instruction, it suspends the program execution and goes into the SLEEP mode.
In the SLEEP mode, the CPU and the on-chip peripheral circuits stop operating. Thus the SLEEP mode can greatly reduce current consumption in comparison to the HALT mode.
The SLEEP mode is canceled by initial reset or an interrupt including NMI. The CPU transits to program execution status through trap processing for the trap factor. When an interrupt cancels the SLEEP mode, the trap processing saves the address of the instruction that follows the "slp" instruction into the stack. Therefore, when the interrupt handler routine finishes by the "reti" instruction, the program flow returns to the instruction that follows the "slp" instruction.
Since the SLEEP mode stops the on-chip oscillation circuit, the peripheral circuits that use the oscillation clock also stop. Therefore the SLEEP mode is canceled by a key-entry interrupt.
When the SLEEP mode is canceled, the on-chip oscillation circuit starts oscillating. The CPU waits until the oscillation stabilizes then starts operating.

Refer to the "Technical Manual" of each model for peripheral circuit status in the HALT mode and SLEEP mode and the cancellation method.

3.5 Bus Release Status

The external bus in which external peripheral devices are connected is normally controlled by the CPU. It can be released for external devices in order to support the DMA (direct memory access) functions and multiprocessor systems.
The #BUSREQ and #BUSACK terminals are used for bus arbitration.
The bus release sequence is as follows:

(1) The external device which requests the bus authority sets the #BUSREQ terminal to low.
(2) The CPU always monitors the #BUSREQ status. When the terminal goes to low level, the CPU finishes the bus cycle being executed and waits 1 cycle, then switches the address bus (A27–A0), data bus (D15–D0) and bus control signals (#RD, #WRL, #WRH) into high-impedance status.
1 cycle later the CPU sets the #BUSACK terminal to low level indicating that the bus is released to the external device.
(3) After Step (2), the external device becomes the external bus master and executes its bus cycles. The external bus master must fix the #BUSREQ terminal at low level while executing the bus cycles.
(4) The external bus master returns the bus to high-impedance and the #BUSREQ terminal to high level after completing the necessary bus cycles.
(5) When the #BUSREQ terminal goes to high level, the CPU sets the #BUSACK terminal to high level 1 cycle later and resumes the suspended processing.

In the some models, the CPU has to take back the bus authority in the bus release status (for example, models using DRAM need refresh cycles). In this case, the CPU requests returning bus authority using a peripheral circuit such as an output port. The external bus master device has to handle the signal. Refer to the "Technical Manual" of each E0C33 Family model for details.

3.6 Debugging Mode

The E0C33000 has a special operating mode called a debugging mode.
This mode has been implemented to support debugging during development and is not used in the application program on the products. This section describes the outline as a CPU function.

3.6.1 Functions of debugging mode

The E0C33000 has incorporated the following debugging functions:

- **Single step**
  A debugging exception can be generated before executing each instruction of the user target program.
- **Instruction break**
  Up to three instruction break points can be set. A debugging exception can be generated before executing the instructions at the set addresses.
- **Data break**
  A data break address and a read/write condition can be set. The specified data access can generate a debugging exception. When the specified address is accessed in the specified read/write condition, a debugging exception occurs after 1 or several instructions is executed from the data access.
- **Software break**
  By executing the "brk" instruction, a debugging exception can be generated. The debugging exception saves the address following the "brk" instruction into the stack for the debugging mode.

When a debugging exception occurs, the CPU executes a trap processing that differs from the user mode and enters the debugging mode.
In the debugging mode, the user target program can be suspended at any address and executed in single stepping by executing debugging routines which are created by the user or provided by Seiko Epson.

3.6.2 Configuration of Area 2

The E0C33000 has reserved Area 2 (0x0060000 to 0x007FFFF, 128KB) in the address space for ICE (in-circuit emulator) use. In this area, the debug-control registers are allocated.
Addresses 0x0060010 to 0x0077FFF are reserved for the ICE control software and the area from address 0x0078000 is reserved for the debug-control registers and exclusive use of the CPU.
Note that writing data to the registers in Area 2 is not allowed in the user mode. It should be done in the debugging mode after a debugging exception occurs. The debugging mode has no such restriction, so all the areas can be accessed.

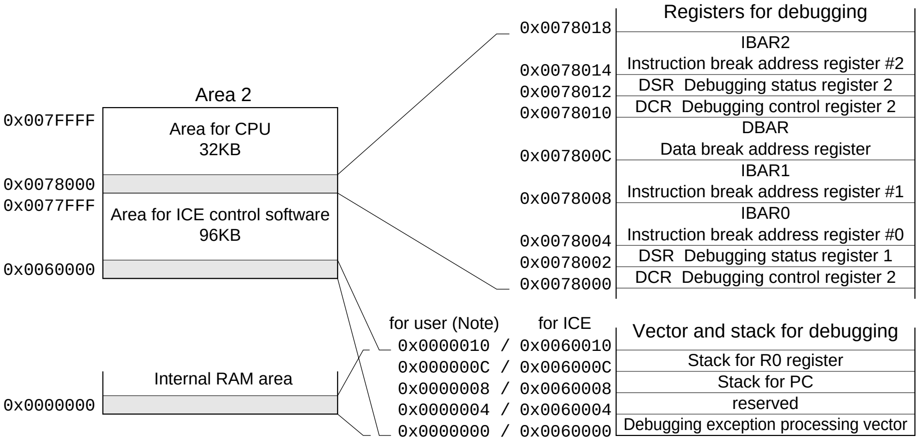


Fig. 3.6.2.1 Configuration of Area 2

Note: When the user sets the debugging mode, the debugging exception processing vector will be read from address 0x0000000. The PC and R0 register values are saved to address 0x0000008 and 0x000000C, respectively.
The MON33 (debug monitor) was created for this condition.

3.6.3 Transition from user mode to debugging mode

When a debugging exception occurs (e.g. the "brk" instruction is executed), the CPU executes the debugging exception processing to switch from the user mode to the debugging mode. The differences between debugging exception processing and normal exception processing are shown as follows:

- It does not use the normal trap table; a vector for entering in the debugging mode is read from address 0x0000000 in Area 0 or address 0x0060000 for ICE use.
- The R0 register and PC values are saved (PSR is not saved) and the stack area for the normal mode is not used. The R0 register is saved to address 0x000000C or address 0x006000C for ICE use and the PC value is saved to address 0x0000008 or address 0x0060008 for ICE use.

To switch from the debugging mode to the user mode, execute the "retd" instruction. The "retd" instruction restores the saved R0 and PC values before returning to the user mode.

3.6.4 Registers for debugging

The registers that control the debugging function are arranged in Area 2, and can be written only in the debugging mode. The following shows the contents and functions of each register:

DCR (Debugging Control Register): 0x0078000/Byte size, 0x0078010/Byte size

	7	6	5	4	3	2	1	0	
0x0078000	–	MWRBE	MRDBE	DBE	IBE(1:0)		SE	DM	R/W (DM: R only)
0x0078010	–	–	(Note)	(Note)	(Note)	(Note)	(Note)	IBE(2)	R/W

Note: Be sure to set bits 5 to 1 in address 0x0078010 to the values as follows. Other settings will cause the debugging mode to not function normally.
Bits 5 and 4: Fixed at 0. Bits 3–1: Fixed at 1.

The DCR enables/disables the debugging functions. At initial reset, all the bits in the DCR are reset to 0.

0x0078000

Name	Bit No.	Bit status		Function
		1	0	
DM	0	Debugging mode	User mode	Debugging Mode: Indicates that the CPU is in the debugging mode. When a debugging exception occurs, the DM is set (1) and the CPU enters the debugging mode. When the "retd" instruction is executed in the debugging routine, the DM is reset (0) and the CPU returns to the user mode. The DM is a read only bit, so it cannot be modified by software.
SE	1	Enabled	Disabled	Single Step Enable: Enables and disables the single step function. When the SE is set (1), the single step function is enabled and a debugging exception will occur before executing each instruction of the user program in the user mode. The debugging mode does not perform single step operations. When the SE is reset (0), the single step function is disabled.
IBE(1:0)	2, 3	Enabled	Disabled	Instruction Break Enable: Enables and disables the instruction break function. IBE(0) (bit 2) and IBE(1) (bit 3) correspond to the instruction break points #0 and #1, respectively. When the IBE(0) (IBE(1)) bit is set (1), the break address that has been set in the IBAR0 (IBAR1) register becomes effective. When the instruction of the address is fetched during program execution in the user mode, a debugging exception occurs before executing the instruction. In the debugging mode, the instruction break does not occur. When the IBE bit is reset (0), the instruction break point is invalidated.
DBE	4	Enabled	Disabled	Data Break Enable: Enables and disables the data break function. When the DBE is set (1), the data break address that has been set in the DBAR register becomes effective. When the address is accessed during program execution in the user mode, a debugging exception occurs after accessing data. In the debugging mode, the data break does not occur. A data access condition (read, write, read/write) for generating a break can be specified using the MRDBE and MWRBE bits. When the DBE is reset (0), the data break function is disabled. When both the MRDBE (read) and MWRBE (write) are reset, a data break does not occur even if the DBE has been set.

Name	Bit No.	Bit status		Function
		1	0	
MRDBE	5	Enabled	Disabled	Memory Read Break Enable: Enables and disables the memory read data break function. When the DBE and the MRDBE are set (1), a data break will occur after the CPU reads data in the specified address. When the MRDBE is reset (0), the memory read data break function is disabled.
MWRBE	6	Enabled	Disabled	Memory Write Break Enable: Enables and disables the memory write data break function. When the DBE and the MWRBE are set (1), a data break will occur after the CPU writes data to the specified address. When the MWRBE is reset (0), the memory write data break function is disabled.

0x0078010

Name	Bit No.	Bit status		Function
		1	0	
IBE(2)	0	Enabled	Disabled	Instruction Break Enable: Enables and disables the instruction break function. IBE(2) corresponds to the instruction break point #2. When the IBE(2) bit is set (1), the break address that has been set in the IBAR2 register becomes effective. When the instruction of the address is fetched during program execution in the user mode, a debugging exception occurs before executing the instruction. In the debugging mode, the instruction break does not occur. When the IBE(2) bit is reset (0), the instruction break point is invalidated.

DSR (Debugging Status Register): 0x0078002/ Byte size, 0x0078012/ Byte size

	7	6	5	4	3	2	1	0	
0x0078002	BKF	MWRB	MRDB	DB	IB1	IB0	SS	DR	R/W
0x0078012	–	–	–	–	–	–	–	IB2	R/W

The DSR is the status register that indicates the debugging exception that has occurred. When a debugging exception occurs, the same vector is used to execute the debugging exception processing. Therefore, the debugging exception service routine must identify the occurred debugging exception type by reading the DSR.

0x0078002

Name	Bit No.	Bit status		Function
		1	0	
DR	0	Occurred	Non	Debug Request: Indicates that the external debugging request was assigned. The DR is set (1) at the falling edge of the external debugging request signal #DBGREQ. This function is only for the ICE, general chips do not have the #DBGREQ terminal.
SS	1	Occurred	Non	Single Step: Indicates that a single step break occurred. The SS is set (1) when a debugging exception occurs by the single step factor.
IB0	2	Occurred	Non	Instruction Break 0: Indicates that the instruction break #0 occurred. The IB0 is set (1) when a debugging exception occurs by the instruction break #0 factor.
IB1	3	Occurred	Non	Instruction Break 1: Indicates that the instruction break #1 occurred. The IB1 is set (1) when a debugging exception occurs by the instruction break #1 factor.
DB	4	Occurred	Non	Data Break: Indicates that the data break occurred. The DB is set (1) when a debugging exception occurs by the data break factor.
MRDB	5	Occurred	Non	Memory Read Break: Indicates that the memory read data break occurred. The MRDB is set (1) when a debugging exception occurs by the data break with a memory read.
MWRB	6	Occurred	Non	Memory Write Break: Indicates that the memory write data break occurred. The MWRB is set (1) when a debugging exception occurs by the data break with a memory write.
BKF	7	Occurred	Non	Break Flag: Indicates that the "brk" instruction was executed. The BKF is set when a debugging exception occurs by executing the "brk" instruction.

0x0078012

Name	Bit No.	Bit status		Function
		1	0	
IB2	0	Occurred	Non	Instruction Break 2: Indicates that the instruction break #2 occurred. The IB2 is set (1) when a debugging exception occurs by the instruction break #2 factor.

IBAR0 (Instruction Break Address Register #0): 0x0078006 (bits 27–16), 0x0078004 (bits 15–0)
IBAR1 (Instruction Break Address Register #1): 0x007800A (bits 27–16), 0x0078008 (bits 15–0)
IBAR2 (Instruction Break Address Register #2): 0x0078016 (bits 27–16), 0x0078014 (bits 15–0)

0x0078007	0x0078006	0x0078005	0x0078004	
31	27	1	0	
Invalid	IBAR0		0	R/W

0x007800B	0x007800A	0x0078009	0x0078008	
31	27	1	0	
Invalid	IBAR1		0	R/W

0x0078017	0x0078016	0x0078015	0x0078014	
31	27	1	0	
Invalid	IBAR2		0	R/W

Three instruction break addresses #0–#2 can be set to these registers. The LSB is always handled as 0, and only bits from bit 27 to bit 1 are effective.
When IBE(0)/IBE(1)/IBE(2) in the DCR has been set (1), the content of IBAR0/IBAR1/IBAR2 is compared with the PC during program execution in the user mode. A debugging exception will occur if they are matched. These registers enable read/write operation.

DBAR (Data Break Address Register): 0x007800E (bits 27–16), 0x007800C (bits 15–0)

0x007800F	0x007800E	0x007800D	0x007800C	
31	27		0	
Invalid	DBAR			R/W

A data break address can be set in this register.
When the DBE in the DCR has been set (1), the content of the DBAR is compared with the accessed memory address during program execution in the user mode. A debugging exception will occur if they are matched and the specified read/write condition is met. This register enables read/write operation.
The data break does not occur if all the bits in the DBAR are not completely matched to the base address of the accessed memory. Therefore, when generating a data break by reading/writing word data, the address to be specified must point a word boundary address (low-order 2 bits are 0). Similarly, a half word boundary address (LSB is 0) should be set in this register for generating by half word access.

3.6.5 Traps in debugging mode

In the debugging mode, the exceptions except for reset, address error, zero division, software exception ("int" instruction) and interrupts (including NMI) are masked and do not occur. The normal exception processing is executed when an address error, zero division or a software exception occurs.
Furthermore, when the CPU returns to the user mode from the debugging mode by the "retd" instruction, exceptions other than reset and address error and interrupts are masked until the instruction at the return address is executed. Exceptions and interrupts after the instruction is executed are not masked.

3.6.6 Simultaneous occurrence of debugging exceptions

When two or more debugging exception factors occur at the same time, one debugging exception is only generated but the status bits in the DSR corresponding to all the occurred factors are set.

CHAPTER 4 DETAILED EXPLANATION OF INSTRUCTIONS

This chapter explains each instruction in the E0C33000 instruction set in alphabetical order.

4.1 Symbol Meanings

4.1.1 Registers

The following symbols indicate a register or the content:

%rd, rd: Indicates a general-purpose register (R0–R15) used as the destination or the content of the register.

%rs, rs: Indicates a general-purpose register (R0–R15) used as the source or the content of the register.

%rb, rb: Indicates a general-purpose register (R0–R15) that has stored a base address accessed in the register indirect addressing mode or the content of the register.

%sd, sd: Indicates a special register (PSR, SP, ALR, AHR) used as the destination or the content of the register.

%ss, ss: Indicates a special register (PSR, SP, ALR, AHR) used as the source or the content of the register.

%sp, sp: Indicates the stack pointer (SP) or the content of the SP.

In the mnemonic notation, a "%" must be prefixed to the register name in order to distinguish from symbols.

General-purpose registers:	%r0, %r1, %r2 · · · %r15, or %R0, %R1, %R2 · · · %R15
Special registers:	PSR .... %psr, or %PSR
	SP ...... %sp, or %SP
	ALR ... %alr, or %ALR
	AHR .. %ahr, or %AHR

The register field (rd, rs, sd, ss) in the instruction code contains the specified register number.

General-purpose registers (rd, rs):	R0 = 0b0000, R1 = 0b0001 · · · R15 = 0b1111
Special registers (sd, ss):	PSR = 0b0000, SP = 0b0001, ALR = 0b0010, AHR = 0b0011

4.1.2 Immediate

The following symbols indicate an immediate data:

immX: Indicates an unsigned X-bit immediate data. X is a number that indicates the bit size.

signX: Indicates a signed X-bit immediate data. X is a number that indicates the bit size. The MSB of the immediate data is handled as the sign bit.

4.1.3 Memories

The following symbols indicate a memory specification or the contents of the memory:

[%rb]: Specifies the register indirect addressing mode. The content of the general-purpose register (rb) is used as the base address to be accessed.

[%rb]+: Specifies the register indirect addressing with post-increment mode. The content of the general-purpose register (rb) is used as the base address to be accessed. The content of the rb register is incremented according the data size after accessing the memory.

[%sp+immX]: Specifies the register indirect addressing with displacement mode and used for specifying an address in the stack. The base address to be accessed is specified by adding the immediate data (immX) to the content of the SP.

B[rb]: Indicates the memory address specified by the general-purpose register (rb) or the byte data stored in the address.

B[rb+immX]: Indicates the memory address specified by adding the immediate data (immX) to the content of the general-purpose register (rb) or the byte data stored in the address.

B[sp+immX]: Indicates the memory address specified by adding the immediate data (immX) to the content of the SP or the byte data stored in the address.

H[rb]: Indicates the half word (16-bit) area in which the base address is specified by the content of the general-purpose register (rb) or the half word data stored in the area. Data in the base address is handled as the low-order byte.

H[rb+immX]: Indicates the half word (16-bit) area in which the base address is specified by adding the immediate data (immX) to the content of the general-purpose register (rb) or the half word data stored in the area. Data in the base address is handled as the low-order byte.

H[sp+immX]: Indicates the half word (16-bit) area in which the base address is specified by adding the immediate data (immX) to the content of the SP or the half word data stored in the area. Data in the base address is handled as the low-order byte.

W[rb]: Indicates the word (32-bit) area in which the base address is specified by the content of the general-purpose register (rb) or the word data stored in the area. Data in the base address is handled as the least significant byte.

W[rb+immX]: Indicates the word (32-bit) area in which the base address is specified by adding the immediate data (immX) to the content of the general-purpose register (rb) or the word data stored in the area. Data in the base address is handled as the least significant byte.

W[sp]: Indicates the word (32-bit) area in which the base address is specified by the content of the SP or the word data stored in the area. Data in the base address is handled as the least significant byte.

W[sp+immX]: Indicates the word (32-bit) area in which the base address is specified by adding the immediate data (immX) to the content of the SP or the word data stored in the area. Data in the base address is handled as the least significant byte.

4.1.4 Bits and bit fields

The symbols below indicate a bit number or a bit field of registers and memory data. They are used with a register or memory symbol.

(X): Indicates Bit X in data. LSB is indicated as (0).

(X:Y): Indicates a bit field from Bit X to Bit Y.

{X, Y . . .}: Indicates a bit (data) configuration. The left item is the high-order bit (data). It is also used to describe the 64-bit register pair {AHR, ALR}.

4.1.5 Flags

The following symbols indicate the flags in the PSR or set/reset status:

IL[3:0]:	Interrupt level field
MO:	MAC overflow flag
DS:	Dividend sign flag
IE:	Interrupt enable
C:	Carry flag
V:	Overflow flag
Z:	Zero flag
N:	Negative flag
–:	Indicates that the instruction does not affect the flag.
↔:	Indicates that the instruction sets (1) or resets (0) the flag.
0:	Indicates that the instruction resets (0) the flag.

4.1.6 Functions and others

The following symbols are used for function explanation:

←:	Indicates that the right item is loaded or set to the left item.
+:	Addition
-:	Subtraction
&:	AND
|:	OR
^:	XOR
!:	NOT
×:	Multiplication
÷:	Division

The following symbol is used for indicating two or more codes or mnemonics with one word:

∗: A number either 1 or 0, or any letter from a to z.

4.2 Instruction Code Class

In the E0C33000 instruction set, all the instructions are 16-bit fixed size.
The bit configuration of the instruction code is classified into 8 types (Class 0 to Class 7) according to the function and addressing mode. The high-order 3 bits indicate a Class.

Instructions for multiplication and division can be executed only in the models that have an optional multiplier. The following instructions function the same as the "nop" instruction in the models that have no multiplier and the AHR and the ALR cannot be used:

mlt.h	multu.h	mlt.w	multu.w	
div0s	div0u	div1	div2s	div3s
mac				
ld.w %rd, %ahr		ld.w %rd, %alr		
ld.w %ahr, %rs		ld.w %alr, %rs		

Class 0

This class contains one-operand instructions and branch instructions.

15		13	12 9	8	7 6	5	4	3 0
0	0	0	op1	d	op2	0	0	imm2/rd/rs

op1	op2	Mnemonic	Function
0000	00	nop	No operation
0000	01	slp	SLEEP mode
0000	10	halt	HALT mode
0000	11	reserved	
0001	00	pushn %rs	Push for general-purpose registers
0001	01	popn %rd	Pop for general-purpose registers
0001	1*	reserved	
0010	00	brk	Debugging exception
0010	01	retd	Return from debugging routine
0010	10	int imm2	Software exception
0010	11	reti	Return from trap handler routine
0011	00	call %rb	Subroutine call
0011	01	ret	Return from subroutine
0011	10	jp %rb	Unconditional jump
0011	11	reserved	

15		13	12 9	8	7 0
0	0	0	op1	d	sign8

op1	Mnemonic	Function
0100	jrgt sign8	PC relative conditional jump Condition = !Z & !(N ^ V)
0101	jrge sign8	PC relative conditional jump Condition = !(N ^ V)
0110	jrlt sign8	PC relative conditional jump Condition = N ^ V
0111	jrle sign8	PC relative conditional jump Condition = Z | (N ^ V)
1000	jrugt sign8	PC relative conditional jump Condition = !Z & !C
1001	jruge sign8	PC relative conditional jump Condition = !C
1010	jrult sign8	PC relative conditional jump Condition = C
1011	jrule sign8	PC relative conditional jump Condition = Z | C
1100	jreq sign8	PC relative conditional jump Condition = Z
1101	jrne sign8	PC relative conditional jump Condition = !Z
1110	call sign8	PC relative subroutine call
1111	jp sign8	PC relative unconditional jump

Class 1

This class contains data transfer instructions between a general-purpose register and memory, and logic/arithmetic operation instructions between general-purpose registers.

15		13	12 10	9 8	7 4	3 0
0	0	1	op1	op2	rb	rs/rd

op1	op2	Mnemonic	Function
000	00	ld.b %rd,[%rb]	Byte data transfer from memory to general-purpose register (with sign extension)
001	00	ld.ub %rd,[%rb]	Byte data transfer from memory to general-purpose register (with zero extension)
010	00	ld.h %rd,[%rb]	Half word data transfer from memory to general-purpose register (with sign extension)
011	00	ld.uh %rd,[%rb]	Half word data transfer from memory to general-purpose register (with zero extension)
100	00	ld.w %rd,[%rb]	Word data transfer from memory to general-purpose register
101	00	ld.b [%rb],%rs	Byte data transfer from general-purpose register to memory
110	00	ld.h [%rb],%rs	Half word data transfer from general-purpose register to memory
111	00	ld.w [%rb],%rs	Word data transfer from general-purpose register to memory
000	01	ld.b %rd,[%rb]+	Byte data transfer from memory to general-purpose register (with sign extension)
001	01	ld.ub %rd,[%rb]+	Byte data transfer from memory to general-purpose register (with zero extension)
010	01	ld.h %rd,[%rb]+	Half word data transfer from memory to general-purpose register (with sign extension)
011	01	ld.uh %rd,[%rb]+	Half word data transfer from memory to general-purpose register (with zero extension)
100	01	ld.w %rd,[%rb]+	Word data transfer from memory to general-purpose register
101	01	ld.b [%rb]+,%rs	Byte data transfer from general-purpose register to memory
110	01	ld.h [%rb]+,%rs	Half word data transfer from general-purpose register to memory
111	01	ld.w [%rb]+,%rs	Word data transfer from general-purpose register to memory

15		13	12 10	9 8	7 4	3 0
0	0	1	op1	op2	rs	rd

op1	op2	Mnemonic	Function
000	10	add %rd,%rs	Addition between general-purpose registers
001	10	sub %rd,%rs	Subtraction between general-purpose registers
010	10	cmp %rd,%rs	Comparison between general-purpose registers
011	10	ld.w %rd,%rs	Data transfer between general-purpose registers
100	10	and %rd,%rs	Logical product between general-purpose registers
101	10	or %rd,%rs	Logical sum between general-purpose registers
110	10	xor %rd,%rs	Exclusive OR between general-purpose registers
111	10	not %rd,%rs	Negation of general-purpose registers
***	11	reserved	

Class 2

This class contains data transfer instructions in the register indirect addressing with displacement mode using the SP.

15		13	12 10	9 4	3 0
0	1	0	op1	imm6	rs/rd

op1	Mnemonic		Function
000	ld.b	%rd,[%sp+imm6]	Byte data transfer from stack to general-purpose register (with sign extension)
001	ld.ub	%rd,[%sp+imm6]	Byte data transfer from stack to general-purpose register (with zero extension)
010	ld.h	%rd,[%sp+imm6]	Half word data transfer from stack to general-purpose register (with sign extension)
011	ld.uh	%rd,[%sp+imm6]	Half word data transfer from stack to general-purpose register (with zero extension)
100	ld.w	%rd,[%sp+imm6]	Word data transfer from stack to general-purpose register
101	ld.b	[%sp+imm6],%rs	Byte data transfer from general-purpose register to stack
110	ld.h	[%sp+imm6],%rs	Half word data transfer from general-purpose register to stack
111	ld.w	[%sp+imm6],%rs	Word data transfer from general-purpose register to stack

Class 3

This class contains data transfer and logic/arithmetic operation instructions using a 6-bit immediate data.

15		13	12 10	9 4	3 0
0	1	1	op1	imm6/sign6	rd

op1	Mnemonic		Function
000	add	%rd,imm6	Addition of immediate data to general-purpose register
001	sub	%rd,imm6	Subtraction of immediate data from general-purpose register
010	cmp	%rd,sign6	Comparison between general-purpose register and immediate data
011	ld.w	%rd,sign6	Immediate data transfer to general-purpose register
100	and	%rd,sign6	Logical product between general-purpose register and immediate data
101	or	%rd,sign6	Logical sum between general-purpose register and immediate data
110	xor	%rd,sign6	Exclusive OR between general-purpose register and immediate data
111	not	%rd,sign6	Negation of immediate data

Class 4

This class contains arithmetic instructions for the SP, shift/rotation instructions and division instructions.

15		13	12 10	9 0
1	0	0	op1	imm10

op1	Mnemonic		Function
000	add	%sp,imm10	Addition of immediate data to the SP
001	sub	%sp,imm10	Subtraction of immediate data from the SP

15		13	12 10	9 8	7 4	3 0
1	0	0	op1	op2	imm4/rs	rd

op1	op2	Mnemonic	Function
010	00	srl %rd,imm4	Logical shift to right (8-bit shift count with imm4)
011	00	sll %rd,imm4	Logical shift to left (8-bit shift count with imm4)
100	00	sra %rd,imm4	Arithmetical shift to right (8-bit shift count with imm4)
101	00	sla %rd,imm4	Arithmetical shift to left (8-bit shift count with imm4)
110	00	rr %rd,imm4	Rotation to right (8-bit shift count with imm4)
111	00	rl %rd,imm4	Rotation to left (8-bit shift count with imm4)
010	01	srl %rd,%rs	Logical shift to right (8-bit shift count with rs)
011	01	sll %rd,%rs	Logical shift to left (8-bit shift count with rs)
100	01	sra %rd,%rs	Arithmetical shift to right (8-bit shift count with rs)
101	01	sla %rd,%rs	Arithmetical shift to left (8-bit shift count with rs)
110	01	rr %rd,%rs	Rotation to right (8-bit shift count with rs)
111	01	rl %rd,%rs	Rotation to left (8-bit shift count with rs)

15		13	12 10	9 8	7 4	3 0
1	0	0	op1	op2	rs	rd

op1	op2	Mnemonic	Function
010	10	scan0 %rd,%rs	Bit search for "0"
011	10	scan1 %rd,%rs	Bit search for "1"
100	10	swap %rd,%rs	Swap in byte units
101	10	mirror %rd,%rs	Change of bit order in byte units
11*	10	reserved	
010	11	div0s %rs	Signed division 1st step
011	11	div0u %rs	Unsigned division 1st step
100	11	div1 %rs	Step division
101	11	div2s %rs	Data correction 1 for signed division
110	11	div3s	Data correction 2 for signed division
111	11	reserved	

Class 5

This class contains data transfer instructions between a general-purpose register and a special register or between general-purpose registers, bit operation instructions, multiplication instructions and a multiplication and accumulation instruction.

15		13	12 10	9 8	7 4	3 0
1	0	1	op1	op2	rs/ss	sd/rd

op1	op2	Mnemonic	Function
000	00	ld.w %sd,%rs	Word data transfer from general-purpose register to special register
001	00	ld.w %rd,%ss	Word data transfer from special register to general-purpose register

15		13	12 10	9 8	7 4	3 0
1	0	1	op1	op2	rb	0,imm3

op1	op2	Mnemonic	Function
010	00	btst [%rb],imm3	Bit test for memory data
011	00	bclr [%rb],imm3	Bit clear for memory data
100	00	bset [%rb],imm3	Bit set for memory data
101	00	bnot [%rb],imm3	Bit reversion for memory data

15		13	12 10	9 8	7 4	3 0
1	0	1	op1	op2	rs	rd

op1	op2	Mnemonic	Function
110	00	adc %rd,%rs	Addition with carry between general-purpose registers
111	00	sbc %rd,%rs	Subtraction with borrow between general-purpose registers
000	01	ld.b %rd,%rs	Byte data transfer between general-purpose registers (with sign extension)
001	01	ld.ub %rd,%rs	Byte data transfer between general-purpose registers (with zero extension)
010	01	ld.h %rd,%rs	Half word data transfer between general-purpose registers (with sign extension)
011	01	ld.uh %rd,%rs	Half word data transfer between general-purpose registers (with zero extension)
1**	01	reserved	
000	10	mlt.h %rd,%rs	Signed 16-bit multiplication
001	10	mltu.h %rd,%rs	Unsigned 16-bit multiplication
010	10	mlt.w %rd,%rs	Signed 32-bit multiplication
011	10	mltu.w %rd,%rs	Unsigned 32-bit multiplication
100	10	mac %rs	Multiplication and accumulation operation
101	10	reserved	
11*	10	reserved	
***	11	reserved	

Class 6

This class contains an immediate extension instruction only.

15		13	12 0
1	1	0	imm13

Mnemonic	Function
ext imm13	Immediate extension

Class 7

This class is reserved for expansion in future.

15		13	12 0
1	1	1	–

4.3 Reference for Individual Instruction

This section explains all the instructions in alphabetical order.

The explanations contain the following items.

Function:

Indicates the functions of the instruction.
"Standard" shows the function when the instruction is executed without extension.
"Extension 1" shows the function when the operand or immediate data is extended by one "ext" instruction described prior to the instruction.
"Extension 2" shows the function when the operand or immediate data is extended by two "ext" instructions described prior to the instruction.
If the "Extension" function is described as "Invalid", the instruction cannot be extended. And the previous "ext" instruction is invalidated.

Code:

Indicates the instruction code.

Flags:

Indicates the flag statuses after executing the instruction.

Mode:

Indicates the addressing mode. "Src" shows the addressing mode for the source and "Dst" shows it for the destination.

Clock:

Indicates the number of execution cycles for the instruction. The described cycle count is only when executing the instruction in the internal ROM and accessing data in the internal RAM.
See Section 3.2.2, "Number of instruction execution cycles", for the number of execution cycles when external memory is used or under other conditions and delay by interlock.

Description:

Explains the functions.

Example:

Shows an example of how to describe in assembler level.

Note:

Shows notes on using.

adc %rd, %rs

Function: Addition with carry

Standard: rd ← rd + rs + C

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	rd	
1	0	1	1	1	0	0	0	rs	rd	0xB800–0xB8FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Adds the contents of the rs register and C (carry) flag to the rd register.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Examples:

```
adc     %r0,%r1      ; r0 = r0 + r1 + C
```

Addition of 64-bit data

data 1 = {r2, r1}, data2 = {r4, r3}, result = {r2, r1}

```
add     %r1,%r3      ; Addition of the low-order word
adc     %r2,%r4      ; Addition of the high-order word
```

add %rd, %rs

Function: Addition

Standard: rd ← rd + rs
Extension 1: rd ← rs + imm13
Extension 2: rd ← rs + imm26

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	0	0	0	1	0	rs	rd	0x2200–0x22FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

```
add      %rd, %rs      ; rd ← rd + rs
```

Adds the contents of the rs register to the rd register.

(2) Extension 1

```
ext      imm13
add      %rd, %rs      ; rd ← rs + imm13
```

Adds the 13-bit immediate data (imm13) to the contents of the rs register, and then stores the results to the rd register. It does not change the contents of the rs register.

(3) Extension 2

```
ext      imm13         ; = imm26(25:13)
ext      imm13'        ; = imm26(12:0)
add      %rd, %rs      ; rd ← rs + imm26
```

Adds the 26-bit immediate data (imm26) to the contents of the rs register, and then stores the results to the rd register. The imm26 is zero-extended into 32 bits prior to the operation. It does not change the contents of the rs register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
add    %r0,%r0      ; r0 = r0 + r0

ext    0x1
ext    0x1fff
add    %r1,%r2      ; r1 = r2 + 0x3fff
```

add %rd, imm6

Function: Addition

Standard: rd ← rd + imm6
Extension 1: rd ← rd + imm19
Extension 2: rd ← rd + imm32

Code:

<table>
<tr><td colspan="3">15 13</td><td colspan="3">12 10</td><td>9 4</td><td>3 0</td><td></td></tr>
<tr><td colspan="3">class 3</td><td colspan="3">op1</td><td>imm6</td><td>rd</td><td></td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>imm6</td><td>rd</td><td>0x6000–0x63FF</td></tr>
<tr><td colspan="3">15</td><td>12</td><td>11</td><td></td><td>8 7 4</td><td>3 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Immediate data (unsigned)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
add     %rd, imm6     ; rd ← rd + imm6
```

Adds the 6-bit immediate data (imm6) to the rd register. The imm6 is zero-extended into 32 bits prior to the operation.

(2) Extension 1

```
ext     imm13         ; = imm19(18:6)
add     %rd, imm6     ; rd ← rd + imm19, imm6 = imm19(5:0)
```

Adds the 19-bit immediate data (imm19) extended with the "ext" instruction to the rd register. The imm19 is zero-extended into 32 bits prior to the operation.

(3) Extension 2

```
ext     imm13         ; = imm32(31:19)
ext     imm13'        ; = imm32(18:6)
add     %rd, imm6     ; rd ← rd + imm32, imm6 = imm32(5:0)
```

Adds the 32-bit immediate data (imm32) extended with the "ext" instructions to the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
add     %r0,0x3f     ; r0 = r0 + 0x3f

ext     0x1fff
ext     0x1fff
add     %r1,0x3f     ; r1 = r1 + 0xffffffff
```

add %sp, imm10

Function: Addition

Standard: sp ← sp + imm10 × 4

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9 0	
class 4			op1			imm10	
1	0	0	0	0	0	imm10	0x8000–0x83FF
15			12	11		8 7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Immediate data (unsigned)

Dst: Register direct (SP)

Clock: 1 cycle

Description: (1) Standard

Quadruples the 10-bit immediate data (imm10) and adds it to the stack pointer SP. The imm10 is zero-extended into 32 bits prior to the operation.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example:

```
add     %sp,0x100    ; sp = sp + 0x400
```

and %rd, %rs

Function: Logical product

Standard: rd ← rd & rs
Extension 1: rd ← rs & imm13
Extension 2: rd ← rs & imm26

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	1	0	0	1	0	rs	rd	0x3200–0x32FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
and        %rd, %rs      ; rd ← rd & rs
```

ANDs the contents of the rs and rd registers, and stores the results to the rd register.

(2) Extension 1

```
ext        imm13
and        %rd, %rs      ; rd ← rs & imm13
```

ANDs the contents of the rs register and the 13-bit immediate data (imm13), and stores the results to the rd register. The imm13 is zero-extended into 32 bits prior to the operation. It does not change the contents of the rs register.

(3) Extension 2

```
ext        imm13         ; = imm26(25:13)
ext        imm13'        ; = imm26(12:0)
and        %rd, %rs      ; rd ← rs & imm26
```

ANDs the contents of the rs register and the 26-bit immediate data (imm26), and stores the results to the rd register. The imm26 is zero-extended into 32 bits prior to the operation. It does not change the contents of the rs register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
and     %r0,%r0       ; r0 = r0 & r0

ext     0x1
ext     0x1fff
and     %r1,%r2       ; r1 = r2 & 0x00003fff
```

and %rd, sign6

Function: Logical product

Standard: rd ← rd & sign6
Extension 1: rd ← rd & sign19
Extension 2: rd ← rd & sign32

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td>10</td><td>9 … 4</td><td>3 … 0</td><td></td></tr>
<tr><td colspan="3">class 3</td><td colspan="3">op1</td><td>sign6</td><td>rd</td><td></td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>sign6</td><td>rd</td><td>0x7000–0x73FF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td colspan="2">11 … 8</td><td>7 … 4</td><td>3 … 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Immediate data (signed)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
and     %rd, sign6     ; rd ← rd & sign6
```

ANDs the contents of the rd register and the 6-bit immediate data (sign6), and stores the results to the rd register. The sign6 is sign-extended into 32 bits prior to the operation.

(2) Extension 1

```
ext     imm13          ; = sign19(18:6)
and     %rd, sign6     ; rd ← rd & sign19, sign6 = sign19(5:0)
```

ANDs the contents of the rd register and the 19-bit immediate data (sign19) extended with the "ext" instruction, and stores the results to the rd register. The sign19 is sign-extended into 32 bits prior to the operation.

(3) Extension 2

```
ext     imm13          ; = sign32(31:19)
ext     imm13'         ; = sign32(18:6)
and     %rd, sign6     ; rd ← rd & sign32, sign6 = sign32(5:0)
```

ANDs the contents of the rd register and the 32-bit immediate data (sign32) extended with the "ext" instructions, and stores the results to the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
and     %r0,0x3e     ; r0 = r0 & 0xfffffffe

ext     0x7ff
and     %r1,0x3f     ; r1 = r1 & 0x0001ffff
```

bclr [%rb], imm3

Function: Bit clear

Standard: B[rb](imm3) ← 0

Extension 1: B[rb + imm13](imm3) ← 0

Extension 2: B[rb + imm26](imm3) ← 0

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td>10</td><td>9</td><td>8</td><td>7</td><td></td><td></td><td>4</td><td>3</td><td></td><td></td><td>0</td><td></td></tr>
<tr><td colspan="3">class 5</td><td colspan="3">op1</td><td colspan="2">op2</td><td colspan="4">rb</td><td>0</td><td colspan="3">imm3</td><td></td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td colspan="4">rb</td><td>0</td><td colspan="3">imm3</td><td>0xAC00–0xACF7</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td></td><td>8</td><td>7</td><td></td><td></td><td>4</td><td>3</td><td></td><td></td><td>0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Immediate data (unsigned)

Dst: Register indirect (%rb = %r0–%r15)

Clock: 3 cycles

Description: (1) Standard

```
bclr      [%rb], imm3   ; B[rb](imm3) ← 0
```

Clears a data bit of the byte data in the address specified with the rb register. The 3-bit immediate data (imm3) specifies the bit number to be cleared (7–0).

(2) Extension 1

```
ext       imm13
bclr      [%rb], imm3   ; B[rb + imm13](imm3) ← 0
```

The "ext" instruction changes the addressing mode to register indirect addressing with displacement. The extended instruction clears the data bit specified with the imm3 in the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. It does not change the contents of the rb register.

(3) Extension 2

```
ext       imm13         ; = imm26(25:13)
ext       imm13'        ; = imm26(12:0)
bclr      [%rb], imm3   ; B[rb + imm26](imm3) ← 0
```

The "ext" instructions change the addressing mode to register indirect addressing with displacement. The extended instruction clears the data bit specified with the imm3 in the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. It does not change the contents of the rb register.

Examples:

```
ld.w    %r0,[%sp+0x10]     ; Sets the memory address to be accessed
                           ; to the R0 register.
bclr    [%r0],0x0          ; Clears Bit 0 of data in the specified
                           ; address.

ext     0x1
bclr    [%r0],0x7          ; Clears Bit 7 of data in the following
                           ; address.
```

bnot [%rb], imm3

Function: Bit negation

Standard: B[rb](imm3) ← !B[rb](imm3)
Extension 1: B[rb + imm13](imm3) ← !B[rb + imm13](imm3)
Extension 2: B[rb + imm26](imm3) ← !B[rb + imm26](imm3)

Code:

15		13	12		10	9	8	7 4	3	0	
class 5			op1			op2		rb	0	imm3	
1	0	1	1	0	1	0	0	rb	0	imm3	0xB400–0xB4F7
15			12	11			8	7 4	3	0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Immediate data (unsigned)
Dst: Register indirect (%rb = %r0–%r15)

Clock: 3 cycles

Description:

(1) Standard

```
bnot      [%rb], imm3   ; B[rb](imm3) ← !B[rb](imm3)
```

Reverses a data bit of the byte data in the address specified with the rb register. The 3-bit immediate data (imm3) specifies the bit number to be reversed (7–0).

(2) Extension 1

```
ext       imm13
bnot      [%rb], imm3   ; B[rb + imm13](imm3) ← !B[rb + imm13](imm3)
```

The "ext" instruction changes the addressing mode to register indirect addressing with displacement. The extended instruction reverses the data bit specified with the imm3 in the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. It does not change the contents of the rb register.

(3) Extension 2

```
ext       imm13         ; = imm26(25:13)
ext       imm13'        ; = imm26(12:0)
bnot      [%rb], imm3   ; B[rb + imm26](imm3) ← !B[rb + imm26](imm3)
```

The "ext" instructions change the addressing mode to register indirect addressing with displacement. The extended instruction reverses the data bit specified with the imm3 in the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. It does not change the contents of the rb register.

Examples:

```
ld.w    %r0,[%sp+0x10]    ; Sets the memory address to be accessed
                          ; to the R0 register.
bnot    [%r0],0x0         ; Reverses Bit 0 of data in the specified
                          ; address.

ext     0x1
bnot    [%r0],0x7         ; Reverses Bit 7 of data in the following
                          ; address.
```

brk

Function: Debugging exception

Standard: W[0x8(or 0x60008)] ← pc + 2, W[0xC(or 0x6000C)] ← r0, pc ← W[0x0(or 0x60000)]
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12			9	8	7	6	5	4	3			0	
class 0			op1				0	op2		0	0	–				
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0x0400
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	0	–	–	–	–

Clock: 10 cycles

Description: Calls a debugging handler routine.

The "brk" instruction stores the address that follows this instruction and the contents of the R0 register into the stack for debugging, then reads the vector for the debugging handler routine from the debugging vector address (0x0000000 or 0x0060000) and sets it to the PC. Thus the program branches to the debugging handler routine. Furthermore the CPU enters the debugging mode.

The "retd" instruction must be used for return from the debugging handler routine.

This instruction is provided for ICE control software. Do not use it in general programs.

Example:

```
brk                  ; Executes the debugging handler routine.
```

bset [%rb], imm3

Function: Bit set

Standard: B[rb](imm3) ← 1

Extension 1: B[rb + imm13](imm3) ← 1

Extension 2: B[rb + imm26](imm3) ← 1

Code:

15		13	12		10	9	8	7 4	3	0	
class 5			op1			op2		rb	0	imm3	
1	0	1	1	0	0	0	0	rb	0	imm3	0xB000–0xB0F7
15			12	11			8	7 4	3	0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Immediate data (unsigned)

Dst: Register indirect (%rb = %r0–%r15)

Clock: 3 cycles

Description: (1) Standard

bset [%rb], imm3 ; B[rb](imm3) ← 1

Sets a data bit of the byte data in the address specified with the rb register. The 3-bit immediate data (imm3) specifies the bit number to be set (7–0).

(2) Extension 1

ext imm13

bset [%rb], imm3 ; B[rb + imm13](imm3) ← 1

The "ext" instruction changes the addressing mode to register indirect addressing with displacement. The extended instruction sets the data bit specified with the imm3 in the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. It does not change the contents of the rb register.

(3) Extension 2

ext imm13 ; = imm26(25:13)

ext imm13' ; = imm26(12:0)

bset [%rb], imm3 ; B[rb + imm26](imm3) ← 1

The "ext" instructions change the addressing mode to register indirect addressing with displacement. The extended instruction sets the data bit specified with the imm3 in the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. It does not change the contents of the rb register.

Examples:

```
ld.w    %r0,[%sp+0x10]    ; Sets the memory address to be accessed
                          ; to the R0 register.
bset    [%r0],0x0         ; Sets Bit 0 of data in the specified
                          ; address.

ext     0x1
bset    [%r0],0x7         ; Sets Bit 7 of data in the following
                          ; address.
```

btst [%rb], imm3

Function: Bit test

Standard: Z flag ← 1 if B[rb](imm3) = 0 else Z flag ← 0

Extension 1: Z flag ← 1 if B[rb + imm13](imm3) = 0 else Z flag ← 0

Extension 2: Z flag ← 1 if B[rb + imm26](imm3) = 0 else Z flag ← 0

Code:

15		13	12		10	9	8	7 4	3	0	
class 5			op1			op2		rb	0	imm3	
1	0	1	0	1	0	0	0	rb	0	imm3	0xA800–0xA8F7
15			12	11			8	7 4	3	0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	–

Mode: Src: Immediate data (unsigned)

Dst: Register indirect (%rb = %r0–%r15)

Clock: 3 cycles

Description: (1) Standard

```
btst      [%rb], imm3   ; Z flag ← 1 if B[rb](imm3) = 0 else Z flag ← 0
```

Tests a data bit of the byte data in the address specified with the rb register and sets the Z (zero) flag if the bit is 0. The 3-bit immediate data (imm3) specifies the bit number to be tested (7–0).

(2) Extension 1

```
ext       imm13
btst      [%rb], imm3   ; Z flag ← 1 if B[rb + imm13](imm3) = 0 else Z flag ← 0
```

The "ext" instruction changes the addressing mode to register indirect addressing with displacement. The extended instruction tests the data bit specified with the imm3 in the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. It does not change the contents of the rb register.

(3) Extension 2

```
ext       imm13         ; = imm26(25:13)
ext       imm13'        ; = imm26(12:0)
btst      [%rb], imm3   ; Z flag ← 1 if B[rb + imm26](imm3) = 0 else Z flag ← 0
```

The "ext" instructions change the addressing mode to register indirect addressing with displacement. The extended instruction tests the data bit specified with the imm3 in the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. It does not change the contents of the rb register.

Example:

```
ld.w    %r0,[%sp+0x10]    ; Sets the memory address to be accessed
                          ; to the R0 register.
btst    [%r0],0x7         ; Tests Bit 7 of data in the specified
                          ; address.
jreq    POSITIVE          ; Jumps if the bit is 0.
```

call %rb / call.d %rb

Function: Subroutine call

Standard: sp ← sp - 4, W[sp] ← pc + 2, pc ← rb

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12			9	8	7	6	5	4	3 0	
class 0			op1				d	op2		0	0	rb	
0	0	0	0	0	1	1	d	0	0	0	0	rb	0x0600–0x060F, 0x070F–0x070F
15			12	11			8	7			4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Register direct (%rb = %r0–%r15)

Clock:

call: 3 cycles

call.d: 2 cycles

Description:

(1) Standard

call %rb

Stores the address of the following instruction into the stack, then sets the contents of the rb register to the PC for calling the subroutine that starts from the address set to the PC. The LSB of the rb register is invalid and is always handled as 0. When the "ret" instruction is executed in the subroutine, the program flow returns to the instruction following the "call" instruction.

(2) Delayed branch (d bit = 1)

call.d %rb

When "call.d" is specified, the d bit in the instruction code is set and the following instruction becomes a delayed instruction.

The delayed instruction is executed before branching to the subroutine. Therefore the address (PC+4) of the instruction that follows the delayed instruction is stored into the stack as the return address.

When the "call.d" instruction is executed, interrupts and exceptions cannot occur because traps are masked between the "call.d" and delayed instructions.

Example:

```
call    %r0          ; Calls the subroutine that starts from the
                     ; address stored in the R0 register.
```

Note: When using the "call.d" instruction (delayed branch), the next instruction must be an instruction available for a delayed instruction. Be aware that the operation is undefined if another instruction is executed. See the instruction list in the Appendix for available instructions.

call sign8 / call.d sign8

Function: Subroutine call

Standard: sp ← sp - 4, W[sp] ← pc + 2, pc ← pc + sign8 × 2

Extension 1: sp ← sp - 4, W[sp] ← pc + 2, pc ← pc + sign22

Extension 2: sp ← sp - 4, W[sp] ← pc + 2, pc ← pc + sign32

Code:

15		13	12			9	8	7 0	
class 0			op1				d	sign8	
0	0	0	1	1	1	0	d	sign8	0x1C00–0x1DFF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock:

call: 3 cycles

call.d: 2 cycles

Description:

(1) Standard

```
call    sign8       ; = "call  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

Stores the address of the following instruction into the stack, then doubles the signed 8-bit immediate data (sign8) and adds it to the PC for calling the subroutine that starts from the address. The sign8 specifies a half word address in 16-bit units. When the "ret" instruction is executed in the subroutine, the program flow returns to the instruction following the "call" instruction.

The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext     imm13       ; = sign22(21:9)
call    sign8       ; = "call  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement into 22 bits using its 13-bit immediate data (imm13). The 22-bit displacement is sign-extended and added to the PC.

The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext     imm13       ; imm13(12:3)= sign32(31:22)
ext     imm13'      ; = sign(21:9)
call    sign8       ; = "call  sign32", sign9 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement into 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space.

(4) Delayed branch (d bit = 1)

```
call.d  sign8
```

When "call.d" is specified, the d bit in the instruction code is set and the following instruction becomes a delayed instruction.

The delayed instruction is executed before branching to the subroutine. Therefore the address (PC+4) of the instruction that follows the delayed instruction is stored into the stack as the return address.

When the "call.d" instruction is executed, interrupts and exceptions cannot occur because traps are masked between the "call.d" and delayed instructions.

Example:

```
ext     0x1fff
call    0x0           ; Calls the subroutine that starts from the
                      ; address specified by PC-0x200.
```

Note: When using the "call.d" instruction (delayed branch), the next instruction must be an instruction available for a delayed instruction. Be aware that the operation is undefined if another instruction is executed. See the instruction list in the Appendix for available instructions.

cmp %rd, %rs

Function: Comparison

Standard: rd - rs

Extension 1: rs - imm13

Extension 2: rs - imm26

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	0	1	0	1	0	rs	rd	0x2A00–0x2AFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

```
cmp     %rd, %rs     ; rd - rs
```

Subtracts the contents of the rs register from the contents of the rd register, and sets or resets the flags (C, V, Z and N) according to the results. It does not change the contents of the rd register.

(2) Extension 1

```
ext     imm13
cmp     %rd, %rs     ; rs - imm13
```

Subtracts the 13-bit immediate data (imm13) from the contents of the rs register, and sets or resets the flags (C, V, Z and N) according to the results. It does not change the contents of the rd and rs registers.

(3) Extension 2

```
ext     imm13        ; = imm26(25:13)
ext     imm13'       ; = imm26(12:0)
cmp     %rd, %rs     ; rs - imm26
```

Subtracts the 26-bit immediate data (imm26) from the contents of the rs register, and sets or resets the flags (C, V, Z and N) according to the results. It does not change the contents of the rd and rs registers.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
cmp     %r0,%r1      ; Changes the flags according to the results of
                     ; r0 - r1.

ext     0x1
ext     0x1fff
cmp     %r1,%r2      ; Changes the flags according to the results of
                     ; r2 - 0x3ff.
```

cmp %rd, sign6

Function: Comparison

Standard: rd - sign6
Extension 1: rd - sign19
Extension 2: rd - sign32

Code:

15		13	12		10	9 4	3 0	
class 3			op1			sign6	rd	
0	1	1	0	1	0	sign6	rd	0x6800–0x6BFF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Immediate data (signed)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
cmp      %rd, sign6     ; rd - sign6
```

Subtracts the signed 6-bit immediate data (sign6) from the contents of the rd register, and sets or resets the flags (C, V, Z and N) according to the results. The sign6 is sign-extended into 32 bits prior to the operation. It does not change the contents of the rd register.

(2) Extension 1

```
ext      imm13          ; = sign19(18:6)
cmp      %rd, sign6     ; rd - sign19, sign6 = sign19(5:0)
```

Subtracts the signed 19-bit immediate data (sign19) from the contents of the rd register, and sets or resets the flags (C, V, Z and N) according to the results. The sign19 is sign-extended into 32 bits prior to the operation. It does not change the contents of the rd register.

(3) Extension 2

```
ext      imm13          ; = sign32(31:19)
ext      imm13'         ; = sign32(18:6)
cmp      %rd, sign6     ; rd - sign32, imm6 = sign32(5:0)
```

Subtracts the signed 32-bit immediate data (sign32) extended with the "ext" instruction from the contents of the rd register, and sets or resets the flags (C, V, Z and N) according to the results. It does not change the contents of the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
cmp     %r0,0x3f      ; Changes the flags according to the results of
                      ; r0 - 0x3f.

ext     0x1fff
ext     0x1fff
cmp     %r1,0x3f      ; Changes the flags according to the results of
                      ; r1 - 0xffffffff.
```

div0s %rs *(option)*

Function: Signed division 1st step

Standard: Initialization for division

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7			4	3			0	
class 4			op1			op2		rs				rd				
1	0	0	0	1	0	1	1	rs				0	0	0	0	0x8B00–0x8BF0
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	↔	–	–	–	–	↔

Mode: Register direct (%rs = %r0–%r15)

Clock: 1 cycle

Description: When performing a signed division, first execute the "div0s" instruction after setting the dividend to the ALR and the divisor to the rs register. The "div0s" instruction initializes the register and flags as follows:

1) Extends the dividend in the ALR into 64 bits with a sign and sets it in {AHR, ALR}.
2) Sets the sign bit of the dividend (MSB of ALR) to the DS flag in the PSR.
3) Sets the sign bit of the divisor (MSB of the rs register) to the N flag in the PSR.

Therefore, it is necessary that the dividend and divisor in the ALR and the rs register have been sign-extended into 32 bits.

The "div1" instruction should be executed after executing the "div0s" instruction. Then correct the results using the "div2s" and "div3s" instructions in signed division.

Example: Signed division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w    %alr,%r0     ; Set the dividend to the ALR.
div0s   %r1          ; Initialization for signed division.
div1    %r1          ; Executing div1 32 times.
 :       :
div1    %r1
div2s   %r1          ; Correction 1
div3s                ; Correction 2
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR.

Note: A zero-division exception occurs if the "div0s" instruction is executed by setting the rs register to 0.

Up to 32-bit data can be used for both dividends and divisors.

This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

div0u %rs *(option)*

Function: Unsigned division 1st step

Standard: Initialization for division
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3			0	
class 4			op1			op2		rs	rd				
1	0	0	0	1	1	1	1	rs	0	0	0	0	0x8F00–0x8FF0
15			12	11			8	7 4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	0	–	–	–	–	0

Mode: Register direct (%rs = %r0–%r15)

Clock: 1 cycle

Description: When performing an unsigned division, first execute the "div0u" instruction after setting the dividend to the ALR and the divisor to the rs register. The "div0u" instruction initializes the register and flags as follows:

1) Clears the AHR to 0.
2) Resets the DS flag in the PSR to 0.
3) Resets the N flag in the PSR to 0.

The "div1" instruction should be executed after executing the "div0u" instruction. In unsigned division, it is not necessary to correct the division results of the "div1" instruction.

Example: Unsigned division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w    %alr,%r0     ; Sets the dividend to the ALR.
div0u   %r1          ; Initialization for unsigned division.
div1    %r1          ; Executing div1 32 times.
 :       :
div1    %r1
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR.

Note: A zero-division exception occurs if the "div0u" instruction is executed by setting the rs register to 0.
Up to 32-bit data can be used for both dividends and divisors.
This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

div1 %rs *(option)*

Function: Division

Standard: Step division

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7			4	3			0	
class 4			op1			op2		rs				rd				
1	0	0	1	0	0	1	1	rs				0	0	0	0	0x9300–0x93F0
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Register direct (%rs = %r0–%r15)

Clock: 1 cycle

Description: The "div1" instruction executes a step division and is used for both signed division and unsigned division. This instruction must be executed a number of times according to the data size of the dividend after finishing the initialization by the "div0s" (for signed division) or "div0u" (for unsigned division) instruction. For example, execute 32 "div1" instructions for 32 bits ÷ 32 bits, and 16 for 16 bits ÷ 16 bits.

One "div1" instruction step performs the following process:

1) Shifts the 64-bit data (dividend) in {AHR, ALR} 1 bit to the left (to upper side). (ALR(0) = 0)

2) Adds rs to the AHR or subtracts rs from the AHR and modifies the AHR and the ALR according to the results.

The addition/subtraction uses the 33-bit data created by extending the contents of the AHR with the DS flag as the sign bit and the 33-bit data created by extending the contents of the rs register with the N flag as the sign bit.

The process varies according to the DS and N flags in the PSR as shown below. "tmp(32)" in the explanation indicates the bit-33 value of the addition/subtraction results.

In the case of DS = 0 (dividend is positive) and N = 0 (divisor is positive):

2-1) Executes tmp = {0, AHR} - {0, rs}

2-2) If tmp(32) = 0, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 1, terminates without changing the AHR and ALR.

In the case of DS = 1 (dividend is negative) and N = 0 (divisor is positive):

2-1) Executes tmp = {1, AHR} + {0, rs}

2-2) If tmp(32) = 1, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 0, terminates without changing the AHR and ALR.

In the case of DS = 0 (dividend is positive) and N = 1 (divisor is negative):

2-1) Executes tmp = {0, AHR} + {1, rs}

2-2) If tmp(32) = 0, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 1, terminates without changing the AHR and ALR.

In the case of DS = 1 (dividend is negative) and N = 1 (divisor is negative):

2-1) Executes tmp = {1, AHR} - {1, rs}

2-2) If tmp(32) = 1, executes AHR = tmp(31:0) and ALR(0) = 1 and then terminates.
If tmp(32) = 0, terminates without changing the AHR and ALR.

In unsigned division, the results are obtained from the following registers by executing the necessary "div1" instruction steps.

The results of unsigned division: ALR = Quotient, AHR = Remainder

In signed division, it is necessary to correct the results using the "div2s" and "div3s" instructions.

Examples: Unsigned division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w    %alr,%r0     ; Sets the dividend to the ALR.
div0u   %r1          ; Initialization for unsigned division.
div1    %r1          ; Executing div1 32 times.
 :       :
div1    %r1
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR.

Signed division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w    %alr,%r0     ; Set the dividend to the ALR.
div0s   %r1          ; Initialization for signed division.
div1    %r1          ; Executing div1 32 times.
 :       :
div1    %r1
div2s   %r1          ; Correction 1
div3s                ; Correction 2
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR.

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

div2s %rs *(option)*

Function: Correction step 1 for signed division results

Standard: Correction process for the execution results of signed division
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3			0
class 4			op1			op2		rs	rd			
1	0	0	1	0	1	1	1	rs	0	0	0	0
15			12	11			8	7 4	3			0

0x9700–0x97F0

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Register direct (%rs = %r0–%r15)

Clock: 1 cycle

Description: The "div2s" instruction corrects the results of signed division. It is not necessary to execute the "div2s" instruction in unsigned division.

When the dividend is a negative number and zero results in a division step (execution of div1), the remainder (AHR) after completing all the steps may be the same as the divisor and the quotient (AHR) may be 1 short from the actual absolute value. The "div2s" instruction corrects such results. The "div2s" instruction operates as follows:

In the case of DS = 0 (dividend is positive):
This problem does not occur when the dividend is a positive number, so the "div2s" instruction terminates without any execution (same as the "nop" instruction).

In the case of DS = 1 (dividend is negative):

1) If N = 0 (divisor is positive), executes tmp = AHR + rs
   If N = 1 (divisor is negative), executes tmp = AHR - rs
2) According to the results of step 1).
   If tmp is zero, executes AHR = tmp(31:0) and ALR = ALR + 1 and then terminates.
   If tmp is not zero, terminates without changing the AHR and ALR.

Example: Signed division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w    %alr,%r0     ; Set the dividend to the ALR.
div0s   %r1          ; Initialization for signed division.
div1    %r1          ; Executing div1 32 times.
 :       :
div1    %r1
div2s   %r1          ; Correction 1
div3s                ; Correction 2
```

Executing the above instructions stores the quotient into the ALR and the remainder into the AHR.

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

div3s *(option)*

Function: Correction step 2 for signed division results

Standard: Correction process for the execution results of signed division
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3			0	
class 4			op1			op2		rs	rd				
1	0	0	1	1	0	1	1	rs	0	0	0	0	0x9B00–0x9BF0
15			12	11			8	7 4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Clock: 1 cycle

Description: The "div3s" instruction corrects the results of signed division. It is not necessary to execute the "div3s" instruction in unsigned division.

Step division always stores a positive number of quotient into the ALR. When the signs of the dividend and divisor are different, the results must be a negative number. The "div3s" instruction corrects the sign in such cases.
The "div2s" instruction operates as follows:

In the case of DS = N (dividend and divisor have the same sign):
This problem does not occur, so the "div3s" instruction terminates without any execution (same as the "nop" instruction).

In the case of DS = !N (dividend and divisor have different sign):
Reverses the sign bit of the ALR (quotient).

In signed division, the results are obtained from the following registers after executing the "div2s" and "div3s" instructions.
The results of unsigned division: ALR = Quotient, AHR = Remainder

Example: Signed division (32 bits ÷ 32 bits)

When the dividend has been set to the R0 register and the divisor to the R1 register:

```
ld.w    %alr,%r0     ; Set the dividend to the ALR.
div0s   %r1          ; Initialization for signed division.
div1    %r1          ; Executing div1 32 times.
 :       :
div1    %r1
div2s   %r1          ; Correction 1
div3s                ; Correction 2
```

Executing the above instructions store the quotient into the ALR and the remainder into the AHR.

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

ext imm13

Function: Immediate extension

Standard: Extends the immediate data/operand of the following instruction.

Extension 1: Up to two "ext" instructions can be used sequentially.

Extension 2: Invalid

Code:

15		13	12 … 0	
class 6			imm13	
1	1	0	imm13	0xC000–0xDFFF
15			12 11 8 7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Immediate data (unsigned)

Clock: 1 cycle

Description: Extends the immediate data or operand of the following instruction.
When extending an immediate data, the immediate data in the "ext" instruction will be placed on the high-order side and the immediate data in the target instruction to be extended is placed on the low-order side.

Up to two "ext" instructions can be used sequentially. In this case, the immediate data in the first "ext" instruction is placed on the most upper part. If three or more "ext" instructions are described sequentially, only two instructions, the first and the last (prior to the target instruction) are effective and the middles are invalidated.
See descriptions of each instruction for the extension contents and the usage.
Traps except for reset and address error are masked by the hardware while executing the "ext" instruction and the following target instruction, and they do not occur.

Example:

```
ext     0x1000       ; Valid
ext     0x1          ; Invalid
ext     0x1fff       ; Valid
add     %r1,0x3f     ; r1 = r1 + 0x8007ffff
```

Note: When a load instruction that transfers data between memory and a register follows the "ext" instruction, an address error exception may occur before executing the load instruction (if the address that is specified with the immediate data in the "ext" instruction as the displacement is not a boundary address according to the transfer data size). When an address error occurs, the trap processing saves the address of the load instruction into the stack as the return address. If the trap handler routine is returned by simply executing the "reti" instruction, the previous "ext" instruction is invalidated. Therefore, it is necessary to modify the return address in that case.

halt

Function: HALT

Standard: Sets the CPU to HALT mode.
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12			9	8	7	6	5	4	3			0	
class 0			op1				0	op2		0	0	–				
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0x0080
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Clock: 1 cycle

Description: Sets the CPU to HALT mode.
In HALT mode, the CPU stops operating, so current consumption can be reduced.
On-chip peripheral circuits operate in HALT mode.
HALT mode is canceled by an interrupt. When HALT mode is canceled, the program flow returns to the next instruction of the "halt" instruction after executing the interrupt handler routine.

Example:

```
halt                 ; Sets the CPU in HALT mode.
```

int imm2

Function: Software exception

Standard: sp ← sp - 4, W[sp] ← pc + 2, sp ← sp - 4, W[sp] ← psr, pc ← Software exception vector
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12			9	8	7	6	5	4	3		0	
class 0			op1				0	op2		0	0	imm2			
0	0	0	0	0	1	0	0	1	0	0	0	0	0	imm2	0x0480–0x0483
15			12	11			8	7			4	3		0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	1	–	–	–	–

Mode: Immediate data (unsigned)

Clock: 10 cycles

Description: Generates a software exception.

The "int" instruction saves the address of the next instruction and the contents of the PSR into the stack, then reads the software exception vector from the trap table and sets it to the PC. By this processing, the program flow branches to the specified software exception handler routine.

The E0C33000 supports four types of software exceptions and the software exception number (0 to 3) is specified by the 2-bit immediate data (imm2).

	imm2	Vector address
Software exception 0	0	Base + 48
Software exception 1	1	Base + 52
Software exception 2	2	Base + 56
Software exception 3	3	Base + 60

The Base is the trap table beginning address. It is address 0x0080000 for the system that boots from the internal ROM (BTA3 terminal is high) or address 0x0C00000 for the system that boots from the external ROM (BTA3 terminal is low).

The "reti" instruction should be used for return from the handler routine.

Example:

```
int     2     ; Executes the software exception 2 handler routine.
```

jp %rb / jp.d %rb

Function: Unconditional jump

Standard: pc ← rb

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12			9	8	7	6	5	4	3 0
class 0			op1				d	op2		0	0	rb
0	0	0	0	0	1	1	d	1	0	0	0	rb
15			12	11			8	7			4	3 0

0x0680–0x068F, 0x0780–0x078F

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Register direct (%rb = %r0–%r15)

Clock:

jp: 2 cycles

jp.d: 1 cycle

Description: (1) Standard

jp %rb

Loads the contents of the rb register to the PC for branching the program flow to the address. The LSB of the rb register is ignored and is always handled as 0.

(2) Delayed branch (d bit = 1)

jp.d %rb

The "jp.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.

Traps that may occur between the "jp.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
jp      %r0    ; Jumps to the address specified by the R0 register.
```

Note: When using the "jp.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jp sign8 / jp.d sign8

Function: Unconditional PC relative jump

Standard: pc ← pc + sign8 × 2
Extension 1: pc ← pc + sign22
Extension 2: pc ← pc + sign32

Code:

15		13	12			9	8	7 0	
class 0			op1				d	sign8	
0	0	0	1	1	1	1	d	sign8	0x1E00–0x1FFF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jp: 2 cycles
jp.d: 1 cycle

Description: (1) Standard

```
jp      sign8       ; = "jp  sign9", sign8 = sign9(8:1), sign9(0)=0
```

Doubles the signed 8-bit immediate data (sign8) and adds it to the PC. The program flow branches to the address. The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext     imm13       ; = sign22(21:9)
jp      sign8       ; = "jp  sign22", sign8 = sign22(8:1), sign22(0)=0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext     imm13       ; imm13(12:3)= sign32(31:22)
ext     imm13'      ; = sign32(21:9)
jp      sign8       ; = "jp  sign32", sign8 = sign32(8:1), sign32(0)=0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jp.d    sign8
```

The "jp.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jp.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
ext     0x8
ext     0x0
jp      0x80     ; Jumps to the address specified by PC+0x400100.
```

Note: When using the "jp.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jreq sign8 / jreq.d sign8

Function: Conditional PC relative jump

Standard: pc ← pc + sign8 × 2 if Z is true

Extension 1: pc ← pc + sign22 if Z is true

Extension 2: pc ← pc + sign32 if Z is true

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td></td><td>9</td><td>8</td><td>7 0</td><td></td></tr>
<tr><td colspan="3">class 0</td><td colspan="4">op1</td><td>d</td><td>sign8</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>d</td><td>sign8</td><td>0x1800–0x19FF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td></td><td>8</td><td>7 4 3 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jreq: 1 cycle (when not branched), 2 cycles (when branched)

jreq.d: 1 cycle

Description:

(1) Standard

```
jreq      sign8       ; = "jreq  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

If the condition below has been met, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- Z flag = 1 (e.g. "A = B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.

The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext       imm13       ; = sign22(21:9)
jreq      sign8       ; = "jreq  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext       imm13       ; imm13(12:3)= sign32(31:22)
ext       imm13'      ; = sign32(21:9)
jreq      sign8       ; = "jreq  sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jreq.d    sign8
```

The "jreq.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.

Traps that may occur between the "jreq.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp    %r0,%r1
jreq   0x2          ; Skips the next instruction if r1 = r0.
```

Note: When using the "jreq.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrge sign8 / jrge.d sign8

Function: Conditional PC relative jump (for judgment of signed operation results)

Standard: pc ← pc + sign8 × 2 if !(N^V) is true
Extension 1: pc ← pc + sign22 if !(N^V) is true
Extension 2: pc ← pc + sign32 if !(N^V) is true

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td></td><td>9</td><td>8</td><td>7 0</td><td></td></tr>
<tr><td colspan="3">class 0</td><td colspan="4">op1</td><td>d</td><td>sign8</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>d</td><td>sign8</td><td>0x0A00–0x0BFF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td></td><td>8</td><td>7 4 3 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrge: 1 cycle (when not branched), 2 cycles (when branched)
jrge.d: 1 cycle

Description: (1) Standard

jrge sign8 ; = "jrge sign9", sign8 = sign9(8:1), sign9(0) = 0

If the condition below has been met by a signed operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

• N flag = V flag (e.g. "A ≥ B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.

The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

ext imm13 ; = sign22(21:9)
jrge sign8 ; = "jrge sign22", sign8 = sign22(8:1), sign22(0) = 0

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

ext imm13 ; imm13(12:3)= sign32(31:22)
ext imm13' ; = sign32(21:9)
jrge sign8 ; = "jrge sign32", sign8 = sign32(8:1), sign32(0) = 0

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

jrge.d sign8

The "jrge.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.

Traps that may occur between the "jrge.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1       ; r0 and r1 contain signed data.
jrge    0x2           ; Skips the next instruction if r0 ≥ r1.
```

Note: When using the "jrge.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrgt sign8 / jrgt.d sign8

Function: Conditional PC relative jump (for judgment of signed operation results)

Standard: pc ← pc + sign8 × 2 if !Z&!(N^V) is true
Extension 1: pc ← pc + sign22 if !Z&!(N^V) is true
Extension 2: pc ← pc + sign32 if !Z&!(N^V) is true

Code:

15		13	12			9	8	7 … 0	
class 0			op1				d	sign8	
0	0	0	0	1	0	0	d	sign8	0x0800–0x09FF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrgt: 1 cycle (when not branched), 2 cycles (when branched)
jrgt.d: 1 cycle

Description: (1) Standard

```
jrgt     sign8       ; = "jrgt  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

If the condition below has been met by a signed operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- Z flag = 0 and N flag = V flag (e.g. "A > B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext      imm13       ; = sign22(21:9)
jrgt     sign8       ; = "jrgt  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext      imm13       ; imm13(12:3)= sign32(31:22)
ext      imm13'      ; = sign32(21:9)
jrgt     sign8       ; = "jrgt  sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jrgt.d   sign8
```

The "jrgt.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrgt.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1      ; r0 and r1 contain signed data.
jrgt    0x2          ; Skips the next instruction if r0 > r1.
```

Note: When using the "jrgt.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrle sign8 / jrle.d sign8

Function: Conditional PC relative jump (for judgment of signed operation results)

Standard: pc ← pc + sign8 × 2 if Z | (N^V) is true
Extension 1: pc ← pc + sign22 if Z | (N^V) is true
Extension 2: pc ← pc + sign32 if Z | (N^V) is true

Code:

15		13	12			9	8	7			0	
class 0			op1				d	sign8				
0	0	0	0	1	1	1	d	sign8				0x0E00–0x0FFF
15			12	11			8	7	4	3	0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock:
jrle: 1 cycle (when not branched), 2 cycles (when branched)
jrle.d: 1 cycle

Description: (1) Standard

```
jrle      sign8        ; = "jrle  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

If the condition below has been met by a signed operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- Z flag = 1 or N flag ≠ V flag (e.g. "A ≤ B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext       imm13        ; = sign22(21:9)
jrle      sign8        ; = "jrle  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext       imm13        ; imm13(12:3)= sign32(31:22)
ext       imm13'       ; = sign32(21:9)
jrle      sign8        ; = "jrle  sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jrle.d    sign8
```

The "jrle.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrle.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1       ; r0 and r1 contain signed data.
jrle    0x2           ; Skips the next instruction if r0 ≤ r1.
```

Note: When using the "jrle.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrlt sign8 / jrlt.d sign8

Function: Conditional PC relative jump (for judgment of signed operation results)

Standard: pc ← pc + sign8 × 2 if N^V is true
Extension 1: pc ← pc + sign22 if N^V is true
Extension 2: pc ← pc + sign32 if N^V is true

Code:

15		13	12			9	8	7 … 0	
class 0			op1				d	sign8	
0	0	0	0	1	1	0	d	sign8	0x0C00–0x0DFF
15			12	11			8	7 … 4 3 … 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrlt: 1 cycle (when not branched), 2 cycles (when branched)
jrlt.d: 1 cycle

Description: (1) Standard

```
jrlt    sign8    ; = "jrlt  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

If the condition below has been met by a signed operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- N flag ≠ V flag (e.g. "A < B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext     imm13    ; = sign22(21:9)
jrlt    sign8    ; = "jrlt  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext     imm13    ; imm13(12:3)= sign32(31:22)
ext     imm13'   ; = sign32(21:9)
jrlt    sign8    ; = "jrlt  sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jrlt.d  sign8
```

The "jrlt.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrlt.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1       ; r0 and r1 contain signed data.
jrlt    0x2           ; Skips the next instruction if r0 < r1.
```

Note: When using the "jrlt.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrne sign8 / jrne.d sign8

Function: Conditional PC relative jump

Standard: pc ← pc + sign8 × 2 if !Z is true
Extension 1: pc ← pc + sign22 if !Z is true
Extension 2: pc ← pc + sign32 if !Z is true

Code:

15		13	12			9	8	7 0	
class 0			op1				d	sign8	
0	0	0	1	1	0	1	d	sign8	0x1A00–0x1BFF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrne: 1 cycle (when not branched), 2 cycles (when branched)
jrne.d: 1 cycle

Description: (1) Standard

```
jrne      sign8       ; = "jrne  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

If the condition below has been met, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- Z flag = 0 (e.g. "A ≠ B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext       imm13       ; = sign22(21:9)
jrne      sign8       ; = "jrne  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext       imm13       ; imm13(12:3)= sign32(31:22)
ext       imm13'      ; = sign32(21:9)
jrne      sign8       ; = "jrne  sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jrne.d    sign8
```

The "jrne.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrne.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1
jrne    0x2          ; Skips the next instruction if r1 ≠ r0.
```

Note: When using the "jrne.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jruge sign8 / jruge.d sign8

Function: Conditional PC relative jump (for judgment of unsigned operation results)

Standard: pc ← pc + sign8 × 2 if !C is true
Extension 1: pc ← pc + sign22 if !C is true
Extension 2: pc ← pc + sign32 if !C is true

Code:

15		13	12			9	8	7 0	
class 0			op1				d	sign8	
0	0	0	1	0	0	1	d	sign8	0x1200–0x13FF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock:

jruge: 1 cycle (when not branched), 2 cycles (when branched)
jruge.d: 1 cycle

Description:

(1) Standard

jruge sign8 ; = "jruge sign9", sign8 = sign9(8:1), sign9(0) = 0

If the condition below has been met by an unsigned operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- C flag = 0 (e.g. "A ≥ B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

ext imm13 ; = sign22(21:9)
jruge sign8 ; = "jruge sign22", sign8 = sign22(8:1), sign22(0) = 0

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

ext imm13 ; imm13(12:3)= sign32(31:22)
ext imm13' ; = sign32(21:9)
jruge sign8 ; = "jruge sign32", sign8 = sign32(8:1), sign32(0) = 0

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

jruge.d sign8

The "jruge.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jruge.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1      ; r0 and r1 contain unsigned data.
jruge   0x2          ; Skips the next instruction if r0 ≥ r1.
```

Note: When using the "jruge.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrugt sign8 / jrugt.d sign8

Function: Conditional PC relative jump (for judgment of unsigned operation results)

Standard: pc ← pc + sign8 × 2 if !Z&!C is true
Extension 1: pc ← pc + sign22 if !Z&!C is true
Extension 2: pc ← pc + sign32 if !Z&!C is true

Code:

15		13	12			9	8	7 … 0	
class 0			op1				d	sign8	
0	0	0	1	0	0	0	d	sign8	0x1000–0x11FF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrugt: 1 cycle (when not branched), 2 cycles (when branched)
jrugt.d: 1 cycle

Description: (1) Standard

jrugt sign8 ; = "jrugt sign9", sign8 = sign9(8:1), sign9(0) = 0

If the condition below has been met by an unsigned operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- Z flag = 0 and C flag = 0 (e.g. "A > B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

ext imm13 ; = sign22(21:9)
jrugt sign8 ; = "jrugt sign22", sign8 = sign22(8:1), sign22(0) = 0

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

ext imm13 ; imm13(12:3)= sign32(31:22)
ext imm13' ; = sign32(21:9)
jrugt sign8 ; = "jrugt sign32", sign8 = sign32(8:1), sign32(0) = 0

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

jrugt.d sign8

The "jrugt.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrugt.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1       ; r0 and r1 contain unsigned data.
jrugt   0x2           ; Skips the next instruction if r0 > r1.
```

Note: When using the "jrugt.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrule sign8 / jrule.d sign8

Function: Conditional PC relative jump (for judgment of unsigned operation results)

Standard: pc ← pc + sign8 × 2 if Z | C is true
Extension 1: pc ← pc + sign22 if Z | C is true
Extension 2: pc ← pc + sign32 if Z | C is true

Code:

15		13	12			9	8	7 0	
class 0			op1				d	sign8	
0	0	0	1	0	1	1	d	sign8	0x1600–0x17FF
15		12	11				8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrule: 1 cycle (when not branched), 2 cycles (when branched)
jrule.d: 1 cycle

Description: (1) Standard

```
jrule     sign8       ; = "jrule  sign9", sign8 = sign9(8:1), sign9(0) = 0
```

If the condition below has been met by an unsigned operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- Z flag = 1 or C flag = 1 (e.g. "A ≤ B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

```
ext       imm13       ; = sign22(21:9)
jrule     sign8       ; = "jrule  sign22", sign8 = sign22(8:1), sign22(0) = 0
```

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

```
ext       imm13       ; imm13(12:3)= sign32(31:22)
ext       imm13'      ; = sign32(21:9)
jrule     sign8       ; = "jrule  sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

```
jrule.d   sign8
```

The "jrule.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrule.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1      ; r0 and r1 contain unsigned data.
jrule   0x2          ; Skips the next instruction if r0 ≤ r1.
```

Note: When using the "jrule.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

jrult sign8 / jrult.d sign8

Function: Conditional PC relative jump (for judgment of unsigned operation results)

Standard: pc ← pc + sign8 × 2 if C is true
Extension 1: pc ← pc + sign22 if C is true
Extension 2: pc ← pc + sign32 if C is true

Code:

15		13	12			9	8	7 0	
class 0			op1				d	sign8	
0	0	0	1	0	1	0	d	sign8	0x1400–0x15FF
15			12	11			8	7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Signed PC relative

Clock: jrult: 1 cycle (when not branched), 2 cycles (when branched)
jrult.d: 1 cycle

Description: (1) Standard

jrult sign8 ; = "jrult sign9", sign8 = sign9(8:1), sign9(0) = 0

If the condition below has been met by an unsigned operation, this instruction doubles the signed 8-bit immediate data (sign8) and adds it to the PC for branching the program flow to the address. It does not branch if the condition has not been met.

- C flag = 1 (e.g. "A < B" has resulted by "cmp A, B")

The sign8 specifies a half word address in 16-bit units.
The sign8 (×2) allows branches within the range of PC-0x100 to PC+0xFE.

(2) Extension 1

ext imm13 ; = sign22(21:9)
jrult sign8 ; = "jrult sign22", sign8 = sign22(8:1), sign22(0) = 0

The "ext" instruction extends the displacement to be added to the PC into signed 22 bits using its 13-bit immediate data (imm13). The sign22 allows branches within the range of PC-0x200000 to PC+0x1FFFFE.

(3) Extension 2

ext imm13 ; imm13(12:3)= sign32(31:22)
ext imm13' ; = sign32(21:9)
jrult sign8 ; = "jrult sign32", sign8 = sign32(8:1), sign32(0) = 0

The "ext" instructions extend the displacement to be added to the PC into signed 32 bits using their 13-bit immediate data (imm13 and imm13'). The displacement covers the entire address space. Note that the low-order 3 bits of the first imm13 are ignored.

(4) Delayed branch (d bit = 1)

jrult.d sign8

The "jrult.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before branching.
Traps that may occur between the "jrult.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
cmp     %r0,%r1       ; r0 and r1 contain unsigned data.
jrult   0x2           ; Skips the next instruction if r0 < r1.
```

Note: When using the "jrult.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

ld.b %rd, %rs

Function: Signed byte data transfer

Standard: rd(7:0) ← rs(7:0), rd(31:8) ← rs(7)
Extension 1: Invalid
Extension 2: Invalid

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td>10</td><td>9</td><td>8</td><td>7</td><td></td><td></td><td>4</td><td>3</td><td></td><td></td><td>0</td><td></td></tr>
<tr><td colspan="3">class 5</td><td colspan="3">op1</td><td colspan="2">op2</td><td colspan="4">rs</td><td colspan="4">rd</td><td></td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="4">rs</td><td colspan="4">rd</td><td>0xA100–0xA1FF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td></td><td>8</td><td>7</td><td></td><td></td><td>4</td><td>3</td><td></td><td></td><td>0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: Extends the low-order 8 bits (byte data) of the rs register into signed 32 bits (sign extended) and loads it to the rd register.

Example:

```
ld.b    %r0,%r1       ; r0←low-order 8 bits of the r1 register
                      ; with sign extension
```

ld.b %rd, [%rb]

Function: Signed byte data transfer

Standard: rd(7:0) ← B[rb], rd(31:8) ← B[rb](7)

Extension 1: rd(7:0) ← B[rb + imm13], rd(31:8) ← B[rb + imm13](7)

Extension 2: rd(7:0) ← B[rb + imm26], rd(31:8) ← B[rb + imm26](7)

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rd	
0	0	1	0	0	0	0	0	rb	rd	0x2000–0x20FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect (%rb = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.b    %rd, [%rb]    ; Memory address = rb
```

Extends the byte data in the specified memory into signed 32 bits (sign extended) and loads it to the rd register. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext     imm13
ld.b    %rd, [%rb]    ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the byte data in the address that is specified by adding the 13-bit immediate data (imm13) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

(3) Extension 2

```
ext     imm13         ; = imm26(25:13)
ext     imm13'        ; = imm26(12:0)
ld.b    %rd, [%rb]    ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the byte data in the address that is specified by adding the 26-bit immediate data (imm26) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

Example:

```
ext     0x10
ld.b    %r0,[%r1]    ; r0←B[r1+0x10] with sign extension
```

ld.b %rd, [%rb]+

Function: Signed byte data transfer

Standard: rd(7:0) ← B[rb], rd(31:8) ← B[rb](7), rb ← rb + 1
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rd	
0	0	1	0	0	0	0	1	rb	rd	0x2100–0x21FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with post increment (%rb = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 2 cycles

Description: Extends the byte data in the specified memory into signed 32 bits (sign extended) and loads it to the rd register. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+1) after the data transfer.

Example:

```
ld.b    %r0,[%r1]+   ; r0←B[r1] with sign extension, r1←r1+1
```

Note: If the same register is specified for rd and rb, the incremented address after transferring data is loaded to the rd register.

ld.b %rd, [%sp + imm6]

Function: Signed byte data transfer

Standard: rd(7:0) ← B[sp + imm6], rd(31:8) ← B[sp + imm6](7)

Extension 1: rd(7:0) ← B[sp + imm19], rd(31:8) ← B[sp + imm19](7)

Extension 2: rd(7:0) ← B[sp + imm32], rd(31:8) ← B[sp + imm32](7)

Code:

15		13	12		10	9 4	3 0	
class 2			op1			imm6	rd	
0	1	0	0	0	0	imm6	rd	0x4000–0x43FF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with displacement

Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description:

(1) Standard

```
ld.b      %rd, [%sp + imm6]   ; Memory address = sp + imm6
```

Extends the byte data in the specified memory into signed 32 bits (sign extended) and loads it to the rd register. The accessed memory address is specified by adding the 6-bit immediate data (imm6) as the displacement to the contents of the current SP.

(2) Extension 1

```
ext       imm13               ; = imm19(18:6)
ld.b      %rd, [%sp + imm6]   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the byte data in the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP is loaded to the rd register.

(3) Extension 2

```
ext       imm13               ; = imm32(31:19)
ext       imm13'              ; = imm32(18:6)
ld.b      %rd, [%sp + imm6]   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the byte data in the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP is loaded to the rd register.

Example:

```
ext     0x1
ld.b    %r0,[%sp+0x1]     ; r0←B[sp+0x41] with sign extension
```

ld.b [%rb], %rs

Function: Byte data transfer

Standard: B[rb] ← rs(7:0)
Extension 1: B[rb + imm13] ← rs(7:0)
Extension 2: B[rb + imm26] ← rs(7:0)

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td>10</td><td>9</td><td>8</td><td>7 4</td><td>3 0</td><td></td></tr>
<tr><td colspan="3">class 1</td><td colspan="3">op1</td><td colspan="2">op2</td><td>rb</td><td>rs</td><td></td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>rb</td><td>rs</td><td>0x3400–0x34FF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td></td><td>8</td><td>7 4</td><td>3 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register indirect (%rb = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

```
ld.b    [%rb], %rs    ; Memory address = rb
```

Transfers the low-order 8 bits of the rs register to the specified memory. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext     imm13
ld.b    [%rb], %rs    ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the low-order 8 bits of the rs register are transferred to the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. The rb register is not modified.

(3) Extension 2

```
ext     imm13         ; = imm26(25:13)
ext     imm13'        ; = imm26(12:0)
ld.b    [%rb], %rs    ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the low-order 8 bits of the rs register are transferred to the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. The rb register is not modified.

Example:

```
ext     0x10
ld.b    [%r1],%r0    ; B[r1+0x10]←low-order 8 bits of r0
```

ld.b [%rb]+, %rs

Function: Byte data transfer

Standard: B[rb] ← rs(7:0), rb ← rb + 1

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rs	
0	0	1	1	0	1	0	1	rb	rs	0x3500–0x35FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register indirect with post increment (%rb = %r0–%r15)

Clock: 1 cycle

Description: Transfers the low-order 8 bits of the rs register to the specified memory. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+1) after the data transfer.

Example:

```
ld.b    [%r1]+,%r0   ; B[r1]←low-order 8 bits of r0, r1←r1+1
```

ld.b [%sp + imm6], %rs

Function: Byte data transfer

Standard: B[sp + imm6] ← rs(7:0)
Extension 1: B[sp + imm19] ← rs(7:0)
Extension 2: B[sp + imm32] ← rs(7:0)

Code:

<table>
<tr><td colspan="3">15 13</td><td colspan="3">12 10</td><td>9 4</td><td>3 0</td><td></td></tr>
<tr><td colspan="3">class 2</td><td colspan="3">op1</td><td>imm6</td><td>rs</td><td></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>imm6</td><td>rs</td><td>0x5400–0x57FF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td>8 7 4</td><td>3 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rd = %r0–%r15)
Dst: Register indirect with displacement

Clock: 1 cycle

Description: (1) Standard

```
ld.b    [%sp + imm6], %rs   ; Memory address = sp + imm6
```

Transfers the low-order 8 bits of the rs register to the specified memory. The accessed memory address is specified by the rb register. The accessed memory address is specified by adding the 6-bit immediate data (imm6) as the displacement to the contents of the current SP.

(2) Extension 1

```
ext     imm13               ; = imm19(18:6)
ld.b    [%sp + imm6], %rs   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the low-order 8 bits of the rs register are transferred to the address specified by adding the 19-bit immediate data (imm19) to the contents of the SP.

(3) Extension 2

```
ext     imm13               ; = imm32(31:19)
ext     imm13'              ; = imm32(18:6)
ld.b    [%sp + imm6], %rs   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the low-order 8 bits of the rs register are transferred to the address specified by adding the 32-bit immediate data (imm32) to the contents of the SP.

Example:

```
ext     0x1
ld.b    [%sp+0x1],%r0     ; B[sp+0x41]←low-order 8 bits of r0
```

ld.h %rd, %rs

Function: Signed half word data transfer

Standard: rd(15:0) ← rs(15:0), rd(31:16) ← rs(15)

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	rd	
1	0	1	0	1	0	0	1	rs	rd	0xA900–0xA9FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: Extends the low-order 16 bits (half word data) of the rs register into signed 32 bits (sign extended) and loads it to the rd register.

Example:

```
ld.h    %r0,%r1      ; r0←low-order 16 bits of the r1 register
                     ; with sign extension
```

ld.h %rd, [%rb]

Function: Signed half word data transfer

Standard: rd(15:0) ← H[rb], rd(31:16) ← H[rb](15)
Extension 1: rd(15:0) ← H[rb + imm13], rd(31:16) ← H[rb + imm13](15)
Extension 2: rd(15:0) ← H[rb + imm26], rd(31:16) ← H[rb + imm26](15)

Code:

15		13	12		10	9	8	7 4	3 0
class 1			op1			op2		rb	rd
0	0	1	0	1	0	0	0	rb	rd
15			12	11			8	7 4	3 0

0x2800–0x28FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect (%rb = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.h    %rd, [%rb]    ; Memory address = rb
```

Extends the half word data in the specified memory into signed 32 bits (sign extended) and loads it to the rd register. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext     imm13
ld.h    %rd, [%rb]    ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the half word data in the address that is specified by adding the 13-bit immediate data (imm13) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

(3) Extension 2

```
ext     imm13         ; = imm26(25:13)
ext     imm13'        ; = imm26(12:0)
ld.h    %rd, [%rb]    ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the half word data in the address that is specified by adding the 26-bit immediate data (imm26) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

Example:

```
ext     0x10
ld.h    %r0,[%r1]     ; r0←H[r1+0x10] with sign extension
```

Note: The rb register and the displacement must specify a half word boundary address (LSB = 0). Specifying an odd address causes an address error exception.

The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.h %rd, [%rb]+

Function: Signed half word data transfer

Standard: rd(15:0) ← H[rb], rd(31:16) ← H[rb](15), rb ← rb + 2
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rd	
0	0	1	0	1	0	0	1	rb	rd	0x2900–0x29FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with post increment (%rb = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 2 cycles

Description: Extends the half word data in the specified memory into signed 32 bits (sign extended) and loads it to the rd register. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+2) after the data transfer.

Example:

```
ld.h    %r0,[%r1]+   ; r0←H[r1] with sign extension, r1←r1+2
```

Notes:

- The rb register must specify a half word boundary address (LSB = 0). Specifying an odd address causes an address error exception.
  The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.
- If the same register is specified for rd and rb, the incremented address after transferring data is loaded to the rd register.

ld.h %rd, [%sp + imm6]

Function: Signed half word data transfer

Standard: rd(15:0) ← H[sp + imm6 × 2], rd(31:16) ← H[sp + imm6 × 2](15)
Extension 1: rd(15:0) ← H[sp + imm19], rd(31:16) ← H[sp + imm19](15)
Extension 2: rd(15:0) ← H[sp + imm32], rd(31:16) ← H[sp + imm32](15)

Code:

<table>
<tr><td colspan="3">15 13</td><td colspan="3">12 10</td><td>9 4</td><td>3 0</td><td></td></tr>
<tr><td colspan="3">class 2</td><td colspan="3">op1</td><td>imm6</td><td>rd</td><td></td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>imm6</td><td>rd</td><td>0x4800–0x4BFF</td></tr>
<tr><td colspan="8">15 12 11 8 7 4 3 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with displacement
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.h     %rd, [%sp + imm6]   ; Memory address = sp + imm6 × 2
```

Extends the half word data in the specified memory into signed 32 bits (sign extended) and loads it to the rd register. The accessed memory address is specified by adding the doubled 6-bit immediate data (imm6) as the displacement to the contents of the current SP. The imm6 specifies a half word address in 16-bit units. The LSB of the displacement is always fixed at 0.

(2) Extension 1

```
ext      imm13               ; = imm19(18:6)
ld.h     %rd, [%sp + imm6]   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the half word data in the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP is loaded to the rd register.

Specify a half word boundary address (LSB = 0) for the imm6.

(3) Extension 2

```
ext      imm13               ; = imm32(31:19)
ext      imm13'              ; = imm32(18:6)
ld.h     %rd, [%sp + imm6]   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the half word data in the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP is loaded to the rd register.

Specify a half word boundary address (LSB = 0) for the imm6.

Example:

```
ext     0x1
ext     0x0
ld.h    %r1,[%sp+0x2]     ; r1←H[SP+0x80002] with sign extension
```

Note: When extending the displacement, the LSB of the imm6 will always be fixed at 0 to point to a half word boundary address. Thus an address error exception will not occur.

The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.h [%rb], %rs

Function: Half word data transfer

Standard: H[rb] ← rs(15:0)
Extension 1: H[rb + imm13] ← rs(15:0)
Extension 2: H[rb + imm26] ← rs(15:0)

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rs	
0	0	1	1	1	0	0	0	rb	rs	0x3800–0x38FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register indirect (%rb = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

```
ld.h      [%rb], %rs     ; Memory address = rb
```

Transfers the low-order 16 bits of the rs register to the specified memory. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext       imm13
ld.h      [%rb], %rs     ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the low-order 16 bits of the rs register are transferred to the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. The rb register is not modified.

(3) Extension 2

```
ext       imm13          ; = imm26(25:13)
ext       imm13'         ; = imm26(12:0)
ld.h      [%rb], %rs     ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the low-order 16 bits of the rs register are transferred to the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. The rb register is not modified.

Example:

```
ext     0x10
ld.h    [%r1],%r0    ; H[r1+0x10]←low-order 16 bits of r0
```

Note: The rb register and the displacement must specify a half word boundary address (LSB = 0). Specifying an odd address causes an address error exception.

The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.h [%rb]+, %rs

Function: Half word data transfer

Standard: H[rb] ← rs(15:0), rb ← rb + 2
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rs	
0	0	1	1	1	0	0	1	rb	rs	0x3900–0x39FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register indirect with post increment (%rb = %r0–%r15)

Clock: 1 cycle

Description: Transfers the low-order 16 bits of the rs register to the specified memory. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+2) after the data transfer.

Example:

```
ld.h    [%r1]+,%r0   ; H[r1]←low-order 16 bits of r0, r1←r1+2
```

Note: The rb register must specify a half word boundary address (LSB = 0). Specifying an odd address causes an address error exception.

The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.h [%sp + imm6], %rs

Function: Half word data transfer

Standard: H[sp + imm6 × 2] ← rs(15:0)
Extension 1: H[sp + imm19] ← rs(15:0)
Extension 2: H[sp + imm32] ← rs(15:0)

Code:

15		13	12		10	9 — 4	3 — 0	
class 2			op1			imm6	rs	
0	1	0	1	1	0	imm6	rs	0x5800–0x5BFF
15			12	11		8 7 — 4	3 — 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register indirect with displacement

Clock: 1 cycle

Description: (1) Standard

```
ld.h    [%sp + imm6], %rs   ; Memory address = sp + imm6 × 2
```

Transfers the low-order 16 bits of the rs register to the specified memory. The accessed memory address is specified by adding the doubled 6-bit immediate data (imm6) as the displacement to the contents of the current SP. The imm6 specifies a half word address in 16-bit units. The LSB of the displacement is always fixed at 0.

(2) Extension 1

```
ext     imm13               ; = imm19(18:6)
ld.h    [%sp + imm6], %rs   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the low-order 16 bits of the rs register are transferred to the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP.
Specify a half word boundary address (LSB = 0) for the imm6.

(3) Extension 2

```
ext     imm13               ; = imm32(31:19)
ext     imm13'              ; = imm32(18:6)
ld.h    [%sp + imm6], %rs   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the low-order 16 bits of the rs register are transferred to the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP.
Specify a half word boundary address (LSB = 0) for the imm6.

Example:

```
ext     0x1
ext     0x0
ld.h    [%sp+0x2],%r1     ; H[SP+0x80002]←low-order 16 bits of r1
```

Note: When extending the displacement, the LSB of the imm6 will always be fixed at 0 to point to a half word boundary address. Thus an address error exception will not occur.
The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.ub %rd, %rs

Function: Unsigned byte data transfer

Standard: rd(7:0) ← rs(7:0), rd(31:8) ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 5			op1			op2		rs	rd
1	0	1	0	0	1	0	1	rs	rd
15			12	11			8	7 4	3 0

0xA500–0xA5FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: Extends the low-order 8 bits (byte data) of the rs register into unsigned 32 bits (zero extended) and loads it to the rd register.

Example:

```
ld.ub    %r0,%r1     ; r0←low-order 8 bits of the r1 register
                     ; with zero extension
```

ld.ub %rd, [%rb]

Function: Unsigned byte data transfer

Standard: rd(7:0) ← B[rb], rd(31:8) ← 0
Extension 1: rd(7:0) ← B[rb + imm13], rd(31:8) ← 0
Extension 2: rd(7:0) ← B[rb + imm26], rd(31:8) ← 0

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rd	
0	0	1	0	0	1	0	0	rb	rd	0x2400–0x24FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect (%rb = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.ub     %rd, [%rb]     ; Memory address = rb
```

Extends the byte data in the specified memory into unsigned 32 bits (zero extended) and loads it to the rd register. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext       imm13
ld.ub     %rd, [%rb]     ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the byte data in the address that is specified by adding the 13-bit immediate data (imm13) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

(3) Extension 2

```
ext       imm13          ; = imm26(25:13)
ext       imm13'         ; = imm26(12:0)
ld.ub     %rd, [%rb]     ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the byte data in the address that is specified by adding the 26-bit immediate data (imm26) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

Example:

```
ext      0x10
ld.ub    %r0,[%r1]    ; r0←B[r1+0x10] with zero extension
```

ld.ub %rd, [%rb]+

Function: Unsigned byte data transfer

Standard: rd(7:0) ← B[rb], rd(31:8) ← 0, rb ← rb + 1

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 1			op1			op2		rb	rd
0	0	1	0	0	1	0	1	rb	rd
15			12	11			8	7 4	3 0

0x2500–0x25FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with post increment (%rb = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 2 cycles

Description: Extends the byte data in the specified memory into unsigned 32 bits (zero extended) and loads it to the rd register. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+1) after the data transfer.

Example:

```
ld.ub    %r0,[%r1]+  ; r0←B[r1] with zero extension, r1←r1+1
```

Note: If the same register is specified for rd and rb, the incremented address after transferring data is loaded to the rd register.

ld.ub %rd, [%sp + imm6]

Function: Unsigned byte data transfer

Standard: rd(7:0) ← B[sp + imm6], rd(31:8) ← 0
Extension 1: rd(7:0) ← B[sp + imm19], rd(31:8) ← 0
Extension 2: rd(7:0) ← B[sp + imm32], rd(31:8) ← 0

Code:

15		13	12		10	9 4	3 0	
class 2			op1			imm6	rd	
0	1	0	0	0	1	imm6	rd	0x4400–0x47FF
15		12	11		8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with displacement
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.ub   %rd, [%sp + imm6]   ; Memory address = sp + imm6
```

Extends the byte data in the specified memory into unsigned 32 bits (zero extended) and loads it to the rd register. The accessed memory address is specified by adding the 6-bit immediate data (imm6) as the displacement to the contents of the current SP.

(2) Extension 1

```
ext     imm13               ; = imm19(18:6)
ld.ub   %rd, [%sp + imm6]   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the byte data in the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP is loaded to the rd register.

(3) Extension 2

```
ext     imm13               ; = imm32(31:19)
ext     imm13'              ; = imm32(18:6)
ld.ub   %rd, [%sp + imm6]   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the byte data in the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP is loaded to the rd register.

Example:

```
ext     0x1
ld.ub   %r0,[%sp+0x1]    ; r0←B[sp+0x41] with zero extension
```

ld.uh %rd, %rs

Function: Unsigned half word data transfer

Standard: rd(15:0) ← rs(15:0), rd(31:16) ← 0
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	rd	
1	0	1	0	1	1	0	1	rs	rd	0xAD00–0xADFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: Extends the low-order 16 bits (half word data) of the rs register into unsigned 32 bits (zero extended) and loads it to the rd register.

Example:

```
ld.uh    %r0,%r1     ; r0←low-order 16 bits of the r1 register
                     ; with zero extension
```

ld.uh %rd, [%rb]

Function: Unsigned half word data transfer

Standard: rd(15:0) ← H[rb], rd(31:16) ← 0

Extension 1: rd(15:0) ← H[rb + imm13], rd(31:16) ← 0

Extension 2: rd(15:0) ← H[rb + imm26], rd(31:16) ← 0

Code:

15		13	12		10	9	8	7 4	3 0
class 1			op1			op2		rb	rd
0	0	1	0	1	1	0	0	rb	rd
15			12	11			8	7 4	3 0

0x2C00–0x2CFF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect (%rb = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.uh   %rd, [%rb]    ; Memory address = rb
```

Extends the half word data in the specified memory into unsigned 32 bits (zero extended) and loads it to the rd register. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext     imm13
ld.uh   %rd, [%rb]    ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the half word data in the address that is specified by adding the 13-bit immediate data (imm13) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

(3) Extension 2

```
ext     imm13         ; = imm26(25:13)
ext     imm13'        ; = imm26(12:0)
ld.uh   %rd, [%rb]    ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the half word data in the address that is specified by adding the 26-bit immediate data (imm26) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

Example:

```
ext     0x10
ld.uh   %r0,[%r1]    ; r0←H[r1+0x10] with zero extension
```

Note: The rb register and the displacement must specify a half word boundary address (LSB = 0). Specifying an odd address causes an address error exception.

The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.uh %rd, [%rb]+

Function: Unsigned half word data transfer

Standard: rd(15:0) ← H[rb], rd(31:16) ← 0, rb ← rb + 2
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 1			op1			op2		rb	rd
0	0	1	0	1	1	0	1	rb	rd
15			12	11			8	7 4	3 0

0x2D00–0x2DFF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with post increment (%rb = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 2 cycles

Description: Extends the half word data in the specified memory into unsigned 32 bits (zero extended) and loads it to the rd register. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+2) after the data transfer.

Example:

```
ld.uh    %r0,[%r1]+  ; r0←H[r1] with zero extension, r1←r1+2
```

Notes:

- The rb register must specify a half word boundary address (LSB = 0). Specifying an odd address causes an address error exception.
  The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.
- If the same register is specified for rd and rb, the incremented address after transferring data is loaded to the rd register.

ld.uh %rd, [%sp + imm6]

Function: Unsigned half word data transfer

Standard: rd(15:0) ← H[sp + imm6 × 2], rd(31:16) ← 0
Extension 1: rd(15:0) ← H[sp + imm19], rd(31:16) ← 0
Extension 2: rd(15:0) ← H[sp + imm32], rd(31:16) ← 0

Code:

15		13	12		10	9 4	3 0	
class 2			op1			imm6	rd	
0	1	0	0	1	1	imm6	rd	0x4C00–0x4FFF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with displacement
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.uh     %rd, [%sp + imm6]   ; Memory address = sp + imm6 × 2
```

Extends the half word data in the specified memory into unsigned 32 bits (zero extended) and loads it to the rd register. The accessed memory address is specified by adding the doubled 6-bit immediate data (imm6) as the displacement to the contents of the current SP. The imm6 specifies a half word address in 16-bit units. The LSB of the displacement is always fixed at 0.

(2) Extension 1

```
ext       imm13               ; = imm19(18:6)
ld.uh     %rd, [%sp + imm6]   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the half word data in the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP is loaded to the rd register.

Specify a half word boundary address (LSB = 0) for the imm6.

(3) Extension 2

```
ext       imm13               ; = imm32(31:19)
ext       imm13'              ; = imm32(18:6)
ld.uh     %rd, [%sp + imm6]   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the half word data in the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP is loaded to the rd register.

Specify a half word boundary address (LSB = 0) for the imm6.

Example:

```
ext     0x1
ext     0x0
ld.uh   %r1,[%sp+0x2]     ; r1←H[SP+0x80002] with zero extension
```

Note: When extending the displacement, the LSB of the imm6 will always be fixed at 0 to point to a half word boundary address. Thus an address error exception will not occur.

The data transfer is performed using data in the specified address as the low-order 8 bits and data in the next address as the high-order 8 bits.

ld.w %rd, %rs

Function: Word data transfer

Standard: rd ← rs

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	0	1	1	1	0	rs	rd	0x2E00–0x2EFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: Transfers the contents of the rs register (word data) to the rd register.

Example:

```
ld.w    %r0,%r1      ; r0←r1
```

Note: The ALR and the AHR can be used only in the models that have an optional multiplier. When using the ALR or the AHR for the source register in other models, this instruction functions the same as the "nop" instruction.

ld.w %rd, %ss

Function: Word data transfer

Standard: rd ← ss
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		ss	rd	
1	0	1	0	0	1	0	0	ss	rd	0xA400–0xA43F
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%ss = %sp, %psr, %alr, %ahr)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: Transfers the contents of the special register (SP, PSR, ALR, AHR) to the rd register.

Example:

```
ld.w    %r0,%psr     ; r0←psr
```

Note: The ALR and the AHR can be used only in the models that have an optional multiplier. When using the ALR or the AHR for the source register in other models, this instruction functions the same as the "nop" instruction.

ld.w %rd, [%rb]

Function: Word data transfer

Standard: rd ← W[rb]
Extension 1: rd ← W[rb + imm13]
Extension 2: rd ← W[rb + imm26]

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rd	
0	0	1	1	0	0	0	0	rb	rd	0x3000–0x30FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect (%rb = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description: (1) Standard

```
ld.w      %rd, [%rb]     ; Memory address = rb
```

Transfers the word data stored in the specified memory to the rd register. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext       imm13
ld.w      %rd, [%rb]     ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the word data in the address that is specified by adding the 13-bit immediate data (imm13) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

(3) Extension 2

```
ext       imm13          ; = imm26(25:13)
ext       imm13'         ; = imm26(12:0)
ld.w      %rd, [%rb]     ; Memory address = rb + imm26
```

The "ext" instructions change the addressing mode to register indirect with displacement. Thus the word data in the address that is specified by adding the 26-bit immediate data (imm26) to the contents of the rb register is loaded to the rd register. The rb register is not modified.

Example:

```
ext     0x10
ld.w    %r0,[%r1]    ; r0←W[r1+0x10]
```

Note: The rb register and the displacement must specify a word boundary address (low-order 2 bits = 0). Specifying other addresses causes an address error exception.

The data transfer is performed for 1 word (4 addresses) using data in the specified address as the low-order 8 bits.

ld.w %rd, [%rb]+

Function: Word data transfer

Standard: rd ← W[rb], rb ← rb + 4

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rd	
0	0	1	1	0	0	0	1	rb	rd	0x3100–0x31FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with post increment (%rb = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 2 cycles

Description: Transfers the word data stored in the specified memory to the rd register. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+4) after the data transfer.

Example:

```
ld.w    %r0,[%r1]+   ; r0←W[r1], r1←r1+4
```

Notes:

- The rb register must specify a word boundary address (low-order 2 bits = 0). Specifying other addresses causes an address error exception.
  The data transfer is performed for 1 word (4 addresses) using data in the specified address as the low-order 8 bits.
- If the same register is specified for rd and rb, the incremented address after transferring data is loaded to the rd register.

ld.w %rd, [%sp + imm6]

Function: Word data transfer

Standard: rd ← W[sp + imm6 × 4]
Extension 1: rd ← W[sp + imm19]
Extension 2: rd ← W[sp + imm32]

Code:

15 13 12 10 9 4 3 0

class 2			op1			imm6	rd	
0	1	0	1	0	0	imm6	rd	0x5000–0x53FF

15 12 11 8 7 4 3 0

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register indirect with displacement
Dst: Register direct (%rd = %r0–%r15)

Clock: 1–2 cycles

(Note) This instruction is normally executed in 1 cycle. However, it takes one more cycle if the rd register which is used in this instruction is also used in the operand of the following instruction as %rd, %rs or %rb.

Description:

(1) Standard

```
ld.w     %rd, [%sp + imm6]   ; Memory address = sp + imm6 × 4
```

Transfers the word data stored in the specified memory to the rd register. The accessed memory address is specified by adding the quadrupled 6-bit immediate data (imm6) as the displacement to the contents of the current SP. The imm6 specifies a word address in 32-bit units. The low-order 2 bits of the displacement is always fixed at 0.

(2) Extension 1

```
ext      imm13               ; = imm19(18:6)
ld.w     %rd, [%sp + imm6]   ; Memory address = sp + imm19, imm6 = imm19(5:0)
```

The "ext" instruction extends the displacement into 19 bits. Thus the word data in the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP is loaded to the rd register.

Specify a word boundary address (low-order 2 bits = 0) for the imm6.

(3) Extension 2

```
ext      imm13               ; = imm32(31:19)
ext      imm13'              ; = imm32(18:6)
ld.w     %rd, [%sp + imm6]   ; Memory address = sp + imm32, imm6 = imm32(5:0)
```

The "ext" instructions extend the displacement into 32 bits. Thus the word data in the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP is loaded to the rd register.

Specify a word boundary address (low-order 2 bits = 0) for the imm6.

Example:

```
ext     0x1
ext     0x0
ld.w    %r1,[%sp+0x4]      ; r1←W[SP+0x80004]
```

Note: When extending the displacement, the low-order 2 bits of the imm6 will always be fixed at 0 to point to a word boundary address. Thus an address error exception will not occur.

The data transfer is performed for 1 word (4 addresses) using data in the specified address as the low-order 8 bits.

ld.w %rd, sign6

Function: Word data transfer

Standard: rd(5:0) ← sign6(5:0), rd(31:6) ← sign6(5)

Extension 1: rd(18:0) ← sign19(18:0), rd(31:19) ← sign19(18)

Extension 2: rd ← sign32

Code:

15		13	12		10	9 4	3 0
class 3			op1			sign6	rd
0	1	1	0	1	1	sign6	rd
15			12	11		8 7 4	3 0

0x6C00–0x6FFF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Immediate data (Signed)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

```
ld.w      %rd, sign6    ; rd ← sign extension ← sign6
```

Extends the 6-bit immediate data (sign6) into signed 32 bits (sign extended) and loads it to the rd register.

(2) Extension 1

```
ext       imm13         ; = sign19(18:6)
ld.w      %rd, sign6    ; rd ← sign extension ← sign19, sign6 = sign19(5:0)
```

Extends the 19-bit immediate data (sign19) extended by the "ext" instruction into signed 32 bits (sign extended) and loads it to the rd register.

(3) Extension 2

```
ext       imm13         ; = sign32(31:19)
ext       imm13'        ; = sign32(18:6)
ld.w      %rd, sign6    ; rd ← sign32, sign6 = sign32(5:0)
```

Loads the 32-bit immediate data (sign32) extended by the "ext" instruction to the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Example:

```
ld.w   %r0,0x3f     ; r0←0xffffffff
```

ld.w %sd, %rs

Function: Word data transfer

Standard: sd ← rs

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	sd	
1	0	1	0	0	0	0	0	rs	sd	0xA000–0xA0F3
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N	
–	–	–	–	–	–	–	–	(All the bits change if %sd=%psr)

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%sd = %sp, %psr, %alr, %ahr)

Clock: 1 cycle

Description: Transfers the contents of the rs register (word data) to the special register (SP, PSR, ALR, AHR).

Example:

```
ld.w    %sp,%r0      ; sp←r0
```

Note: The ALR and the AHR can be used only in the models that have an optional multiplier. When using the ALR or the AHR for the destination register in other models, this instruction functions the same as the "nop" instruction.

ld.w [%rb], %rs

Function: Word data transfer

Standard: W[rb] ← rs

Extension 1: W[rb + imm13] ← rs

Extension 2: W[rb + imm26] ← rs

Code:

15 13	12 10	9 8	7 4	3 0	
class 1	op1	op2	rb	rs	
0 0 1	1 1 1	0 0	rb	rs	0x3C00–0x3CFF
15 12	11	8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register indirect (%rb = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

```
ld.w      [%rb], %rs     ; Memory address = rb
```

Transfers the contents of the rs register (word data) to the specified memory. The accessed memory address is specified by the rb register.

(2) Extension 1

```
ext       imm13
ld.w      [%rb], %rs     ; Memory address = rb + imm13
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the contents of the rs register (word data) are transferred to the address specified by adding the 13-bit immediate data (imm13) to the contents of the rb register. The rb register is not modified.

(3) Extension 2

```
ext       imm13          ; = imm26(25:13)
ext       imm13'         ; = imm26(12:0)
ld.w      [%rb], %rs     ; Memory address = rb + imm26
```

The "ext" instruction changes the addressing mode to register indirect with displacement. Thus the contents of the rs register (word data) are transferred to the address specified by adding the 26-bit immediate data (imm26) to the contents of the rb register. The rb register is not modified.

Example:

```
ext     0x10
ld.w    [%r1],%r0    ; W[r1+0x10]←r0
```

Note: The rb register and the displacement must specify a word boundary address (low-order 2 bits = 0). Specifying other addresses causes an address error exception.

The data transfer is performed for 1 word (4 addresses) using data in the specified address as the low-order 8 bits.

ld.w [%rb]+, %rs

Function: Word data transfer

Standard: W[rb] ← rs, rb ← rb + 4

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			op2		rb	rs	
0	0	1	1	1	1	0	1	rb	rs	0x3D00–0x3DFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register indirect with post increment (%rb = %r0–%r15)

Clock: 1 cycle

Description: Transfers the contents of the rs register (word data) to the specified memory. The accessed memory address is specified by the rb register. The address stored in the rb register is incremented (+4) after the data transfer.

Example:

```
ld.w    [%r1]+,%r0   ; W[r1]←r0, r1←r1+4
```

Note: The rb register must specify a word boundary address (low-order 2 bits = 0). Specifying other addresses causes an address error exception.

The data transfer is performed for 1 word (4 addresses) using data in the specified address as the low-order 8 bits.

ld.w [%sp + imm6], %rs

Function: Word data transfer

Standard: W[sp + imm6 × 4] ← rs
Extension 1: W[sp + imm19] ← rs
Extension 2: W[sp + imm32] ← rs

Code:

15		13	12		10	9 4	3 0	
class 2			op1			imm6	rs	
0	1	0	1	1	1	imm6	rs	0x5C00–0x5FFF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register indirect with displacement

Clock: 1 cycle

Description: (1) Standard

ld.w [%sp + imm6], %rs ; Memory address = sp + imm6 × 4

Transfers the contents of the rs register (word data) to the specified memory. The accessed memory address is specified by adding the quadrupled 6-bit immediate data (imm6) as the displacement to the contents of the current SP. The imm6 specifies a word address in 32-bit units. The low-order 2 bits of the displacement is always fixed at 0.

(2) Extension 1

ext imm13 ; = imm19(18:6)
ld.w [%sp + imm6], %rs ; Memory address = sp + imm19, imm6 = imm19(5:0)

The "ext" instruction extends the displacement into 19 bits. Thus the contents of the rs register (word data) are transferred to the address that is specified by adding the 19-bit immediate data (imm19) to the contents of the SP.

Specify a word boundary address (low-order 2 bits = 0) for the imm6.

(3) Extension 2

ext imm13 ; = imm32(31:19)
ext imm13' ; = imm32(18:6)
ld.w [%sp + imm6], %rs ; Memory address = sp + imm32, imm6 = imm32(5:0)

The "ext" instructions extend the displacement into 32 bits. Thus the contents of the rs register (word data) are transferred to the address that is specified by adding the 32-bit immediate data (imm32) to the contents of the SP.

Specify a word boundary address (low-order 2 bits = 0) for the imm6.

Example:

```
ext     0x1
ext     0x0
ld.w    [%sp+0x4],%r1     ; H[SP+0x80004]←r1
```

Note: When extending the displacement, the low-order 2 bits of the imm6 will always be fixed at 0 to point to a word boundary address. Thus an address error exception will not occur.

The data transfer is performed for 1 word (4 addresses) using data in the specified address as the low-order 8 bits.

mac %rs *(option)*

Function: Multiplication and accumulation

Standard: Repeats "{ahr, alr}←{ahr, alr} + H[<rs+1>] × H[<rs+2>], <rs+1>←<rs+1> + 2, <rs+2>←<rs+2> + 2" × rs times

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3			0	
class 5			op1			op2		rs	–				
1	0	1	1	0	0	1	0	rs	0	0	0	0	0xB200–0xB2F0
15			12	11			8	7 4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	↔	–	–	–	–	–	–

Mode: Register direct (%rs = %r0–%r15)

Clock: 2×N+4 cycles (N: A repeat count that is set to the rs register)

Description: The "mac %rs" instruction repeats execution of the "{AHR, ALR} ← {AHR, ALR} + H[<rs+1>]+ × H[<rs+2>]+" operation (64 bits + 16 bits × 16 bits) for the count number specified by the rs register. The rs register is used as a counter and is decremented by each operation. The "mac" instruction terminates operation when the rs register becomes 0. Thus it is possible to repeat operation up to 2^{32}-1 (4,294,967,295) times. When the "mac" instruction is executed by setting the rs register to 0, the "mac" instruction does not perform multiplication and accumulation and does not change the AHR and the ALR. The rs register is not decremented as it is 0.

<rs+1> and <rs+2> are the general-purpose registers which follow the rs register.

Example: When the R0 register is specified for rs: <rs+1>=R1 register, <rs+2>=R2 register
When the R15 register is specified for rs: <rs+1>=R0 register, <rs+2>=R1 register

The "mac" instruction uses the data stored in the addresses that are specified by these registers as the base address as signed 16-bit data for multiplication. The base addresses are incremented (+2) in each operation step.
The operation result is obtained as a 64-bit data from the AHR for the high-order 32 bits and the ALR for the low-order 32 bits.

When the temporary result overflows the signed 64-bit range during multiplication and accumulation, the MO flag in the PSR is set to 1. However, the operation continues until the repeat count that is set in the rs register goes to 0. Since the MO flag stays 1 until it is reset by software, it is possible to check whether the results are valid or not by reading the MO flag after completing execution of the "mac" instruction.

Interrupts are accepted even if the "mac" instruction is executing halfway through the repeat count. The trap processing saves the address of the "mac" instruction into the stack as the return address before branching to the interrupt handler routine. Thus when the interrupt handler routine is finished by the "reti" instruction, the suspended "mac" instruction resumes execution. The contents of the rs register at that point are used as the remaining repeat count, therefore if the interrupt handler routine has modified the rs register the "mac" instruction cannot obtain the expected results. Similarly, when the <rs+1> and/or <re+2> registers have been modified in the interrupt handler routine, the resumed "mac" instruction cannot be executed properly.

Example:

```
mac     %r1          ; Repeats "{ahr, alr}←{ahr,alr}+H[r2]+ ×
                     ; H[r3]+" r1 times
```

Note: The <rs+1> and <rs+2> registers must specify half word boundary addresses (LSB = 0). Specifying an odd address causes an address error exception.
This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

mirror %rd, %rs

Function: Mirror

Standard: rd(31:24)← rs(24:31), rd(23:16)← rs(16:23), rd(15:8)← rs(8:15), rd(7:0)← rs(0:7)

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 4			op1			op2		rs	rd	
1	0	0	1	0	1	1	0	rs	rd	0x9600–0x96FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

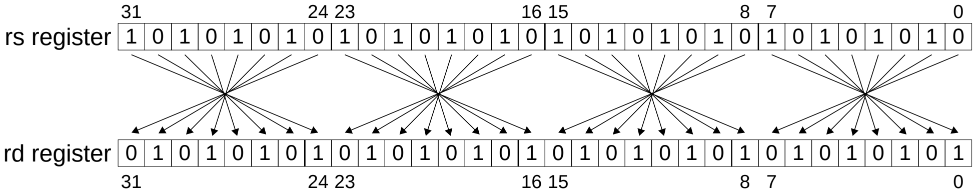
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

Swaps the bit order of the rs register high and low in byte data units and loads the results to the rd register.



(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: When r1 contains 0x88442211:

```
mirror  %r0,%r1      ; r0←0x11224488
```

Mirror operation for 32-bit data (when r1 contains 0x44332211)

```
swap    %r1,%r1      ; r1←0x11223344
mirror  %r1,%r1      ; r1←0x8844CC22
```

mlt.h %rd, %rs *(option)*

Function: Signed 16-bit multiplication

Standard: alr ← rd(15:0) × rs(15:0)
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 5			op1			op2		rs	rd
1	0	1	0	0	0	1	0	rs	rd
15			12	11			8	7 4	3 0

0xA200–0xA2FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Multiplies the low-order 16 bits of the rd register and the low-order 16 bits of the rs register with the signs and loads the results to the ALR.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example:

```
mlt.h    %r0,%r1     ; alr = r0(15:0) × r1(15:0)
                     ; signed multiplication
```

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

mltu.h %rd, %rs *(option)*

Function: Unsigned 16-bit multiplication

Standard: alr ← rd(15:0) × rs(15:0)
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	rd	
1	0	1	0	0	1	1	0	rs	rd	0xA600–0xA6FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Multiplies the low-order 16 bits of the rd register and the low-order 16 bits of the rs register without signs and loads the results to the ALR.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example:

```
mltu.h  %r0,%r1      ; alr = r0(15:0) × r1(15:0)
                     ; unsigned multiplication
```

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

mlt.w %rd, %rs *(option)*

Function: Signed 32-bit multiplication

Standard: {ahr, alr} ← rd × rs

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 5			op1			op2		rs	rd
1	0	1	0	1	0	1	0	rs	rd
15			12	11			8	7 4	3 0

0xAA00–0xAAFF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 5 cycles

Description: Multiplies the 32-bit data in the rd register and the 32-bit data in the rs register with the signs and loads the 64-bit result to the AHR (high-order 32 bits) and the ALR (low-order 32 bits).

Example:

```
mlt.w   %r0,%r1      ; {ahr, alr} = r0 × r1  signed multiplication
```

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

mltu.w %rd, %rs *(option)*

Function: Unsigned 32-bit multiplication

Standard: {ahr, alr} ← rd × rs
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	rd	
1	0	1	0	1	1	1	0	rs	rd	0xAE00–0xAEFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 5 cycles

Description: Multiplies the 32-bit data in the rd register and the 32-bit data in the rs register without signs and loads the 64-bit result to the AHR (high-order 32 bits) and the ALR (low-order 32 bits).

Example:

```
mltu.w  %r0,%r1     ; {ahr, alr} = r0 × r1  unsigned multiplication
```

Note: This instruction can be executed only in the models that have an optional multiplier. In other models, this instruction functions the same as the "nop" instruction.

nop

Function: No operation

Standard: No operation
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12			9	8	7	6		4	3			0	
class 0			op1				0	op2		0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0000
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Clock: 1 cycle

Description: The "nop" instruction just takes 1 cycle and no operation results. The PC is incremented (+2).

Example:

```
nop
nop          ; Waits 2 cycles
```

not %rd, %rs

Function: Logical negation

Standard: rd ← ! rs

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	1	1	1	1	0	rs	rd	0x3E00–0x3EFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Reverses all the bits of the rs register and loads them to the rd register.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: When the r1 register contains 0x5555555:

```
not     %r0,%r1      ; r0 = 0xAAAAAAAA
```

not %rd, sign6

Function: Logical negation

Standard: rd ← ! sign6
Extension 1: rd ← ! sign19
Extension 2: rd ← ! sign32

Code:

15		13	12		10	9 4	3 0	
class 3			op1			sign6	rd	
0	1	1	1	1	1	sign6	rd	0x7C00–0x7FFF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Immediate data (signed)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
not    %rd, sign6    ; rd ← ! sign6
```

Extends the signed 6-bit immediate data (sign6) into signed 32-bits (sign extended) and reverses all the bits, then loads the results to the rd register.

(2) Extension 1

```
ext    imm13         ; = sign19(18:6)
not    %rd, sign6    ; rd ← ! sign19, sign6 = sign19(5:0)
```

Extends the signed 19-bit immediate data (sign19) into signed 32-bits (sign extended) and reverses all the bits, then loads the results to the rd register.

(3) Extension 2

```
ext    imm13         ; = sign32(31:19)
ext    imm13'        ; = sign32(18:6)
not    %rd, sign6    ; rd ← ! sign32, sign6 = sign32(5:0)
```

Reverses all the bits of the signed 32-bit immediate data (sign32) extended by the "ext" instructions, then loads the results to the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
not     %r0,0x1f     ; r0 = 0xffffffe0

ext     0x7ff
not     %r1,0x3f     ; r1 = 0xfffe0000
```

or %rd, %rs

Function: Logical sum

Standard: rd ← rd | rs

Extension 1: rd ← rs | imm13

Extension 2: rd ← rs | imm26

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	1	0	1	1	0	rs	rd	0x3600–0x36FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
or      %rd, %rs     ; rd ← rd | rs
```

ORs the contents of the rs register and rd register and loads the results to the rd register.

(2) Extension 1

```
ext     imm13
or      %rd, %rs     ; rd ← rs | imm13
```

ORs the contents of the rs register and the 13-bit immediate data (imm13) with zero extension and loads the results to the rd register. It does not change the contents of the rs register.

(3) Extension 2

```
ext     imm13        ; = imm26(25:13)
ext     imm13'       ; = imm26(12:0)
or      %rd, %rs     ; rd ← rs | imm26
```

ORs the contents of the rs register and the 26-bit immediate data (imm26) with zero extension and loads the results to the rd register. It does not change the contents of the rs register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
or      %r0,%r0      ; r0 = r0 | r0

ext     0x1
ext     0x1fff
or      %r1,%r2      ; r1 = r2 | 0x00003fff
```

or %rd, sign6

Function: Logical sum

Standard: rd ← rd | sign6

Extension 1: rd ← rd | sign19

Extension 2: rd ← rd | sign32

Code:

15		13	12		10	9 4	3 0	
class 3			op1			sign6	rd	
0	1	1	1	0	1	sign6	rd	0x7400–0x77FF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Immediate data (signed)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
or      %rd, sign6     ; rd ← rd | sign6
```

ORs the contents of the rd register and the 6-bit immediate data (sign6) with sign extension and loads the results to the rd register.

(2) Extension 1

```
ext     imm13          ; = sign19(18:6)
or      %rd, sign6     ; rd ← rd | sign19, sign6 = sign19(5:0)
```

ORs the contents of the rd register and the 19-bit immediate data (sign19) with sign extension and loads the results to the rd register.

(3) Extension 2

```
ext     imm13          ; = sign32(31:19)
ext     imm13'         ; = sign32(18:6)
or      %rd, sign6     ; rd ← rd | sign32, sign6 = sign32(5:0)
```

ORs the contents of the rd register and the signed 32-bit immediate data (sign32) extended by the "ext" instructions and loads the results to the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
or      %r0,0x3e      ; r0 = r0 | 0xfffffffe

ext     0x7ff
or      %r1,0x3f      ; r1 = r1 | 0x0001ffff
```

popn %rd

Function: Pop

Standard: rN ← W[sp], sp ← sp + 4, repeats rN = r0 to rd

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12			9	8	7	6		4	3 … 0	
class 0			op1				0	op2		0	0	rd	
0	0	0	0	0	0	1	0	0	1	0	0	rd	0x0240–0x024F
15			12	11			8	7			4	3 … 0	

Flags:

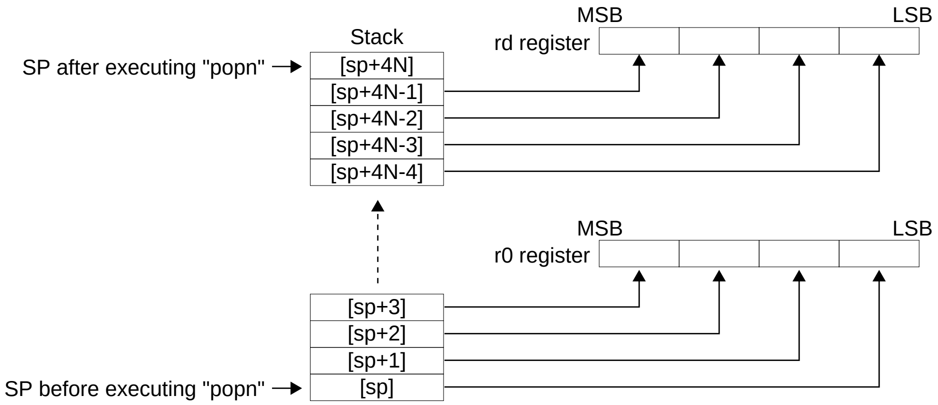
IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Register direct (%rd = %r0–%r15)

Clock: N cycles (N = number of registers to be returned)

Description: Returns data of the general-purpose registers that have been evacuated in the stack by the "pushn" instruction to each register.

The "popn" instruction first returns the word data in the address indicated by the SP to the r0 register, then increments the SP by 1 word (4 bytes). It repeats a similar operation up to the rd register sequentially. The rd register must be the same register specified by the corresponding "pushn" instruction.



Example:

```
popn    %r3      ; Returns the stacked data to r0, r1, r2 and r3.
```

pushn %rs

Function: Push

Standard: sp ← sp - 4, W[sp] ← rN, repeats rN = rs to r0

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12			9	8	7	6		4	3 0	
class 0			op1				0	op2		0	0	rs	
0	0	0	0	0	0	1	0	0	0	0	0	rs	0x0200–0x020F
15			12	11			8	7			4	3 0	

Flags:

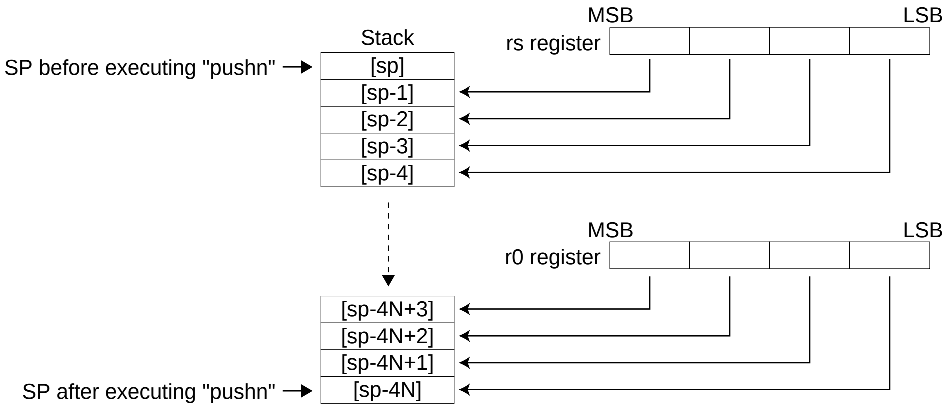
IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Register direct (%rd = %r0–%r15)

Clock: N cycles (N = number of registers to be evacuated)

Description: Saves data of the general-purpose registers into the stack.

The "pushn" instruction first decrements the current SP value by 1 word (4 bytes), then saves the contents of the rs register to the address. It repeats a similar operation up to the r0 register sequentially.



Example:

```
pushn    %r3         ; Saves r3, r2, r1 and r0
```

ret / ret.d

Function: Return from subroutine

Standard: pc ← W[sp], sp ← sp + 4
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12			9	8	7	6		4	3			0
class 0			op1				d	op2		0	0	–			
0	0	0	0	0	1	1	d	0	1	0	0	0	0	0	0
15			12	11			8	7			4	3			0

0x0640, 0x0740

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Clock: ret: 4 cycles
ret.d: 3 cycles

Description:

(1) Standard

ret

Returns the PC value (return address) that was saved into the stack when the "call" instruction was executed for returning the program flow from the subroutine to the routine that called the subroutine. The SP is incremented by 1 word.

If the SP has been modified in the subroutine, it is necessary to return the SP value before executing the "ret" instruction.

(2) Delayed branch (d bit = 1)

ret.d

The "ret.d" instruction sets the d bit in the instruction code, so the following instruction becomes a delayed instruction. The delayed instruction is executed before return from the subroutine. Traps that may occur between the "ret.d" instruction and the next delayed instruction are masked, thus interrupts and exceptions cannot occur.

Example:

```
ret.d
add     %r0,%r1      ; Executed before return from the subroutine.
```

Note: When using the "ret.d" instruction (for delayed branch), the following instruction must be an instruction that can be used as a delayed instruction. Be aware that the operation will be undefined if other instructions are executed. See the instruction list in the Appendix for the instructions that can be used as delayed instructions.

retd

Function: Return from debugging routine

Standard: r0 ← W[0xC (or 0x6000C)], pc ← W[0x8 (or 0x60008)]
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12			9	8	7	6		4	3			0	
class 0			op1				0	op2		0	0	–				
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0x0440
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Clock: 5 cycles

Description: Returns the R0 and PC values that were saved into the stack for debugging when the "brk" instruction was executed for returning the program flow from the debugging routine (debugging mode). This instruction is provided for ICE control software. Do not use it in general programs.

Example:

```
retd                    ; Returns from debugging mode.
```

reti

Function: Return from trap handler routine

Standard: psr ← W[sp], sp ← sp + 4, pc ← W[sp], sp ← sp + 4

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12			9	8	7	6		4	3			0	
class 0			op1				0	op2		0	0	–				
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0x04C0
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
↔	↔	↔	↔	↔	↔	↔	↔

Clock: 5 cycles

Description: Returns the PSR and PC values that were saved into the stack when the exception or interrupt occurred for returning the program flow from the trap handler routine. The SP is incremented by 2 words.

Example:

```
reti                  ; Returns from the trap handler routine.
```

rl %rd, %rs

Function: Rotation to left

Standard: Rotates the contents of the rd register to the left by the shift count (0–8) specified with the rs register; LSB ← MSB

Extension 1: Invalid

Extension 2: Invalid

Code:

15 13	12 10	9 8	7 4	3 0	
class 4	op1	op2	rs	rd	
1 0 0	1 1 1	0 1	rs	rd	0x9D00–0x9DFF
15	12 11	8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

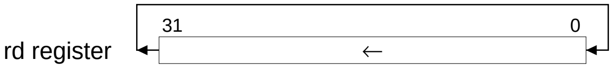
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Rotates the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register.



rs(3:0)	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x55555555 and r0 register = 1:

```
rl      %r1,%r0      ; r1 = 0xAAAAAAAA
```

rl %rd, imm4

Function: Rotation to left

Standard: Rotates the contents of the rd register to the left by the shift count (0–8) specified with the imm4; LSB ← MSB

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 – 4	3 – 0	
class 4			op1			op2		imm4	rd	
1	0	0	1	1	1	0	0	imm4	rd	0x9C00–0x9CFF
15			12	11			8	7 – 4	3 – 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

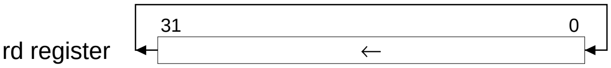
Mode: Src: Immediate data (unsigned)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Rotates the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the 4-bit immediate data (imm4).



imm4	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x01010101:

```
rl      %r1,0x4      ; r1 = 0x10101010
```

rr %rd, %rs

Function: Rotation to right

Standard: Rotates the contents of the rd register to the right by the shift count (0–8) specified with the rs register; MSB ← LSB

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 4			op1			op2		rs	rd
1	0	0	1	1	0	0	1	rs	rd
15			12	11			8	7 4	3 0

0x9900–0x99FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

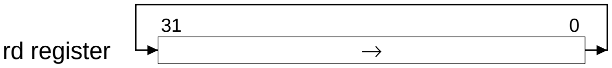
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Rotates the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register.



rs(3:0)	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x55555555 and r0 register = 1:

```
rr      %r1,%r0      ; r1 = 0xAAAAAAAA
```

rr %rd, imm4

Function: Rotation to right

Standard: Rotates the contents of the rd register to the right by the shift count (0–8) specified with the imm4; MSB ← LSB

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 4			op1			op2		imm4	rd	
1	0	0	1	1	0	0	0	imm4	rd	0x9800–0x98FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

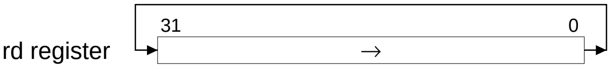
Mode: Src: Immediate data (unsigned)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Rotates the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the 4-bit immediate data (imm4).



imm4	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x01010101:

```
rr      %r1,0x4      ; r1 = 0x10101010
```

sbc %rd, %rs

Function: Subtraction with borrow

Standard: rd ← rd - rs - C

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 5			op1			op2		rs	rd	
1	0	1	1	1	1	0	0	rs	rd	0xBC00–0xBCFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Subtracts the contents of the rs register and C (carry) flag from the rd register.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example:

```
sbc     %r0,%r1      ; r0 = r0 - r1 - C
```

Subtraction of 64-bit data

data 1 = {r2, r1}, data2 = {r4, r3}, result = {r2, r1}

```
sub     %r1,%r3      ; Subtraction of the low-order word
sbc     %r2,%r4      ; Subtraction of the high-order word
                       {r2,r1} ← {r2,r1} - {r4,r3}
```

scan0 %rd, %rs

Function: 0 bit scan

Standard: rd ← 0 bit offset in rs(31:24)
Extension 1: Invalid
Extension 2: Invalid

Code:

15 13	12 10	9 8	7 4	3 0	
class 4	op1	op2	rs	rd	
1 0 0	0 1 0	1 0	rs	rd	0x8A00–0x8AFF
15	12 11	8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	0	↔	0

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Scans the most significant byte (bits 31 to 24) of the rs register. When a 0 bit is found, it loads the location (offset from MSB) to the rd register. If the MSB is 0, 0 is loaded to the rd register and the Z flag is set. If there is no 0 bit in the most significant byte of the rs register, 0x00000008 is loaded in the rd register and the C flag is set.
Bits 31 to 4 of the rd register become 0.

High-order 8 bits of rs	Low-order 8 bits of rd	C	V	Z	N
0xxx xxxx	0000 0000	0	0	1	0
10xx xxxx	0000 0001	0	0	0	0
110x xxxx	0000 0010	0	0	0	0
1110 xxxx	0000 0011	0	0	0	0
1111 0xxx	0000 0100	0	0	0	0
1111 10xx	0000 0101	0	0	0	0
1111 110x	0000 0110	0	0	0	0
1111 1110	0000 0111	0	0	0	0
1111 1111	0000 1000	1	0	0	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: Bit scan for 32-bit data

r0 = temporary register, r1 = bit-scan source data, r2 = result

```
scan0   %r0,%r1      ; 1st bit-scan
sll     %r1,%r0
ld.w    %r2,%r0
scan0   %r0,%r1      ; 2nd bit-scan
sll     %r1,%r0
add     %r2,%r0
scan0   %r0,%r1      ; 3rd bit-scan
sll     %r1,%r0
add     %r2,%r0
scan0   %r0,%r1      ; 4th bit-scan
sll     %r1,%r0
add     %r2,%r0
```

scan1 %rd, %rs

Function: 1 bit scan

Standard: rd ← 1 bit offset in rs(31:24)
Extension 1: Invalid
Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 4			op1			op2		rs	rd	
1	0	0	0	1	1	1	0	rs	rd	0x8E00–0x8EFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	0	↔	0

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Scans the most significant byte (bits 31 to 24) of the rs register. When a 1 bit is found, it loads the location (offset from MSB) to the rd register. If the MSB is 1, 0 is loaded to the rd register and the Z flag is set. If there is no 1 bit in the most significant byte of the rs register, 0x00000008 is loaded in the rd register and the C flag is set.
Bits 31 to 4 of the rd register become 0.

High-order 8 bits of rs	Low-order 8 bits of rd	C	V	Z	N
1xxx xxxx	0000 0000	0	0	1	0
01xx xxxx	0000 0001	0	0	0	0
001x xxxx	0000 0010	0	0	0	0
0001 xxxx	0000 0011	0	0	0	0
0000 1xxx	0000 0100	0	0	0	0
0000 01xx	0000 0101	0	0	0	0
0000 001x	0000 0110	0	0	0	0
0000 0001	0000 0111	0	0	0	0
0000 0000	0000 1000	1	0	0	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: Bit scan for 32-bit data

r0 = temporary register, r1 = bit-scan source data, r2 = result

```
scan1   %r0,%r1      ; 1st bit-scan
sll     %r1,%r0
ld.w    %r2,%r0
scan1   %r0,%r1      ; 2nd bit-scan
sll     %r1,%r0
add     %r2,%r0
scan1   %r0,%r1      ; 3rd bit-scan
sll     %r1,%r0
add     %r2,%r0
scan0   %r0,%r1      ; 4th bit-scan
sll     %r1,%r0
add     %r2,%r0
```

sla %rd, %rs

Function: Arithmetical shift to left

Standard: Shifts the contents of the rd register to the left by the shift count (0–8) specified with the rs register; LSB ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 – 4	3 – 0	
class 4			op1			op2		rs	rd	
1	0	0	1	0	1	0	1	rs	rd	0x9500–0x95FF
15			12	11			8	7 – 4	3 – 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

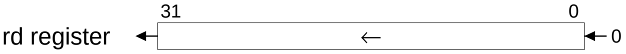
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. 0 enters to the LSB.



rs(3:0)	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x55555555 and r0 register = 1:

```
sla     %r1,%r0      ; r1 = 0xAAAAAAAA
```

sla %rd, imm4

Function: Arithmetical shift to left

Standard: Shifts the contents of the rd register to the left by the shift count (0–8) specified with the imm4; LSB ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

15 13	12 10	9 8	7 4	3 0	
class 4	op1	op2	imm4	rd	
1 0 0	1 0 1	0 0	imm4	rd	0x9400–0x94FF
15	12 11	8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

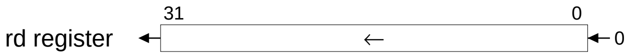
Mode: Src: Immediate data (unsigned)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the 4-bit immediate data (imm4). 0 enters to the LSB.



imm4	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x01010101:

```
sla     %r1,0x4      ; r1 = 0x10101010
```

sll %rd, %rs

Function: Logical shift to left

Standard: Shifts the contents of the rd register to the left by the shift count (0–8) specified with the rs register; LSB ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 4			op1			op2		rs	rd	
1	0	0	0	1	1	0	1	rs	rd	0x8D00–0x8DFF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

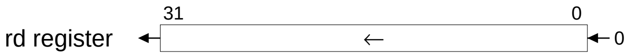
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. 0 enters to the LSB.



rs(3:0)	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x55555555 and r0 register = 1:

```
sll     %r1,%r0       ; r1 = 0xAAAAAAAA
```

sll %rd, imm4

Function: Logical shift to left

Standard: Shifts the contents of the rd register to the left by the shift count (0–8) specified with the imm4; LSB ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td>10</td><td>9</td><td>8</td><td>7 4</td><td>3 0</td></tr>
<tr><td colspan="3">class 4</td><td colspan="3">op1</td><td colspan="2">op2</td><td>imm4</td><td>rd</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>imm4</td><td>rd</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td>11</td><td></td><td></td><td>8</td><td>7 4</td><td>3 0</td></tr>
</table>

0x8C00–0x8CFF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

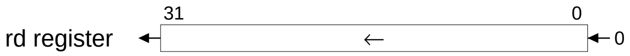
Mode: Src: Immediate data (unsigned)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. 0 enters to the LSB.



imm4	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x01010101:

```
sll     %r1,0x4      ; r1 = 0x10101010
```

slp

Function: SLEEP

Standard: Sets the CPU to SLEEP mode

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12			9	8	7	6	5	4	3			0	
class 0			op1				0	op2		0	0	–				
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0x0040
15			12	11			8	7			4	3			0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Clock: 1 cycle

Description: Sets the CPU to SLEEP mode.

In SLEEP mode, the CPU and the on-chip peripheral circuits stop operating, so current consumption can greatly be reduced.

SLEEP mode is canceled by an interrupt. When SLEEP mode is canceled, the program flow returns to the next instruction of the "slp" instruction after executing the interrupt handler routine.

Example:

```
slp                  ; Sets the CPU to SLEEP mode.
```

sra %rd, %rs

Function: Arithmetical shift to right

Standard: Shifts the contents of the rd register to the right by the shift count (0–8) specified with the rs register; MSB ← MSB

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 4			op1			op2		rs	rd
1	0	0	1	0	0	0	1	rs	rd
15			12	11			8	7 4	3 0

0x9100–0x91FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

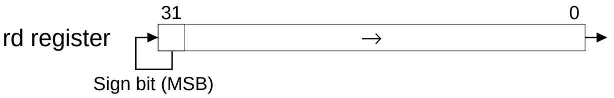
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. The sign bit is copied to the MSB.



rs(3:0)	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x55555555 and r0 register = 1:

```
sra     %r1,%r0      ; r1 = 0x2AAAAAAA
```

sra %rd, imm4

Function: Arithmetical shift to right

Standard: Shifts the contents of the rd register to the right by the shift count (0–8) specified with the imm4; MSB ← MSB

Extension 1: Invalid

Extension 2: Invalid

Code:

15 13	12 10	9 8	7 4	3 0	
class 4	op1	op2	imm4	rd	
1 0 0	1 0 0	0 0	imm4	rd	0x9000–0x90FF
15	12 11	8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

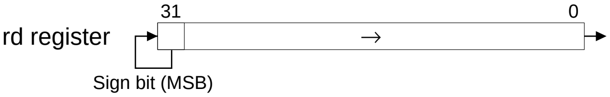
Mode: Src: Immediate data (unsigned)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. The sign bit is copied to the MSB.



imm4	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x81010101:

```
sra     %r1,0x4      ; r1 = 0xF8101010
```

srl %rd, %rs

Function: Logical shift to right

Standard: Shifts the contents of the rd register to the right by the shift count (0–8) specified with the rs register; MSB ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0
class 4			op1			op2		rs	rd
1	0	0	0	1	0	0	1	rs	rd
15			12	11			8	7 4	3 0

0x8900–0x89FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

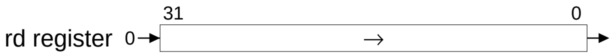
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. 0 enters to the MSB.



rs(3:0)	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x55555555 and r0 register = 1:

```
srl    %r1,%r0      ; r1 = 0x2AAAAAAA
```

srl %rd, imm4

Function: Logical shift to right

Standard: Shifts the contents of the rd register to the right by the shift count (0–8) specified with the imm4; MSB ← 0

Extension 1: Invalid

Extension 2: Invalid

Code:

15	14	13	12	11	10	9	8	7–4	3–0	
class 4			op1			op2		imm4	rd	
1	0	0	0	1	0	0	0	imm4	rd	0x8800–0x88FF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

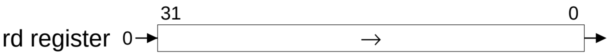
Mode: Src: Immediate data (unsigned)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Shifts the bits of the rd register as in the figure below. The shift count can be specified from 0 to 8 using the low-order 4 bits of the rs register. 0 enters to the MSB.



imm4	1xxx	0111	0110	0101	0100	0011	0010	0001	0000
Shift count	8	7	6	5	4	3	2	1	0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: In the case of r1 register = 0x01010101:

```
srl     %r1,0x4      ; r1 = 0x00101010
```

sub %rd, %rs

Function: Subtraction

Standard: rd ← rd - rs

Extension 1: rd ← rs - imm13

Extension 2: rd ← rs - imm26

Code:

15		13	12		10	9	8	7 4	3 0	
class 1			op1			1	0	rs	rd	
0	0	1	0	0	1	1	0	rs	rd	0x2600–0x26FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
sub     %rd, %rs      ; rd ← rd - rs
```

Subtracts the contents of the rs register from the rd register.

(2) Extension 1

```
ext     imm13
sub     %rd, %rs      ; rd ← rs - imm13
```

Subtracts the 13-bit immediate data (imm13) from the contents of the rs register, and then stores the results to the rd register. It does not change the contents of the rs register.

(3) Extension 2

```
ext     imm13         ; = imm26(25:13)
ext     imm13'        ; = imm26(12:0)
sub     %rd, %rs      ; rd ← rs - imm26
```

Subtracts the 26-bit immediate data (imm26) from the contents of the rs register, and then stores the results to the rd register. It does not change the contents of the rs register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
sub     %r0,%r0       ; r0 = r0 - r0

ext     0x1
ext     0x1fff
sub     %r1,%r2       ; r1 = r2 - 0x3fff
```

sub %rd, imm6

Function: Subtraction

Standard: rd ← rd - imm6
Extension 1: rd ← rd - imm19
Extension 2: rd ← rd - imm32

Code:

15		13	12		10	9 4	3 0	
class 3			op1			imm6	rd	
0	1	1	0	0	1	imm6	rd	0x6400–0x67FF
15			12	11		8 7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	↔	↔	↔	↔

Mode: Src: Immediate data (unsigned)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
sub     %rd, imm6     ; rd ← rd - imm6
```

Subtracts the 6-bit immediate data (imm6) from the rd register.

(2) Extension 1

```
ext     imm13         ; = imm19(18:6)
sub     %rd, imm6     ; rd ← rd - imm19, imm6 = imm19(5:0)
```

Subtracts the 19-bit immediate data (imm19) extended with the "ext" instruction from the rd register.

(3) Extension 2

```
ext     imm13         ; = imm32(31:19)
ext     imm13'        ; = imm32(18:6)
sub     %rd, imm6     ; rd ← rd - imm32, imm6 = imm32(5:0)
```

Subtracts the 32-bit immediate data (imm32) extended with the "ext" instructions from the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
sub     %r0,0x3f      ; r0 = r0 - 0x3f

ext     0x1fff
ext     0x1fff
sub     %r1,0x3f      ; r1 = r1 - 0xffffffff
```

sub %sp, imm10

Function: Subtraction

Standard: sp ← sp - imm10 × 4

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9 0	
class 4			op1			imm10	
1	0	0	0	0	1	imm10	0x8400–0x87FF
15			12	11		8 7 4 3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

Mode: Src: Immediate data (unsigned)

Dst: Register direct (SP)

Clock: 1 cycle

Description: (1) Standard

Quadruples the 10-bit immediate data (imm10) and subtracts it from the stack pointer SP.

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example:

```
sub     %sp,0x1      ; sp = sp - 0x4
```

swap %rd, %rs

Function: Swap

Standard: rd(31:24)← rs(7:0), rd(23:16)← rs(15:8), rd(15:8)← rs(23:16), rd(7:0)← rs(31:24)

Extension 1: Invalid

Extension 2: Invalid

Code:

15		13	12		10	9	8	7 4	3 0	
class 4			op1			op2		rs	rd	
1	0	0	1	0	0	1	0	rs	rd	0x9200–0x92FF
15			12	11			8	7 4	3 0	

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	–	–

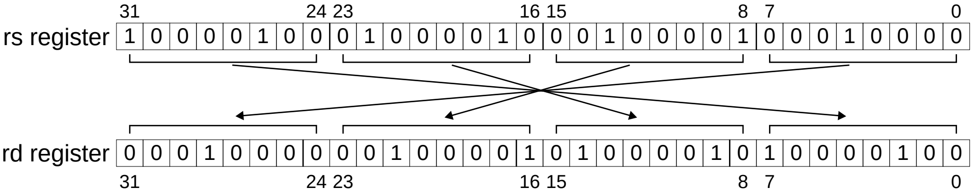
Mode: Src: Register direct (%rs = %r0–%r15)

Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description: (1) Standard

Swaps the byte order of the rs register high and low and loads the results to the rd register.



	31 … 24	23 … 16	15 … 8	7 … 0
rs register	1 0 0 0 0 1 0 0	0 1 0 0 0 0 1 0	0 0 1 0 0 0 0 1	0 0 0 1 0 0 0 0
rd register	0 0 0 1 0 0 0 0	0 0 1 0 0 0 0 1	0 1 0 0 0 0 1 0	1 0 0 0 0 1 0 0

(2) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set.

Example: When r1contains 0x87654321:

```
swap  %r0,%r1        ; r0 ← 0x21436587
```

xor %rd, %rs

Function: Exclusive OR

Standard: rd ← rd ^ rs
Extension 1: rd ← rs ^ imm13
Extension 2: rd ← rs ^ imm26

Code:

15		13	12		10	9	8	7 4	3 0
class 1			op1			1	0	rs	rd
0	0	1	1	1	0	1	0	rs	rd
15			12	11			8	7 4	3 0

0x3A00–0x3AFF

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Register direct (%rs = %r0–%r15)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
xor     %rd, %rs     ; rd ← rd ^ rs
```

Exclusive ORs the contents of the rs register and rd register and loads the results to the rd register.

(2) Extension 1

```
ext     imm13
xor     %rd, %rs     ; rd ← rs ^ imm13
```

Exclusive ORs the contents of the rs register and the 13-bit immediate data (imm13) with zero extension and loads the results to the rd register. It does not change the contents of the rs register.

(3) Extension 2

```
ext     imm13        ; = imm26(25:13)
ext     imm13'       ; = imm26(12:0)
xor     %rd, %rs     ; rd ← rs ^ imm26
```

Exclusive ORs the contents of the rs register and the 26-bit immediate data (imm26) with zero extension and loads the results to the rd register. It does not change the contents of the rs register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
xor     %r0,%r0      ; r0 = r0 ^ r0

ext     0x1
ext     0x1fff
xor     %r1,%r2      ; r1 = r2 ^ 0x00003fff
```

xor %rd, sign6

Function: Exclusive OR

Standard: rd ← rd ^ sign6
Extension 1: rd ← rd ^ sign19
Extension 2: rd ← rd ^ sign32

Code:

<table>
<tr><td>15</td><td></td><td>13</td><td>12</td><td></td><td>10</td><td>9 … 4</td><td>3 … 0</td><td></td></tr>
<tr><td colspan="3">class 3</td><td colspan="3">op1</td><td>sign6</td><td>rd</td><td></td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>sign6</td><td>rd</td><td>0x7800–0x7BFF</td></tr>
<tr><td>15</td><td></td><td></td><td>12</td><td colspan="2">11 … 8</td><td>7 … 4</td><td>3 … 0</td><td></td></tr>
</table>

Flags:

IL(3:0)	MO	DS	IE	C	V	Z	N
–	–	–	–	–	–	↔	↔

Mode: Src: Immediate data (signed)
Dst: Register direct (%rd = %r0–%r15)

Clock: 1 cycle

Description:

(1) Standard

```
xor     %rd, sign6    ; rd ← rd ^ sign6
```

Exclusive ORs the contents of the rd register and the 6-bit immediate data (sign6) with sign extension and loads the results to the rd register.

(2) Extension 1

```
ext     imm13         ; = sign19(18:6)
xor     %rd, sign6    ; rd ← rd ^ sign19, sign6 = sign19(5:0)
```

Exclusive ORs the contents of the rd register and the 19-bit immediate data (sign19) with sign extension and loads the results to the rd register.

(3) Extension 2

```
ext     imm13         ; = sign32(31:19)
ext     imm13'        ; = sign32(18:6)
xor     %rd, sign6    ; rd ← rd ^ sign32, sign6 = sign32(5:0)
```

Exclusive ORs the contents of the rd register and the signed 32-bit immediate data (sign32) extended by the "ext" instructions and loads the results to the rd register.

(4) Delayed instruction

This instruction is executed as a delayed instruction if it is described as following a branch instruction in which the d bit is set. In this case, this instruction cannot be extended with the "ext" instruction.

Examples:

```
xor     %r0,0x3e      ; r0 = r0 ^ 0xfffffffe

ext     0x7ff
xor     %r1,0x3f      ; r1 = r1 ^ 0x0001ffff
```

Appendix

EPSON

CMOS 32-bit Single Chip Microcomputer

E0C33000
Quick Reference

Memory Map and Trap Table

E0C33000 Core CPU

Memory Map

Address	Area		Area size
0xFFFFFFF	Area 18	External memory	64MB
	Area 17	External memory	64MB
	Area 16	External memory	32MB
	Area 15	External memory	32MB
	Area 14	External memory	16MB
	Area 13	External memory	16MB
	Area 12	External memory	8MB
0x1000000	Area 11	External memory	8MB
0x0C00000	Area 10	External memory	4MB
	Area 9	External memory	4MB
	Area 8	External memory	2MB
	Area 7	External memory	2MB
	Area 6	External I/O	1MB
	Area 5	External memory	1MB
0x0100000	Area 4	External memory	1MB
0x0080000	Area 3	On-chip ROM	512KB
0x0060000	Area 2	Reserved	128KB
0x0040000	Area 1	Internal I/O	128KB
	Area 0	On-chip RAM	256KB
0x0000000			

Trap Table

	Vector address
Reset	base + 0
Reserved	base + 4–12
Zero division	base + 16
Reserved	base + 20
Address error	base + 24
NMI	base + 28
Reserved	base + 32–44
Software exception 0	base + 48
:	:
Software exception 3	base + 60
External maskable interrupt 0	base + 64
:	:
External maskable interrupt 215	base + 924

base: Trap table start address
= 0x0080000 (when booting by on-chip ROM)
= 0x0C00000 (when booting by external ROM)

Registers

E0C33000 Core CPU

General-purpose registers (16)

31 0
R15
R14
R13
:
R4
R3
R2
R1
R0

Special registers (5)

31 0	
PC	Program counter
PSR	Processor status register
SP	Stack pointer
ALR	Arithmetic operation low register
AHR	Arithmetic operation high register

(AHR, ALR: Option for Multiplication & Accumulation, Multiplication, and Division)

PSR

31–12	11–8	7	6	5	4	3	2	1	0
Reserved	IL	MO	DS	–	IE	C	V	Z	N

IL:	Interrupt level	(0–15: Enabled interrupt level)
MO:	MAC overflow flag	(1: MAC overflow, 0: Not overflown)
DS:	Dividend sign flag	(1: Negative, 0: Positive)
IE:	Interrupt enable	(1: Enabled, 0: Disabled)
Z:	Zero flag	(1: Zero, 0: Non zero)
N:	Negative flag	(1: Negative, 0: Positive)
C:	Carry flag	(1: Carry/borrow, 0: No carry)
V:	Overflow flag	(1: Overflow, 0: Not overflown)

Symbols

Registers/Register Data

%rd, rd: A general-purpose register (R0–R15) used as the destination register or the contents of the register.

%rs, rs: A general-purpose register (R0–R15) used as the source register or the contents of the register.

%rb, rb: A general-purpose register (R0–R15) that has stored a base address accessed in the register indirect addressing mode or the contents of the register.

%sd, sd: A special register (PSR, SP, ALR, AHR) used as the destination register or the contents of the register.

%ss, ss: A special register (PSR, SP, ALR, AHR) used as the source register or the contents of the register.

%sp, sp: Stack pointer or the contents of the stack pointer.

∗ Register bit field in the code is replaced with a number according to the specified register (R0–R15=0–15, PSR=0, SP=1, ALR=2, AHR=3).

Memory/Addresses/Memory Data

[%rb]: Specification for register indirect addressing.

[%rb]+: Specification for register indirect addressing with post-increment.

[%sp+immX]: Specification for register indirect addressing with a displacement.

B[rb]: The address specified with the rb register, or the byte data stored in the address.

H[rb]: The half-word space in which the base address is specified with the rb register, or the half-word data stored in the space.

W[rb]: The word space in which the base address is specified with the rb register, or the word data stored in the space.

W[sp]: The word space in which the base address is specified with the SP, or the word data stored in the space.

B[sp+imm6]: The address specified with the SP and the displacement imm6, or the byte data stored in the address.

H[sp+imm7]: The half-word space in which the base address is specified with the SP and the displacement imm6 x 2, or the half-word data stored in the space.

W[sp+imm8]: The word space in which the base address is specified with SP and the displacement imm6 x 4, or the word data stored in the space.

Immediate

immX: A X-bit unsigned immediate data.

signX: A X-bit signed immediate data.

Bit Field

(X): Bit X of data.

(X:Y): A bit field from bit X to bit Y.

{X, Y···}: Indicates a bit (data) configuration.

Flags

MO:	MAC overflow flag
DS:	Dividend sign flag
Z:	Zero flag
N:	Negative flag
C:	Carry flag
V:	Overflow flag
–:	Not changed
↔:	Set (1) or reset (0)
0:	Reset (0)

Functions

←: Indicates that the right item is loaded or set to the left item.

+:	Addition
-:	Subtraction
&:	AND
|:	OR
^:	XOR
!:	NOT
×:	Multiplication

Cycle Indicates the number of execution cycles when the instruction has been stored in the internal ROM and the internal RAM is accessed.

EXT

–: Indicates that the ext instruction cannot be used for the instruction.

D

○: Indicates that the instruction can be used as a delayed instruction.

–: Indicates that the instruction cannot be used as a delayed instruction.

Data Transfer **E0C33000 Instruction Set**

Mnemonic		Code										Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB									LSB			MO	DS	C	V	Z	N		
ld.b	%rd, %rs	1	0	1	0	0	0	0	1	rs	rd	rd(7:0)←rs(7:0), rd(31:8)←rs(7)	1	–	–	–	–	–	–	–	–
	%rd, [%rb]	0	0	1	0	0	0	0	0	rb	rd	rd(7:0)←B[rb], rd(31:8)←B[rb](7)	1–2 *4	–	–	–	–	–	–	*1	–
	%rd, [%rb]+	0	0	1	0	0	0	0	1	rb	rd	rd(7:0)←B[rb], rd(31:8)←B[rb](7), rb←rb+1	2	–	–	–	–	–	–	–	–
	%rd, [%sp+imm6]	0	1	0	0	0	0			imm6	rd	rd(7:0)←B[sp+imm6], rd(31:8)←B[sp+imm6](7)	1–2 *4	–	–	–	–	–	–	*2	–
	[%rb], %rs	0	0	1	1	0	1	0	0	rb	rs	B[rb]←rs(7:0)	1	–	–	–	–	–	–	*1	–
	[%rb]+, %rs	0	0	1	1	0	1	0	1	rb	rs	B[rb]←rs(7:0), rb←rb+1	1	–	–	–	–	–	–	–	–
	[%sp+imm6], %rs	0	1	0	1	0	1			imm6	rs	B[sp+imm6]←rs(7:0)	1	–	–	–	–	–	–	*2	–
ld.ub	%rd, %rs	1	0	1	0	0	1	0	1	rs	rd	rd(7:0)←rs(7:0), rd(31:8)←0	1	–	–	–	–	–	–	–	–
	%rd, [%rb]	0	0	1	0	0	1	0	0	rb	rd	rd(7:0)←B[rb], rd(31:8)←0	1–2 *4	–	–	–	–	–	–	*1	–
	%rd, [%rb]+	0	0	1	0	0	1	0	1	rb	rd	rd(7:0)←B[rb], rd(31:8)←0, rb←rb+1	2	–	–	–	–	–	–	–	–
	%rd, [%sp+imm6]	0	1	0	0	0	1			imm6	rd	rd(7:0)←B[sp+imm6], rd(31:8)←0	1–2 *4	–	–	–	–	–	–	*2	–
ld.h	%rd, %rs	1	0	1	0	1	0	0	1	rs	rd	rd(15:0)←rs(15:0), rd(31:16)←rs(15)	1	–	–	–	–	–	–	–	–
	%rd, [%rb]	0	0	1	0	1	0	0	0	rb	rd	rd(15:0)←H[rb], rd(31:16)←H[rb](15)	1–2 *4	–	–	–	–	–	–	*1	–
	%rd, [%rb]+	0	0	1	0	1	0	0	1	rb	rd	rd(15:0)←H[rb], rd(31:16)←H[rb](15), rb←rb+2	2	–	–	–	–	–	–	–	–
	%rd, [%sp+imm6]	0	1	0	0	1	0			imm6	rd	rd(15:0)←H[sp+imm7], rd(31:16)←H[sp+imm7](15); imm7={imm6,0}	1–2 *4	–	–	–	–	–	–	*2	–
	[%rb], %rs	0	0	1	1	1	0	0	0	rb	rs	H[rb]←rs(15:0)	1	–	–	–	–	–	–	*1	–
	[%rb]+, %rs	0	0	1	1	1	0	0	1	rb	rs	H[rb]←rs(15:0), rb←rb+2	1	–	–	–	–	–	–	–	–
	[%sp+imm6], %rs	0	1	0	1	1	0			imm6	rs	H[sp+imm7]←rs(15:0); imm7={imm6,0}	1	–	–	–	–	–	–	*2	–
ld.uh	%rd, %rs	1	0	1	0	1	1	0	1	rs	rd	rd(15:0)←rs(15:0), rd(31:16)←0	1	–	–	–	–	–	–	–	–
	%rd, [%rb]	0	0	1	0	1	1	0	0	rb	rd	rd(15:0)←H[rb], rd(31:16)←0	1–2 *4	–	–	–	–	–	–	*1	–
	%rd, [%rb]+	0	0	1	0	1	1	0	1	rb	rd	rd(15:0)←H[rb], rd(31:16)←0, rb←rb+2	2	–	–	–	–	–	–	–	–
	%rd, [%sp+imm6]	0	1	0	0	1	1			imm6	rd	rd(15:0)←H[sp+imm7], rd(31:16)←0; imm7={imm6,0}	1–2 *4	–	–	–	–	–	–	*2	–
ld.w	%rd, %rs	0	0	1	0	1	1	1	0	rs	rd	rd←rs	1	–	–	–	–	–	–	–	○
	%sd, %rs	1	0	1	0	0	0	0	0	rs	sd	sd←rs	1	–	–	–	–	–	–	–	–
	%rd, %ss	1	0	1	0	0	1	0	0	ss	rd	rd←ss	1	–	–	–	–	–	–	–	–
	%rd, sign6	0	1	1	0	1	1			sign6	rd	rd(5:0)←sign6(5:0), rd(31:6)←sign6(5)	1	–	–	–	–	–	–	*3	○
	%rd, [%rb]	0	0	1	1	0	0	0	0	rb	rd	rd←W[rb]	1–2 *4	–	–	–	–	–	–	*1	–
	%rd, [%rb]+	0	0	1	1	0	0	0	1	rb	rd	rd←W[rb], rb←rb+4	2	–	–	–	–	–	–	–	–
	%rd, [%sp+imm6]	0	1	0	1	0	0			imm6	rd	rd←W[sp+imm8]; imm8={imm6,00}	1–2 *4	–	–	–	–	–	–	*2	–
	[%rb], %rs	0	0	1	1	1	1	0	0	rb	rs	W[rb]←rs	1	–	–	–	–	–	–	*1	–
	[%rb]+, %rs	0	0	1	1	1	1	0	1	rb	rs	W[rb]←rs, rb←rb+4	1	–	–	–	–	–	–	–	–
	[%sp+imm6], %rs	0	1	0	1	1	1			imm6	rs	W[sp+imm8]←rs; imm8={imm6,00}	1	–	–	–	–	–	–	*2	–

Remarks

*1) With one EXT: base address = rb+imm13, With two EXT: base address = rb+imm26

*2) With one EXT: base address = sp+imm19, With two EXT: base address = sp+imm32
(imm19 = {imm13, imm6}, imm32 = {imm13, imm13, imm6} regardless of the transfer data size)

*3) With one EXT: data = sign19, With two EXT: data = sign32

4) "ld. %rd,[%rb]" and "ld.* %rd,[%sp+imm6]" instructions are normally executed in 1 cycle. However, they take 2 cycles if the following instruction uses the rd register as the source register, destination register or base address register.

Logic Operation

E0C33000 Instruction Set

Mnemonic		Code																Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB															LSB			MO	DS	C	V	Z	N		
and	%rd, %rs	0	0	1	1	0	0	1	0	rs				rd				rd←rd & rs	1	–	–	–	–	↔	↔	∗1	○
	%rd, sign6	0	1	1	1	0	0	sign6						rd				rd←rd & sign6(with sign extension)	1	–	–	–	–	↔	↔	∗2	○
or	%rd, %rs	0	0	1	1	0	1	1	0	rs				rd				rd←rd | rs	1	–	–	–	–	↔	↔	∗1	○
	%rd, sign6	0	1	1	1	0	1	sign6						rd				rd←rd | sign6(with sign extension)	1	–	–	–	–	↔	↔	∗2	○
xor	%rd, %rs	0	0	1	1	1	0	1	0	rs				rd				rd←rd ^ rs	1	–	–	–	–	↔	↔	∗1	○
	%rd, sign6	0	1	1	1	1	0	sign6						rd				rd←rd ^ sign6(with sign extension)	1	–	–	–	–	↔	↔	∗2	○
not	%rd, %rs	0	0	1	1	1	1	1	0	rs				rd				rd←!rs	1	–	–	–	–	↔	↔	–	○
	%rd, sign6	0	1	1	1	1	1	sign6						rd				rd←!sign6(with sign extension)	1	–	–	–	–	↔	↔	∗2	○

Remarks

∗1) With one EXT: rd←rs <op> imm13, With two EXT: rd←rs <op> imm26 ∗2) With one EXT: data = sign19, With two EXT: data = sign32

Arithmetic Operation

E0C33000 Instruction Set

Mnemonic		Code																Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB															LSB			MO	DS	C	V	Z	N		
add	%rd, %rs	0	0	1	0	0	0	1	0	rs				rd				rd←rd + rs	1	–	–	↔	↔	↔	↔	∗1	○
	%rd, imm6	0	1	1	0	0	0	imm6						rd				rd←rd + imm6(with zero extension)	1	–	–	↔	↔	↔	↔	∗2	○
	%sp, imm10	1	0	0	0	0	0	imm10										sp←sp + imm12(with zero extension); imm12={imm10,00}	1	–	–	–	–	–	–	–	○
adc	%rd, %rs	1	0	1	1	1	0	0	0	rs				rd				rd←rd + rs + C	1	–	–	↔	↔	↔	↔	–	○
sub	%rd, %rs	0	0	1	0	0	1	1	0	rs				rd				rd←rd - rs	1	–	–	↔	↔	↔	↔	∗1	○
	%rd, imm6	0	1	1	0	0	1	imm6						rd				rd←rd - imm6(with zero extension)	1	–	–	↔	↔	↔	↔	∗2	○
	%sp, imm10	1	0	0	0	0	1	imm10										sp←sp - imm12(with zero extension); imm12={imm10,00}	1	–	–	–	–	–	–	–	○
sbc	%rd, %rs	1	0	1	1	1	1	0	0	rs				rd				rd←rd - rs - C	1	–	–	↔	↔	↔	↔	–	○
cmp	%rd, %rs	0	0	1	0	1	0	1	0	rs				rd				rd - rs	1	–	–	↔	↔	↔	↔	∗1	○
	%rd, sign6	0	1	1	0	1	0	sign6						rd				rd - sign6(with sign extension)	1	–	–	↔	↔	↔	↔	∗3	○
mlt.h	%rd, %rs	1	0	1	0	0	0	1	0	rs				rd				alr←rd(15:0) × rs(15:0); calculated with sign (∗6)	1	–	–	–	–	–	–	–	○
mltu.h	%rd, %rs	1	0	1	0	0	1	1	0	rs				rd				alr←rd(15:0) × rs(15:0); calculated without sign (∗6)	1	–	–	–	–	–	–	–	○
mlt.w	%rd, %rs	1	0	1	0	1	0	1	0	rs				rd				{ahr, alr}←rd × rs; calculated with sign (∗6)	5	–	–	–	–	–	–	–	–
mltu.w	%rd, %rs	1	0	1	0	1	1	1	0	rs				rd				{ahr, alr}←rd × rs; calculated without sign (∗6)	5	–	–	–	–	–	–	–	–
div0s	%rs	1	0	0	0	1	0	1	1	rs				0	0	0	0	Setup for signed division (∗6); alr = dividend, rs = divisor	1	–	↔	–	–	–	↔	–	–
div0u	%rs	1	0	0	0	1	1	1	1	rs				0	0	0	0	Setup for unsigned division (∗6); alr = dividend, rs = divisor	1	–	0	–	–	–	0	–	–
div1	%rs	1	0	0	1	0	0	1	1	rs				0	0	0	0	Step division for one bit (∗4, ∗6); alr←quotient, ahr←remainder (unsigned)	1	–	–	–	–	–	–	–	–
div2s	%rs	1	0	0	1	0	1	1	1	rs				0	0	0	0	Correction step 1 for signed division (∗5, ∗6)	1	–	–	–	–	–	–	–	–
div3s		1	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0	Correction step 2 for signed division (∗5, ∗6); alr←quotient, ahr←remainder	1	–	–	–	–	–	–	–	–

Remarks

∗1) With one EXT: rd←rs <op> imm13, With two EXT: rd←rs <op> imm26

∗2) With one EXT: data = imm19, With two EXT: data = imm32 ∗3) With one EXT: data = sign19, With two EXT: data = sign32

∗4) The div1 instruction must be executed 32 times when performing 32-bit data ÷ 32-bit data. In unsigned division, the division result is loaded to the alr and ahr registers.

∗5) It is not necessary to execute the div2s and div3s instructions for unsigned division. ∗6) These instructions can be executed only in the models that have an optional multiplier.

Shift & Rotation

E0C33000 Instruction Set

Mnemonic		Code										Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB									LSB			MO	DS	C	V	Z	N		
srl	%rd, imm4	1	0	0	0	1	0	0	0	imm4	rd	Logical shift to right imm4 bits; imm4=0–8, zero enters to MSB	1	–	–	–	–	↔	↔	–	○
	%rd, %rs	1	0	0	0	1	0	0	1	rs	rd	Logical shift to right rs bits; rs=0–8, zero enters to MSB	1	–	–	–	–	↔	↔	–	○
sll	%rd, imm4	1	0	0	0	1	1	0	0	imm4	rd	Logical shift to left imm4 bits; imm4=0–8, zero enters to LSB	1	–	–	–	–	↔	↔	–	○
	%rd, %rs	1	0	0	0	1	1	0	1	rs	rd	Logical shift to left rs bits; rs=0–8, zero enters to LSB	1	–	–	–	–	↔	↔	–	○
sra	%rd, imm4	1	0	0	1	0	0	0	0	imm4	rd	Arithmetical shift to right imm4 bits; imm4=0–8, sign copied to MSB	1	–	–	–	–	↔	↔	–	○
	%rd, %rs	1	0	0	1	0	0	0	1	rs	rd	Arithmetical shift to right rs bits; rs=0–8, sign copied to MSB	1	–	–	–	–	↔	↔	–	○
sla	%rd, imm4	1	0	0	1	0	1	0	0	imm4	rd	Arithmetical shift to left imm4 bits; imm4=0–8, zero enters to LSB	1	–	–	–	–	↔	↔	–	○
	%rd, %rs	1	0	0	1	0	1	0	1	rs	rd	Arithmetical shift to left rs bits; rs=0–8, zero enters to LSB	1	–	–	–	–	↔	↔	–	○
rr	%rd, imm4	1	0	0	1	1	0	0	0	imm4	rd	Rotation to right imm4 bits; imm4=0–8, LSB goes to MSB	1	–	–	–	–	↔	↔	–	○
	%rd, %rs	1	0	0	1	1	0	0	1	rs	rd	Rotation to right rs bits; rs=0–8, LSB goes to MSB	1	–	–	–	–	↔	↔	–	○
rl	%rd, imm4	1	0	0	1	1	1	0	0	imm4	rd	Rotation to left imm4 bits; imm4=0–8, MSB goes to LSB	1	–	–	–	–	↔	↔	–	○
	%rd, %rs	1	0	0	1	1	1	0	1	rs	rd	Rotation to left rs bits; rs=0–8, MSB goes to LSB	1	–	–	–	–	↔	↔	–	○

Bit Operation

E0C33000 Instruction Set

Mnemonic		Code											Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB										LSB			MO	DS	C	V	Z	N		
btst	[%rb], imm3	1	0	1	0	1	0	0	0	rb	0	imm3	Z flag←1 if B[rb](imm3)=0	3	–	–	–	–	↔	–	∗1	–
bclr	[%rb], imm3	1	0	1	0	1	1	0	0	rb	0	imm3	B[rb](imm3)←0	3	–	–	–	–	–	–	∗1	–
bset	[%rb], imm3	1	0	1	1	0	0	0	0	rb	0	imm3	B[rb](imm3)←1	3	–	–	–	–	–	–	∗1	–
bnot	[%rb], imm3	1	0	1	1	0	1	0	0	rb	0	imm3	B[rb](imm3)←!B[rb](imm3)	3	–	–	–	–	–	–	∗1	–

Remarks

∗1) With one EXT: address = rb+imm13, With two EXT: address = rb+imm26

Immediate Extension

E0C33000 Instruction Set

Mnemonic		Code				Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB			LSB			MO	DS	C	V	Z	N		
ext	imm13	1	1	0	imm13	Extends the immediate or operand of the following instruction.	1	–	–	–	–	–	–	∗1	–

Remarks

∗1) One or two ext instruction can be placed prior to the instructions that can be extended.

Push & Pop

E0C33000 Instruction Set

Mnemonic		Code													Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB												LSB			MO	DS	C	V	Z	N		
pushn	%rs	0	0	0	0	0	0	1	0	0	0	0	0	rs	Repeats "sp←sp-4, W[sp]←rn"; rn=rs to r0	1xn	–	–	–	–	–	–	–	–
popn	%rd	0	0	0	0	0	0	1	0	0	1	0	0	rd	Repeats "rn←W[sp], sp←sp+4"; rn=r0 to rd	1xn	–	–	–	–	–	–	–	–

Branch

Mnemonic		Code																Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB															LSB			MO	DS	C	V	Z	N		
jrgt jrgt.d	sign8	0	0	0	0	1	0	0	d	sign8								pc←pc+sign9 if !Z&!(N^V) is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrge irge.d	sign8	0	0	0	0	1	0	1	d	sign8								pc←pc+sign9 if !(N^V) is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrlt jrlt.d	sign8	0	0	0	0	1	1	0	d	sign8								pc←pc+sign9 if N^V is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrle jrle.d	sign8	0	0	0	0	1	1	1	d	sign8								pc←pc+sign9 if Z | (N^V) is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrugt jrugt.d	sign8	0	0	0	1	0	0	0	d	sign8								pc←pc+sign9 if !Z&!C is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jruge jruge.d	sign8	0	0	0	1	0	0	1	d	sign8								pc←pc+sign9 if !C is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrult jrult.d	sign8	0	0	0	1	0	1	0	d	sign8								pc←pc+sign9 if C is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrule jrule.d	sign8	0	0	0	1	0	1	1	d	sign8								pc←pc+sign9 if Z | C is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jreq jreq.d	sign8	0	0	0	1	1	0	0	d	sign8								pc←pc+sign9 if Z is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
jrne jrne.d	sign8	0	0	0	1	1	0	1	d	sign8								pc←pc+sign9 if !Z is true; sign9={sign8,0} (*2)	1–2*3, 1(.d)	–	–	–	–	–	–	*1	–
call	sign8	0	0	0	1	1	1	0	d	sign8								sp←sp-4, W[sp]←pc+2, pc←pc+sign9; sign9={sign8,0} (*2)	3,2(.d)	–	–	–	–	–	–	*1	–
call.d	%rb	0	0	0	0	0	1	1	d	0	0	0	0	rb				sp←sp-4, W[sp]←pc+2, pc←rb (*2)	3,2(.d)	–	–	–	–	–	–	–	–
jp	sign8	0	0	0	1	1	1	1	d	sign8								pc←pc+sign9; sign9={sign8,0} (*2)	2,1(.d)	–	–	–	–	–	–	*1	–
jp.d	%rb	0	0	0	0	0	1	1	d	1	0	0	0	rb				pc←rb (*2)	2,1(.d)	–	–	–	–	–	–	–	–
ret ret.d		0	0	0	0	0	1	1	d	0	1	0	0	0	0	0	0	pc←W[sp], sp←sp+4 (*2)	4, 3(.d)	–	–	–	–	–	–	–	–
reti		0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	psr←W[sp], sp←sp+4, pc←W[sp], sp←sp+4	5	↔	↔	↔	↔	↔	↔	–	–
retd		0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	Returns from debugging routine (for ICE software)	5	–	–	–	–	–	–	–	–
int	imm2	0	0	0	0	0	1	0	0	1	0	0	0	0	0	imm2		sp←sp-4, W[sp]←pc+2, sp←sp-4, W[sp]←psr, pc←software exception vector	10	–	–	–	–	–	–	–	–
brk		0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	Interrupt for debugging (for ICE software)	10	–	–	–	–	–	–	–	–

Remarks

*1) With one EXT: displacement = sign22 (= {imm13, sign8, 0}), With two EXT: displacement = sign32 (= {1st imm13(12:3), 2nd imm13, sign8, 0})

*2) These instructions become a delayed branch instruction when the d bit in the code is set to 1 by suffixing ".d" to the opcode (jrgt.d, call.d, etc.).
A delayed branch instruction executes the following delayed instruction before branching. The delayed call instruction saves the pc+4 address into the stack.

*3) The conditional branch instructions without a delayed instruction (without ".d") are executed in 1 cycle when the program flow does not branch and 2 cycles when the program flow branches.

Multiplication & Accumulation

E0C33000 Instruction Set

Mnemonic		Code																Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB															LSB			MO	DS	C	V	Z	N		
mac	%rs	1	0	1	1	0	0	1	0	rs				0	0	0	0	Repeats "{ahr, alr}←{ahr, alr} + H[<rs+1>]+ × H[<rs+2>]+" rs times	2xn+4	↔	–	–	–	–	–	–	–

Remarks

<rs+1>, <rs+2>: contents of the registers that follow rs. (eg. rs=r0: <rs+1>=r1, <rs+2>=r2; rs=r15: <rs+1>=r0, <rs+2>=r1); They are incremented (+2) after each operation.

The mac instruction can be executed only in the models that have an optional multiplier.

System Control

E0C33000 Instruction Set

Mnemonic		Code																Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB															LSB			MO	DS	C	V	Z	N		
nop		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	No operation; pc←pc+2	1	–	–	–	–	–	–	–	–
halt		0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	Sets Halt mode	1	–	–	–	–	–	–	–	–
slp		0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	Sets Sleep mode	1	–	–	–	–	–	–	–	–

Others

E0C33000 Instruction Set

Mnemonic		Code																Function	Cycle	Flags						EXT	D
Opcode	Operand	MSB															LSB			MO	DS	C	V	Z	N		
scan0	%rd, %rs	1	0	0	0	1	0	1	0	rs				rd				Scan 0 bit for 1 byte from MSB in rs, rd←offset from MSB of found bit	1	–	–	↔	0	↔	0	–	○
scan1	%rd, %rs	1	0	0	0	1	1	1	0	rs				rd				Scan 1 bit for 1 byte from MSB in rs, rd←offset from MSB of found bit	1	–	–	↔	0	↔	0	–	○
swap	%rd, %rs	1	0	0	1	0	0	1	0	rs				rd				rd(31:24)←rs(7:0), rd(23:16)←rs(15:8), rd(15:8)←rs(23:16), rd(7:0)←rs(31:24)	1	–	–	–	–	–	–	–	○
mirror	%rd, %rs	1	0	0	1	0	1	1	0	rs				rd				rd(31:24)←rs(24:31), rd(23:16)←rs(16:23), rd(15:8)←rs(8:15), rd(7:0)←rs(0:7)	1	–	–	–	–	–	–	–	○

Immediate Extension 1

E0C33000 Instruction Set

Classification	Target instruction		Extension with one ext instruction Usage: ext imm13 Target instruction			Extension with two ext instructions Usage: ext imm13 ext imm13' Target instruction		
	Opcode	Operand						
Register indirect data transfer (using rb register)	ld.b	%rd, [%rb]	ld.b	%rd, [%rb+imm13]		ld.b	%rd, [%rb+imm26]	imm26={imm13,imm13'}
	ld.ub		ld.ub			ld.ub		
	ld.h		ld.h			ld.h		
	ld.uh		ld.uh			ld.uh		
	ld.w		ld.w			ld.w		
	ld.b	[%rb], %rs	ld.b	[%rb+imm13], %rs		ld.b	[%rb+imm26], %rs	imm26={imm13,imm13'}
	ld.h		ld.h			ld.h		
	ld.w		ld.w			ld.w		
Register indirect data transfer with displacement (using SP)	ld.b	%rd, [%sp+imm6]	ld.b	%rd, [%sp+imm19]	imm19={imm13,imm6}	ld.b	%rd, [%sp+imm32]	imm32={imm13,imm13',imm6}
	ld.ub		ld.ub			ld.ub		
	ld.h		ld.h			ld.h		
	ld.uh		ld.uh			ld.uh		
	ld.w		ld.w			ld.w		
	ld.b	[%sp+imm6], %rs	ld.b	[%sp+imm19], %rs	imm19={imm13,imm6}	ld.b	[%sp+imm32], %rs	imm32={imm13,imm13',imm6}
	ld.h		ld.h			ld.h		
	ld.w		ld.w			ld.w		
Immediate load	ld.w	%rd, sign6	ld.w	%rd, sign19	sign19={1mm13, sign6}	ld.w	%rd, sign32	sign32={imm13,imm13',sign6}
Arithmetic and logic operation between registers	add	%rd, %rs	add	%rd, %rs, imm13	rd ← rs <op> imm13	add	%rd, %rs, imm26	rd ← rs <op> imm26 imm26={imm13,imm13'}
	sub		sub			sub		
	and		and			and		
	or		or			or		
	xor		xor			xor		
	cmp		cmp			cmp		
Arithmetic and logic operation with immediate	add	%rd, imm6	add	%rd, imm19	imm19={imm13,imm6}	add	%rd, imm32	imm32={imm13,imm13'imm6}
	sub		sub			sub		
	and	%rd, sign6	and	%rd, sign19	sign19={imm13,sign6}	and	%rd, sign32	sign32={imm13,imm13',sign6}
	or		or			or		
	xor		xor			xor		
	not		not			not		
	cmp		cmp			cmp		
Bit operation	btst	[%rb], imm3	btst	[%rb+imm13], imm3		btst	[%rb+imm26], imm3	imm26={imm13,imm13'}
	bset		bset			bset		
	bclr		bclr			bclr		
	bnot		bnot			bnot		

Immediate Extension 2

Classification	Target instruction		Extension with one ext instruction Usage: ext imm13 Target instruction			Extension with two ext instructions Usage: ext imm13 ext imm13' Target instruction		
	Opcode	Operand						
PC relative branch	jrgt jrgt.d	sign8	jrgt jrgt.d	sign22	sign22={imm13,sign8,0}	jrgt jrgt.d	sign32	sign32={imm13(12:3),imm13',sign8,0}
	jrge irge.d		jrge irge.d			jrge irge.d		
	jrlt jrlt.d		jrlt jrlt.d			jrlt jrlt.d		
	jrle jrle.d		jrle jrle.d			jrle jrle.d		
	jrugt jrugt.d		jrugt jrugt.d			jrugt jrugt.d		
	jruge jruge.d		jruge jruge.d			jruge jruge.d		
	jrult jrult.d		jrult jrult.d			jrult jrult.d		
	jrule jrule.d		jrule jrule.d			jrule jrule.d		
	jreq jreq.d		jreq jreq.d			jreq jreq.d		
	jrne jrne.d		jrne jrne.d			jrne jrne.d		
	call call.d		call call.d			call call.d		
	jp jp.d		jp jp.d			jp jp.d		

INSTRUCTION INDEX

EPSON International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.

- HEADQUARTERS -
1960 E. Grand Avenue
El Segundo, CA 90245, U.S.A.
Phone: +1-310-955-5300 Fax: +1-310-955-5400

- SALES OFFICES -

West
150 River Oaks Parkway
San Jose, CA 95134, U.S.A.
Phone: +1-408-922-0200 Fax: +1-408-922-0238

Central
101 Virginia Street, Suite 290
Crystal Lake, IL 60014, U.S.A.
Phone: +1-815-455-7630 Fax: +1-815-455-7633

Northeast
301 Edgewater Place, Suite 120
Wakefield, MA 01880, U.S.A.
Phone: +1-781-246-3600 Fax: +1-781-246-5443

Southeast
3010 Royal Blvd. South, Suite 170
Alpharetta, GA 30005, U.S.A.
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

EUROPE

EPSON EUROPE ELECTRONICS GmbH

- HEADQUARTERS -
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

- GERMANY -
SALES OFFICE
Altstadtstrasse 176
51379 Leverkusen, GERMANY
Phone: +49-(0)2171-5045-0 Fax: +49-(0)2171-5045-10

- UNITED KINGDOM -
UK BRANCH OFFICE
Unit 2.4, Doncastle House, Doncastle Road
Bracknell, Berkshire RG12 8PE, ENGLAND
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

- FRANCE -
FRENCH BRANCH OFFICE
1 Avenue de l' Atlantique, LP 915 Les Conquerants
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

ASIA

- CHINA -
EPSON (CHINA) CO., LTD.
28F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: 64106655 Fax: 64107319

SHANGHAI BRANCH
4F, Bldg., 27, No. 69, Gui Jing Road
Caohejing, Shanghai, CHINA
Phone: 21-6485-5552 Fax: 21-6485-0775

- HONG KONG, CHINA -
EPSON HONG KONG LTD.
20/F., Harbour Centre, 25 Harbour Road
Wanchai, HONG KONG
Phone: +852-2585-4600 Fax: +852-2827-4346
Telex: 65542 EPSCO HX

- TAIWAN -
EPSON TAIWAN TECHNOLOGY & TRADING LTD.
10F, No. 287, Nanking East Road, Sec. 3
Taipei, TAIWAN
Phone: 02-2717-7360 Fax: 02-2712-9164
Telex: 24444 EPSONTB

HSINCHU OFFICE
13F-3, No. 295, Kuang-Fu Road, Sec. 2
HsinChu 300, TAIWAN
Phone: 03-573-9900 Fax: 03-573-9169

- SINGAPORE -
EPSON SINGAPORE PTE., LTD.
No. 1 Temasek Avenue, #36-00
Millenia Tower, SINGAPORE 039192
Phone: +65-337-7911 Fax: +65-334-2716

- KOREA -
SEIKO EPSON CORPORATION KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: 02-784-6027 Fax: 02-767-3677

- JAPAN -
SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

Electronic Device Marketing Department
IC Marketing & Engineering Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5816 Fax: +81-(0)42-587-5624

ED International Marketing Department I (Europe & U.S.A.)
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5812 Fax: +81-(0)42-587-5564

ED International Marketing Department II (Asia)
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5110

ENERGY
SAVING
EPSON

In pursuit of **“Saving” Technology**, Epson electronic devices.
Our lineup of semiconductors, liquid crystal displays and quartz devices
assists in creating the products of our customers’ dreams.
Epson IS energy savings.

EPSON

SEIKO EPSON CORPORATION

ELECTRONIC DEVICES MARKETING DIVISION

■ EPSON Electronic Devices Website

http://www.epson.co.jp/device/

First issue MAY 1998, Printed FEBRUARY 2000 in Japan Ⓜ A