EPSON
EXCEED YOUR VISION

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER

S1C33 Family C33 PE

Core Manual

SEIKO EPSON CORPORATION

Configuration of product number

Devices

S1 C 33209 F 00E1 00

- **00** — **Packing specifications**
  - 00 : Besides tape & reel
  - 0A : TCP BL 2 directions
  - 0B : Tape & reel BACK
  - 0C : TCP BR 2 directions
  - 0D : TCP BT 2 directions
  - 0E : TCP BD 2 directions
  - 0F : Tape & reel FRONT
  - 0G : TCP BT 4 directions
  - 0H : TCP BD 4 directions
  - 0J : TCP SL 2 directions
  - 0K : TCP SR 2 directions
  - 0L : Tape & reel LEFT
  - 0M : TCP ST 2 directions
  - 0N : TCP SD 2 directions
  - 0P : TCP ST 4 directions
  - 0Q : TCP SD 4 directions
  - 0R : Tape & reel RIGHT
  - 99 : Specs not fixed
- **00E1** — **Specification**
- **F** — **Package**
  [D: die form; F: QFP, B: BGA]
- **33209** — **Model number**
- **C** — **Model name**
  [C: microcomputer, digital products]
- **S1** — **Product classification**
  [S1: semiconductor]

Development tools

S5U1 C 33000 H2 1 00

- **00** — **Packing specifications**
  [00: standard packing]
- **1** — **Version**
  [1: Version 1]
- **H2** — **Tool type**
  - Hx : ICE
  - Dx : Evaluation board
  - Ex : ROM emulation board
  - Mx : Emulation memory for external ROM
  - Tx : A socket for mounting
  - Cx : Compiler package
  - Sx : Middleware package
- **33000** — **Corresponding model number**
  [33L01: for S1C33L01]
- **C** — **Tool classification**
  [C: microcomputer use]
- **S5U1** — **Product classification**
  [S5U1: development tool for semiconductor products]

– Contents –

1 Summary

The C33 PE is a RISC type processor in the S1C33 series of Seiko Epson 32-bit microcomputers.
The C33 PE (Processor Element) Core is a Seiko Epson original 32-bit RISC-type core processor for the S1C33 Family microprocessors. Based on the C33 STD Core CPU features, some useful C33 ADV Core functions/ instructions were added and some of the infrequently used ones in general applications are removed to realize a high cost-performance core unit with high processing speed.
The C33 PE Core has been designed with optimization for embedded applications (full RTL design) in mind to short development time and to reduce cost.
As the principal instructions are object-code compatible with the C33 STD Core CPU, the software assets that the user has accumulated in the past can be effectively utilized.

1.1 Features

Processor type

- Seiko Epson original 32-bit RISC processor
- 32-bit internal data processing
- Contains a 32-bit × 16-bit multiplier

Operating-clock frequency

- DC to 66 MHz or higher (depending on the processor model and process technology)

Instruction set

• Code length	16-bit fixed length
• Number of instructions	125
• Execution cycle	Main instructions executed in one cycles
• Extended immediate instructions	Immediate extended up to 32 bits
• Multiplication instructions	Multiplications for 16 × 16 and 32 × 32 bits supported

Register set

- 32-bit general-purpose registers
- 32-bit special registers

Memory space and external bus

- Instruction, data, and I/O coexisting linear space
- Up to 4G bytes of memory space
- Harvard architecture using separated instruction bus and data bus

Interrupts

- Reset, NMI, and 240 external interrupts supported
- Four software exceptions
- Three instruction execution exceptions
- Direct branching from vector table to interrupt handler routine

Power-down mode

- HALT mode
- SLEEP mode

1.2 Summary of Added/Changed Functions of the C33 PE

The functions below have been added to or changed for the C33 PE Core, based on functions of the C33 STD Core CPU (S1C33000). For details, see the description of each function in subsequent sections of this manual.

1.2.1 Instructions

The C33 PE Core instruction set is compatible with the C33 STD Core CPU, note, however, that some existing instructions have been function extended or removed and new instructions have been added for high-performance operations and cost reduction.

Function-extended instructions

The C33 PE Core has the following function-extended instructions. For details, see the description of each instruction in subsequent sections of this manual.

1. The number of bits shifted by shift/rotate instructions has been increased from 8 to 32.

shift	*%rd*, *imm5* *	0–8 bits shift → 0–32 bits shift, *shift* = `srl`, `sll`, `sra`, `sla`, `rr`, `rl`
shift	*%rd*, *%rs*	0–8 bits shift → 0–32 bits shift, *shift* = `srl`, `sll`, `sra`, `sla`, `rr`, `rl`

   * Although the "*shift %rd*, *imm5*" instruction uses two actual instruction codes, they are each counted as one in the number of instructions shown on the preceding page.

2. The data transfer instructions between a general-purpose register and a special register have been modified to support newly added special registers.

`ld.w`	*%sd*, *%rs*	Special register specifiable in *%sd* added
`ld.w`	*%rd*, *%ss*	Special register specifiable in *%ss* added

Added instructions

The instructions added to the C33 PE Core are listed below. For details, see the description of each instruction in subsequent sections of this manual.

1. Instructions specifically designed to save and restore single or special registers have been added.

`push`	*%rs*	Pushes single register
`pop`	*%rd*	Pops single register
`pushs`	*%ss*	Pushes special registers successively
`pops`	*%sd*	Pops special registers successively

2. Instructions specifically designed for use with the coprocessor interface have been added.

`ld.c`	*%rd*, *imm4*	Coprocessor data transfer
`ld.c`	*imm4*, *%rs*	Coprocessor data transfer
`do.c`	*imm6*	Coprocessor execution
`ld.cf`		Coprocessor flag transfer

3. Other special instructions have been added.

`swaph`	*%rd*, *%rs*	Switches between big and little endians
`psrset`	*imm5*	Sets the PSR bit
`psrclr`	*imm5*	Clears the PSR bit
`jpr`	*%rb*	Register indirect unconditional relative branch

Instructions removed

In the C33 PE Core, the instructions listed below have been removed from the instruction set of the C33 STD Core CPU.

`div0s`	Preprocessing for signed step division
`div0u`	Preprocessing for unsigned step division
`div1`	Step division
`div2s`	Correction of the result of signed step division, 1
`div3s`	Correction of the result of signed step division, 2
`mac`	Multiply-accumulate operation
`scan0`	Scan bits for 0
`scan1`	Scan bits for 1
`mirror`	Mirroring

These functions can be realized using the software library provided or by other means.

1.2.2 Registers

The general-purpose registers (R0 to R15) are basically the same as in the C33 STD Core CPU.
The special registers have been functionally extended as described below.

PC

All 32 bits can now be used.
Moreover, the PC can now be read out to enable high-speed leaf calls.

Trap table base register

A trap table base register (TTBR) has been added.
TTBR, which was mapped at address 0x48134 in the C33 STD Core CPU, is incorporated in the C33 PE Core as a special register. The initial value (boot address) has not changed from 0xC00000.

Processor identification register

A processor identification register (IDIR) has been added for identifying the core type and version.

Debug base register

A debug base register (DBBR) has been added. This register indicates the start address of the debug area. It normally is fixed to 0x60000.

Processor status register

The following flags in PSR have been removed as have the related instructions:

MO flag (bit 7)	Mac overflow flag
DS flag (bit 6)	Divide sign

1.2.3 Address Space and Other

Address space

The C33 PE Core supports a 4G-byte space based on a 32-bit address bus.

Other

1. Interrupt/exception processing
   The Trap Table Base Register (TTBR) now serves as an internal special register of the processor.
   Furthermore, this processor has come to generate an exception when an undefined instruction (an object code not defined in the instruction set) is executed or more than two `ext` instructions are described.

2. Pipeline
   The 3-stage pipeline in the C33 STD Core CPU has been modified to a 2-stage pipeline in the C33 PE Core (consisting of fetch/decode and execute/access/write back).

2 Registers

The C33 PE Core contains 16 general-purpose registers and 8 special registers.

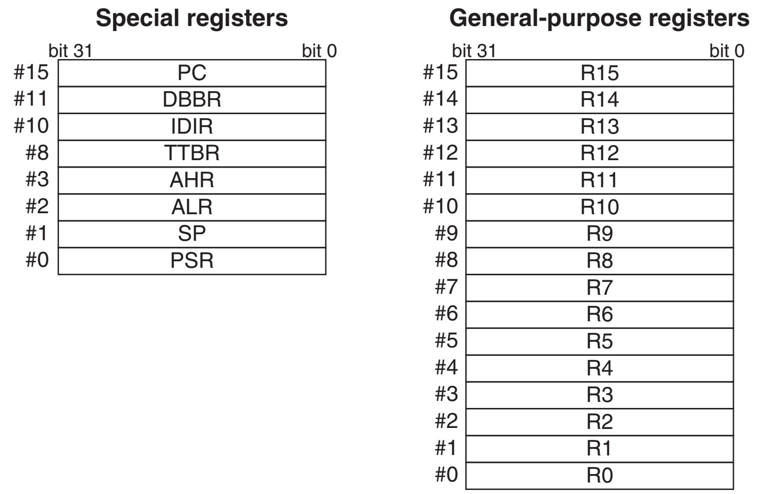


Figure 2.1 Registers

2.1 General-Purpose Registers (R0–R15)

Symbol	Register name	Size	R/W	Initial value
R0–R15	General-Purpose Register	32 bits	R/W	Indeterminate

The 16 registers R0–R15 (r0–r15) are the 32-bit general-purpose registers that can be used for data manipulation, data transfer, memory addressing, or other general purposes. The contents of all of these registers are handled as 32-bit data or addresses, so 8- or 16-bit data is sign- or zero-extended to a 32-bit quantity when it is loaded into one of these registers depending on the instruction used. When these registers are used for address references in the C33 PE Core, 32-bit space can be accessed directly.
During initialization at power-on, the contents of the general-purpose registers are indeterminate.

2.2 Program Counter (PC)

Symbol	Register name	Size	R/W	Initial value
PC	Program Counter	32 bits	R	Indeterminate

The Program Counter (hereinafter referred to as the "PC") is a 32-bit counter for holding the address of an instruction to be executed. More specifically, the PC value indicates the address of the next instruction to be executed.
As the instructions in the C33 PE Core are fixed at 16 bits in length, the low-order one bit of the PC (bit 0) is always 0. Although the C33 PE Core allows the PC to be referenced in a program, the user cannot alter it. Note, however, that the value actually loaded into the register when a `ld.w %rd,%pc` instruction (can be executed as a delayed instruction) is executed is the "PC value for the `ld` instruction + 2."
During reset, the address written at the reset vector in the vector table indicated by TTBR is loaded into the PC, and the processor starts executing a program from the address indicated by the PC.
During cold reset, TTBR is initialized to "0xC00000," so that the address written at the address "0xC00000" is the start address of the program.

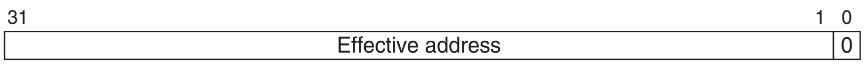


Figure 2.2.1 Program Counter (PC)

2.3 Processor Status Register (PSR)

Symbol	Register name	Size	R/W	Initial value
PSR	Processor Status Register	32 bits	R/W	0x00000000

The Processor Status Register (hereinafter referred to as the "PSR") is a 32-bit register for storing the internal status of the processor.
The PSR stores the internal status of the processor when the status has been changed by instruction execution. It is referenced in arithmetic operations or branch instructions, and therefore constitutes an important internal status in program composition. The PSR can be altered by a program.
As the PSR affects program execution, whenever an interrupt or exception occurs, the PSR is saved to the stack, except for debug exceptions, to maintain the PSR value. The IE flag (bit 4) in it is cleared to 0. The `reti` instruction is used to return from interrupt handling, and the PSR value is restored from the stack at the same time.

	31		12	11	10	9	8	7	6	5	4	3	2	1	0
	–			IL[3:0]				–	–	–	IE	C	V	Z	N
Initial value	0	.....	0	0	0	0	0	0	0	0	0	0	0	0	0
R/W	R		R	R/W	R/W	R/W	R/W	R	R	R	R/W	R/W	R/W	R/W	R/W

Figure 2.3.1 Processor Status Register (PSR)

The dash "–" in the above diagram indicates unused bits. Writing to these bits has no effect, and their value when read out is always 0.

IL[3:0] (bits 11–8): Interrupt Level

These bits indicate the priority levels of the processor interrupts. Maskable interrupt requests are accepted only when their priority levels are higher than that set in the IL bit field. When an interrupt request is accepted, the IL bit field is set to the priority level of that interrupt, and all interrupt requests generated thereafter with the same or lower priority levels are masked, unless the IL bit field is set to a different level or the interrupt handler routine is terminated by the `reti` instruction.

IE (bit 4): Interrupt Enable

This bit controls maskable external interrupts by accepting or disabling them. When IE bit = 1, the processor enables maskable external interrupts. When IE bit = 0, the processor disables maskable external interrupts.
When an interrupt or exception is accepted, the PSR is saved to the stack and this bit is cleared to 0. However, the PSR is not saved to the stack for debug exceptions, nor is this bit cleared to 0.

C (bit 3): Carry

This bit indicates a carry or borrow. More specifically, this bit is set to 1 when, in an add or subtract instruction in which the result of operation is handled as an unsigned 32-bit integer, the execution of the instruction resulted in exceeding the range of values representable by an unsigned 32-bit integer, or is reset to 0 when the result is within the range of said values.
The C flag is set under the following conditions:

(1) When an addition executed by an add instruction resulted in a value greater than the maximum value 0xFFFFFFFF representable by an unsigned 32-bit integer

(2) When a subtraction executed by a subtract instruction resulted in a value smaller than the minimum value 0x00000000 representable by an unsigned 32-bit integer

V (bit 2): OVerflow

This bit indicates that an overflow or underflow occurred in an arithmetic operation. More specifically, this bit is set to 1 when, in an add or subtract instruction in which the result of operation is handled as a signed 32-bit integer, the execution of the instruction resulted in an overflow or underflow, or is reset to 0 when the result of the add or subtract operation is within the range of values representable by a signed 32-bit integer. This flag is also reset to 0 by executing a logical operation instruction.

The V flag is set under the following conditions:

(1) When negative integers are added together, the operation produced a 0 (positive) in the sign bit (most significant bit of the result)

(2) When positive integers are added together, the operation resulted in a 1 (negative) in the sign bit (most significant bit of the result)

(3) When a negative integer is subtracted from a positive integer, the operation resulted in producing a 1 (negative) in the sign bit (most significant bit of the result)

(4) When a positive integer is subtracted from a negative integer, the operation resulted in producing a 0 (positive) in the sign bit (most significant bit of the result)

Z (bit 1): Zero

This bit indicates that an operation resulted in 0. More specifically, this bit is set to 1 when the execution of a logical operation, arithmetic operation, or shift instruction resulted in 0, or is otherwise reset to 0.

N (bit 0): Negative

This bit indicates a sign. More specifically, the most significant bit (bit 31) of the result of a logical operation, arithmetic operation, or shift instruction is copied to this N flag. If the operation being executed is step division, the sign bit of the division is set in the N flag, which affects the execution of the division.

2.4 Stack Pointer (SP)

Symbol	Register name	Size	R/W	Initial value
SP	Stack Pointer	32 bits	R/W	Indeterminate

The Stack Pointer (hereinafter referred to as the "SP") is a 32-bit register for holding the start address of the stack. The stack is an area locatable at any place in the system RAM, the start address of which is set in the SP during the initialization process. The 2 low-order bits of the SP are fixed to 0 and cannot be accessed for writing. Therefore, the addresses specifiable by the SP are those that lie on word boundaries.

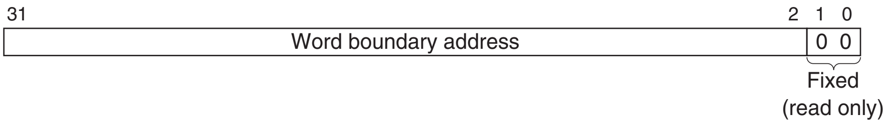


Figure 2.4.1 Stack Pointer (SP)

2.4.1 About the Stack Area

The size of an area usable as the stack is limited according to the RAM size available for the system and the size of the area occupied by ordinary RAM data. Care must be taken to prevent the stack and data area from overlapping. Furthermore, as the SP becomes indeterminate when it is initialized upon reset, "last stack address + 4, with 2 low-order bits = 0" must be written to the SP in the beginning part of the initialization routine. A load instruction may be used to write this address. If an interrupt or exception occurs before the stack is set up, it is possible that the PC or PSR will be saved to an indeterminate location, and normal operation of a program cannot be guaranteed. To prevent such a problem, NMIs (nonmaskable interrupts) that cannot be controlled in software are masked out in hardware until the SP is initialized.

2.4.2 SP Operation during Execution of `Push`-Related Instructions

In a `push`-related instruction, first the stack pointer indicated by the SP is decremented by 4 to move the SP to a lower address location.

SP = SP - 4

Next, the content of the register specified in the `push` instruction is stored at the address pointed to by the SP.

rs → [SP]

Example: `pushn   %r2`

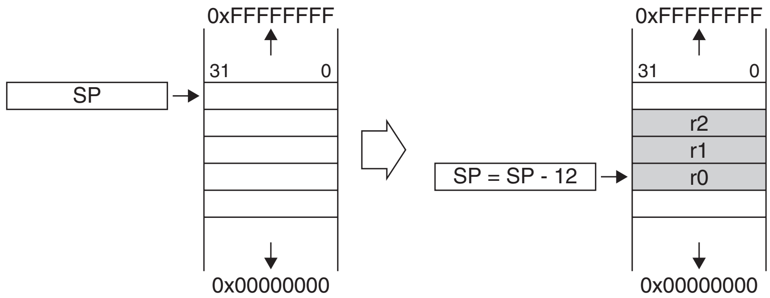


Figure 2.4.2.1 SP and Stack (1)

2.4.3 SP Operation during Execution of `Pop`-Related Instructions

In a `pop`-related instruction, first data is restored from the address indicated by the SP into the register.

[SP] → *rs*

Next, the SP is incremented by 4 to move the pointer to a higher address location.

SP = SP + 4

Example: `popn %r2`

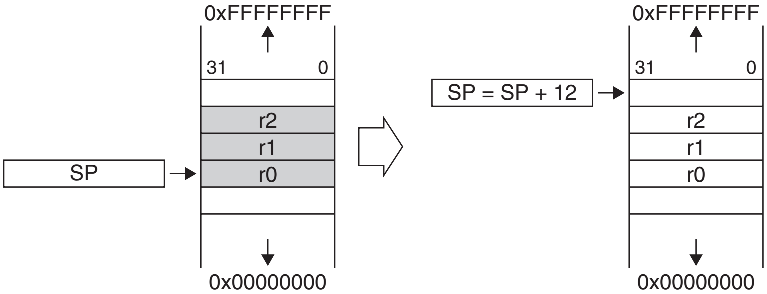


Figure 2.4.3.1 SP and Stack (2)

2.4.4 SP Operation during Execution of a `Call` Instruction

A subroutine call instruction, `call`, uses one word (32 bits) of the stack. The `call` instruction pushes the content of the PC (return address) onto the stack before branching to a subroutine. The pushed address is restored into the PC by the `ret` instruction, and the program is returned to the address next to that of the `call` instruction.

SP operation by the `call` instruction

(1) SP = SP - 4

(2) PC → [SP]

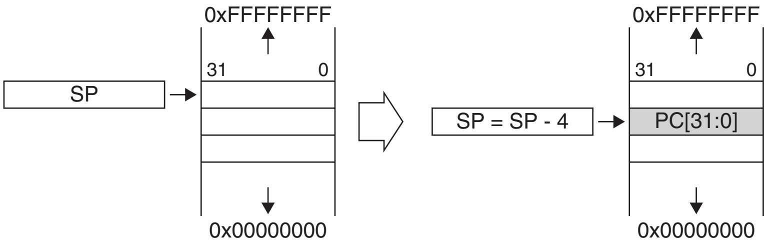


Figure 2.4.4.1 SP and Stack (3)

SP operation by the `ret` instruction

(1) [SP] → PC

(2) SP = SP + 4

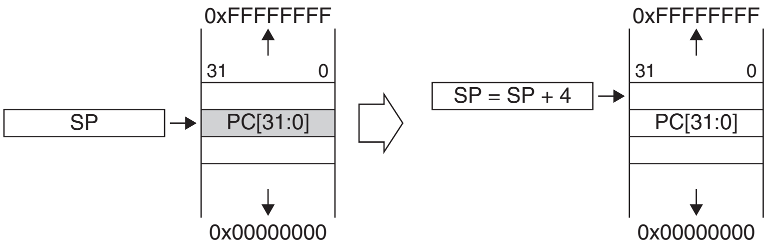


Figure 2.4.4.2 SP and Stack (4)

2.4.5 SP Operation when an Interrupt or Exception Occurs

If an interrupt or software exception resulting from the `int` instruction occurs, the processor enters an exception handling process.
The processor pushes the contents of the PC and PSR onto the stack indicated by the SP before branching to the relevant interrupt handler routine. This is to save the contents of the two registers before they are altered by interrupt or exception handling. The PC and PSR data is pushed onto the stack as shown in the diagram below.
For returning from the handler routine, the `reti` instruction is used to pop the contents of the PC and PSR off the stack. In the `reti` instruction, unlike in ordinary pop operation, the PC and PSR are read out of the stack in that order, and the SP address is altered as shown in the diagram below.

SP operation when an interrupt occurred

(1) SP = SP - 4
(2) PC → [SP]
(3) SP = SP - 4
(4) PSR → [SP]

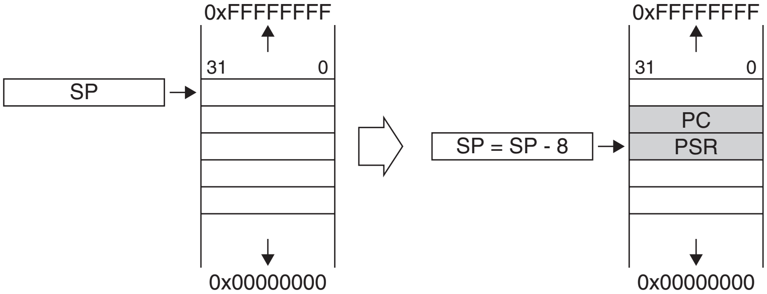


Figure 2.4.5.1 SP and Stack (5)

SP operation when the `reti` instruction is executed

(1) [SP + 4] → PC
(2) [SP] → PSR
(3) SP = SP + 8

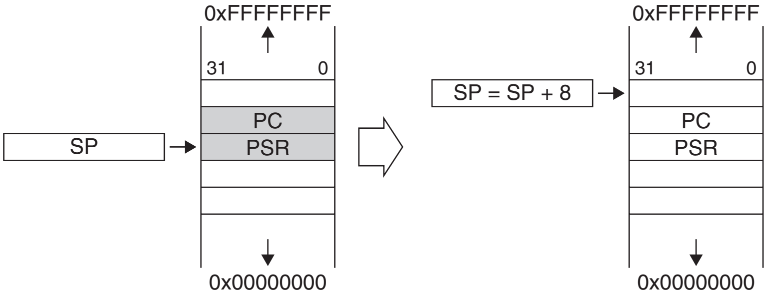


Figure 2.4.5.2 SP and Stack (6)

2.5 Trap Table Base Register (TTBR)

Symbol	Register name	Size	R/W	Initial value
TTBR	Trap Table Base Register	32 bits	R/W	0x00C00000*

The Trap Table Base Register (hereinafter referred to as the "TTBR") is a 32-bit register that is used to store the start address of the vector table to be referenced when an interrupt or exception occurs. During cold reset, the TTBR is initialized to 0x00C00000*, and the program is executed from the address indicated by the reset vector.

TTBR is a read/writable register, and can be set to any address in the software. However, bits 9–0 in the TTBR are fixed at 0 and cannot be accessed for writing. Therefore, the addresses that can be set in the TTBR are those that lie on 1K-byte boundaries.

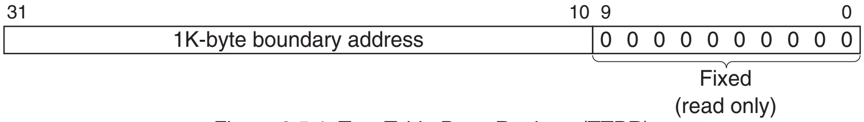


Figure 2.5.1 Trap Table Base Register (TTBR)

* The initial value (0xC00000 by default) can be changed by configuring the hardware parameters.

2.6 Arithmetic Operation Registers (ALR and AHR)

Symbol	Register name	Size	R/W	Initial value
ALR	Arithmetic Operation Low Register	32 bits	R/W	Indeterminate
AHR	Arithmetic Operation High Register	32 bits	R/W	Indeterminate

One of the special registers included in the C33 PE Core is the arithmetic operation register used in multiply operations, which consists of the Arithmetic Operation Low Register (hereinafter referred to as the "ALR") and the Arithmetic Operation High Register (hereinafter referred to as the "AHR"). Each is a 32-bit data register that allows data to be transferred to and from the general-purpose registers using load instructions. Multiply instructions use the ALR and the AHR to store the 32 low-order bits and 32 high-order bits of the result of operation, respectively. When initialized upon reset, the ALR and AHR become indeterminate.

2.7 Processor Identification Register (IDIR)

Symbol	Register name	Size	R/W	Initial value
IDIR	Processor Identification Register	32 bits	R	0x06*XXXXXX*

The Processor Identification Register (hereinafter referred to as the "IDIR") is a 32-bit register that contains the processor type, revision, and other information. The IDIR is a read-only register, and its readout value varies by model.

The bit configuration in the IDIR is detailed below.

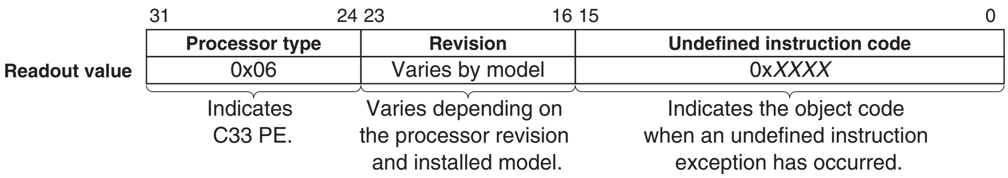


Figure 2.7.1 Processor Identification Register (IDIR)

2.8 Debug Base Register (DBBR)

Symbol	Register name	Size	R/W	Initial value
DBBR	Debug Base Register	32 bits	R	0x00060000

The Debug Base Register (hereinafter referred to as the "DBBR") is a 32-bit register that contains the base address of a memory area used for debugging. The DBBR is a read-only register which, in the C33 PE Core, is fixed to 0x00060000.

2.9 Register Notation and Register Numbers

The following describes the register notation and register numbers in the C33 PE Core instruction set.
In the instruction code, a register is specified using a 4-bit field, with the register number entered in that field. In the mnemonic, a register is specified by prefixing the register name with "`%`."

2.9.1 General-Purpose Registers

%rs *rs* is a metasymbol indicating the general-purpose register that holds the source data to be operated on or transferred. The register is actually written as `%r0`, `%r1`, ... or `%r15`.

%rd *rd* is a metasymbol indicating the general-purpose register that is the destination in which the result of operation is to be stored or data is to be loaded. The register is actually written as `%r0`, `%r1`, ... or `%r15`.

%rb *rb* is a metasymbol indicating the general-purpose register that holds the base address of memory to be accessed. In this case, the general-purpose registers serve as an index register. The register is actually written as `[%r0]`, `[%r1]`, ... or `[%r15]`, with each register name enclosed in brackets "`[]`" to denote register indirect addressing. In register indirect addressing, the post-increment function provided for continuous memory addresses can be used. In such a case, the register name is suffixed by "+," as in `[%r0]+`. When post-increment is specified, each time memory is accessed, the base address is incremented by an amount equal to the accessed size.
rb is also used as a symbol indicating the register that contains the jump address for the `call` or `jp` instruction. In this case, the brackets "`[]`" are unnecessary, and the register is written as `%r0`, `%r1`, ... or `%r15`.

The bit field that specifies a register in the instruction code contains the code corresponding to a given register number. The relationship between the general-purpose registers and the register numbers is listed in the table below.

Table 2.9.1.1 General-Purpose Registers

General-purpose register	Register number	Register notation
R0	0	%r0
R1	1	%r1
R2	2	%r2
R3	3	%r3
R4	4	%r4
R5	5	%r5
R6	6	%r6
R7	7	%r7
R8	8	%r8
R9	9	%r9
R10	10	%r10
R11	11	%r11
R12	12	%r12
R13	13	%r13
R14	14	%r14
R15	15	%r15

2.9.2 Special Registers

%ss *ss* is a metasymbol indicating the special register that holds the source data to be transferred to a general-purpose register. The instruction that operates on a special register as the source is as follows:

```
ld.w   %rd,%ss
```

%sd *sd* is a metasymbol indicating the special register to which data is to be loaded from a general-purpose register. The instruction that operates on a special register as the destination is as follows:

```
ld.w   %sd,%rs
```

The bit field that specifies a register in the instruction code contains the code corresponding to a given register number. The relationship between the special registers and the register numbers is listed in the table below.

Table 2.9.2.1 Special Registers

Special register	Register number	Register notation
PSR	0	%psr
SP	1	%sp
ALR	2	%alr
AHR	3	%ahr
TTBR *	8	%ttbr
IDIR *	10	%idir
DBBR *	11	%dbbr
PC	15	%pc

The new registers added to the C33 PE Core are marked with * in the above table.

3 Data Formats

The C33 PE Core can handle data of 8, 16, and 32 bits in length. In this manual, data sizes are expressed as follows:

8-bit data **Byte**, **B**, or **b**
16-bit data **Halfword**, **H**, or **h**
32-bit data **Word**, **W**, or **w**

Data sizes can be selected only in data transfer (load instruction) between memory and a general-purpose register, and between one general-purpose register and another.
As all internal processing in the processor is performed in 32 bits, in a 16-bit or 8-bit data transfer with a general-purpose register as the destination, the data is sign- or zero-extended to 32 bits before being loaded into the register. Whether the data will be sign- or zero-extended is determined by the load instruction used.
In a 16-bit or 8-bit data transfer using a general-purpose register as the source, the data to be transferred is stored in the low-order halfword or the 1 low-order byte of the source register.

Memory is accessed in little endian format one byte, halfword, or word at a time.
If memory is to be accessed in halfword or word units, the specified base address must be on a halfword boundary (least significant address bit = 0) or word boundary (2 low-order address bits = 00), respectively. Unless this condition is satisfied, an address-misaligned exception is generated.

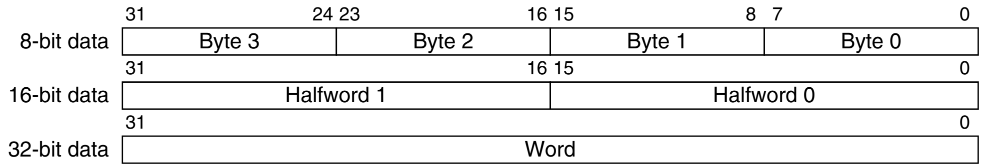


Figure 3.1 Little Endian Format

The data transfer sizes and types are described below.

3.1 Unsigned 8-Bit Transfer (Register → Register)

Example: `ld.ub` *`%rd,%rs`*

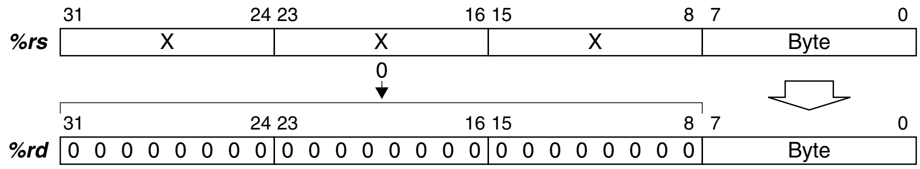


Figure 3.1.1 Unsigned 8-Bit Transfer (Register → Register)

Bits 31–8 in the destination register are zero-extended.

3.2 Signed 8-Bit Transfer (Register → Register)

Example: `ld.b` *`%rd,%rs`*

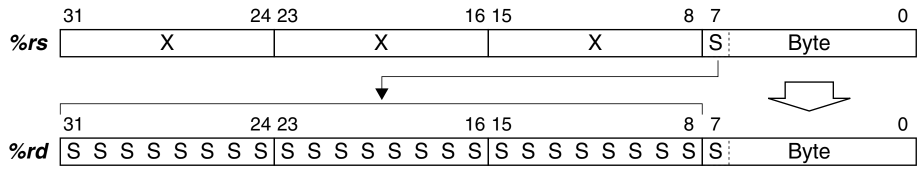


Figure 3.2.1 Signed 8-Bit Transfer (Register → Register)

Bits 31–8 in the destination register are sign-extended.

3.3 Unsigned 8-Bit Transfer (Memory → Register)

Example: `ld.ub    %rd, [%rb]`

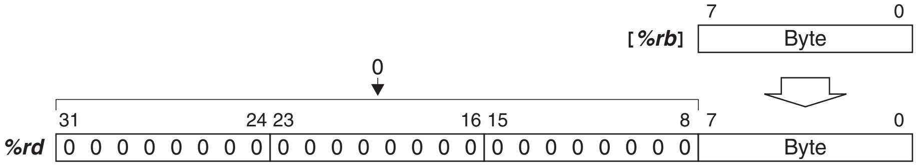


Figure 3.3.1 Unsigned 8-Bit Transfer (Memory → Register)

Bits 31–8 in the destination register are zero-extended.

3.4 Signed 8-Bit Transfer (Memory → Register)

Example: `ld.b    %rd, [%rb]`

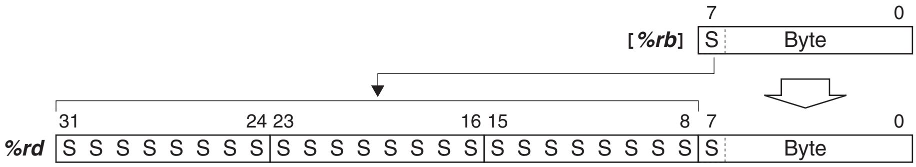


Figure 3.4.1 Signed 8-Bit Transfer (Memory → Register)

Bits 31–8 in the destination register are sign-extended.

3.5 8-Bit Transfer (Register → Memory)

Example: `ld.b    [%rb], %rs`

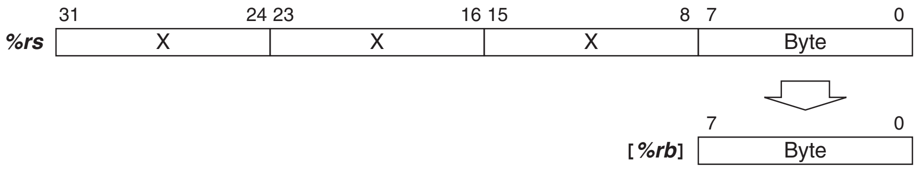


Figure 3.5.1 8-Bit Transfer (Register → Memory)

3.6 Unsigned 16-Bit Transfer (Register → Register)

Example: `ld.uh    %rd, %rs`

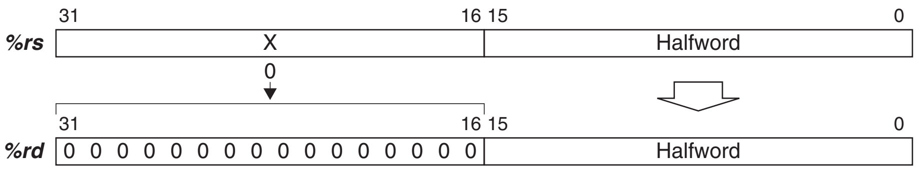


Figure 3.6.1 Unsigned 16-Bit Transfer (Register → Register)

Bits 31–16 in the destination register are zero-extended.

3.7 Signed 16-Bit Transfer (Register → Register)

Example: `ld.h   %rd,%rs`

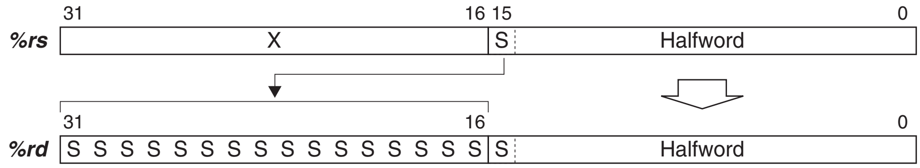


Figure 3.7.1 Signed 16-Bit Transfer (Register → Register)

Bits 31–16 in the destination register are sign-extended.

3.8 Unsigned 16-Bit Transfer (Memory → Register)

Example: `ld.uh   %rd,[%rb]`

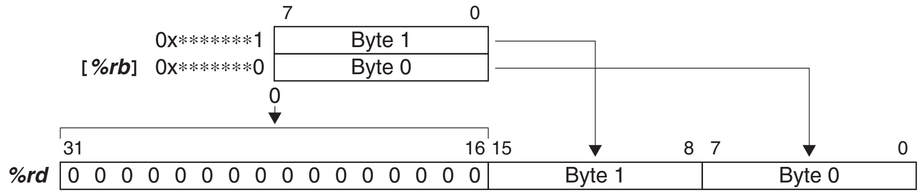


Figure 3.8.1 Unsigned 16-Bit Transfer (Memory → Register)

Bits 31–16 in the destination register are zero-extended.

3.9 Signed 16-Bit Transfer (Memory → Register)

Example: `ld.h   %rd,[%rb]`

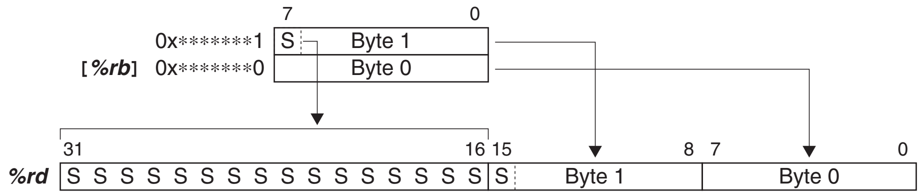


Figure 3.9.1 Signed 16-Bit Transfer (Memory → Register)

Bits 31–16 in the destination register are sign-extended.

3.10 16-Bit Transfer (Register → Memory)

Example: `ld.h   [%rb],%rs`

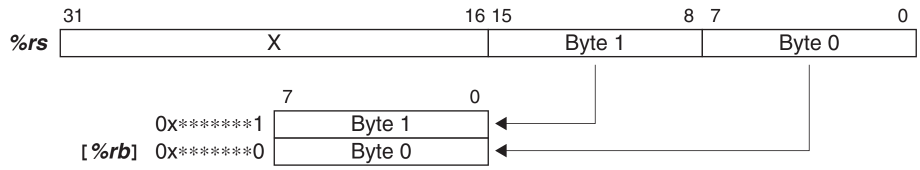


Figure 3.10.1 16-Bit Transfer (Register → Memory)

3.11 32-Bit Transfer (Register → Register)

Example: `ld.w  %rd,%rs`

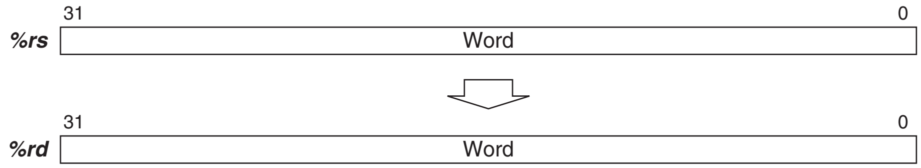


Figure 3.11.1 32-Bit Transfer (Register → Register)

3.12 32-Bit Transfer (Memory → Register)

Example: `ld.w  %rd,[%rb]`

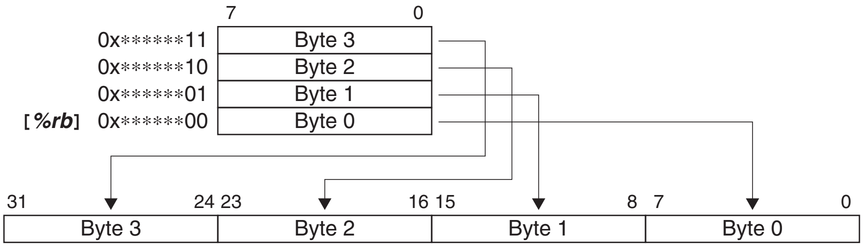


Figure 3.12.1 32-Bit Transfer (Memory → Register)

3.13 32-Bit Transfer (Register → Memory)

Example: `ld.w  [%rb],%rs`

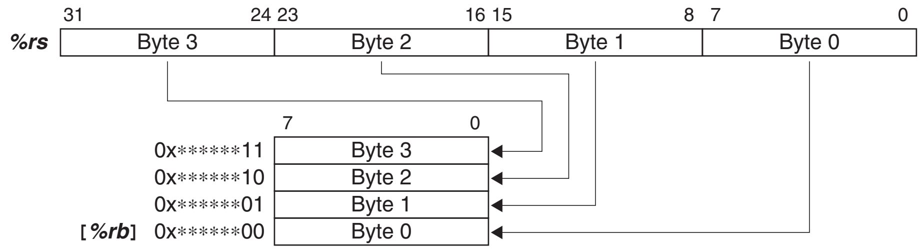


Figure 3.13.1 32-Bit Transfer (Register → Memory)

4 Address Map

The C33 PE Core has a 4GB address space. Figure 4.1 shows the C33 PE Core address map.

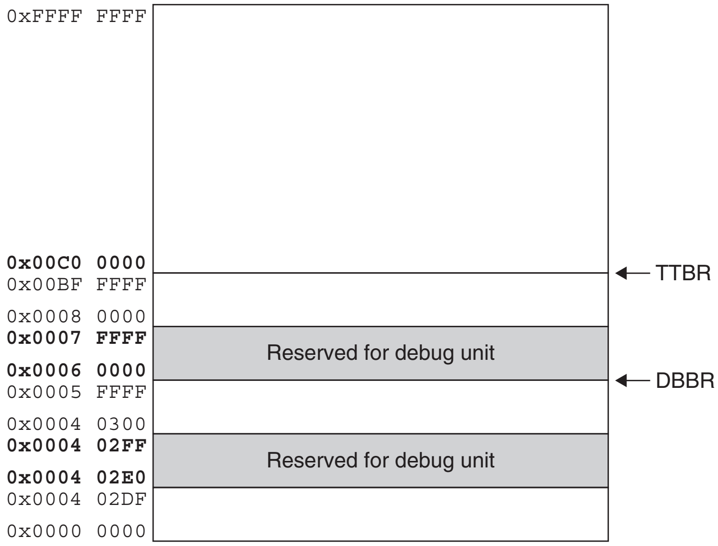


Figure 4.1 C33 PE Address Map

Memories or I/O devices can be mapped anywhere in the address space. Note, however, that the addresses shown below cannot be used for user applications as they are reserved.

0xC00000

This is the default reset vector address (TTBR initial value). The C33 PE Core starts executing the program from the boot address written to this address.

0x402E0–0x402FF, 0x4812D (byte), 0x48134 (word), 0x60000–0x7FFFF

These areas and addresses are reserved for debugging functions. Do not allocate these addresses to memories and I/O devices.

5 Instruction Set

The C33 PE Core instruction set consists of the function-extended instruction set of the C33 STD Core CPU and the new instructions, in addition to the conventional S1C33-series instructions. Some instructions of the C33 STD Core CPU are deleted. As the C33 PE Core is object-code compatible with the C33 STD Core CPU, software assets can be transported from the S1C33 series to the C33 PE model easily, with minimal modifications required.
All of the instruction codes are fixed to 16 bits in length which, combined with pipelined processing, allows most important instructions to be executed in one cycle. For details, refer to the description of each instruction in the latter sections of this manual.

5.1 S1C33-Series-Compatible Instructions

Table 5.1.1 S1C33-Series-Compatible Instructions

Classification	Mnemonic		Function
Arithmetic operation	add	%rd,%rs	Addition between general-purpose registers
		%rd,imm6	Addition of a general-purpose register and immediate
		%sp,imm10	Addition of SP and immediate (with immediate zero-extended)
	adc	%rd,%rs	Addition with carry between general-purpose registers
	sub	%rd,%rs	Subtraction between general-purpose registers
		%rd,imm6	Subtraction of general-purpose register and immediate
		%sp,imm10	Subtraction of SP and immediate (with immediate zero-extended)
	sbc	%rd,%rs	Subtraction with carry between general-purpose registers
	cmp	%rd,%rs	Arithmetic comparison between general-purpose registers
		%rd,sign6	Arithmetic comparison of general-purpose register and immediate (with immediate zero-extended)
	mlt.h	%rd,%rs	Signed integer multiplication (16 bits × 16 bits → 32 bits)
	mltu.h	%rd,%rs	Unsigned integer multiplication (16 bits × 16 bits → 32 bits)
	mlt.w	%rd,%rs	Signed integer multiplication (32 bits × 32 bits → 64 bits)
	mltu.w	%rd,%rs	Unsigned integer multiplication (32 bits × 32 bits → 64 bits)
Branch	jrgt jrgt.d	sign8	PC relative conditional jump Branch condition: !Z & !(N ^ V) Delayed branching possible
	jrge jrge.d	sign8	PC relative conditional jump Branch condition: !(N ^ V) Delayed branching possible
	jrlt jrlt.d	sign8	PC relative conditional jump Branch condition: N ^ V Delayed branching possible
	jrle jrle.d	sign8	PC relative conditional jump Branch condition: Z | N ^ V Delayed branching possible
	jrugt jrugt.d	sign8	PC relative conditional jump Branch condition: !Z & !C Delayed branching possible
	jruge jruge.d	sign8	PC relative conditional jump Branch condition: !C Delayed branching possible
	jrult jrult.d	sign8	PC relative conditional jump Branch condition: C Delayed branching possible
	jrule jrule.d	sign8	PC relative conditional jump Branch condition: Z | C Delayed branching possible
	jreq jreq.d	sign8	PC relative conditional jump Branch condition: Z Delayed branching possible
	jrne jrne.d	sign8	PC relative conditional jump Branch condition: !Z Delayed branching possible
	jp	sign8	PC relative jump Delayed branching possible
	jp.d	%rb	Absolute jump Delayed branching possible
	call	sign8	PC relative subroutine call Delayed call possible
	call.d	%rb	Absolute subroutine call Delayed call possible
	ret ret.d		Subroutine return Delayed return possible
	reti		Return from interrupt or exception handling
	retd		Return from the debug processing routine
	int	imm2	Software exception
	brk		Debug exception

Classification	Mnemonic		Function
Data transfer	ld.b	%rd,%rs	General-purpose register (byte) → general-purpose register (sign-extended)
		%rd,[%rb] %rd,[%rb]+	Memory (byte) → general-purpose register (sign-extended) Postincrement possible
		%rd,[%sp+imm6]	Stack (byte) → general-purpose register (sign-extended)
		[%rb],%rs [%rb]+,%rs	General-purpose register (byte) → memory Postincrement possible
		[%sp+imm6],%rs	General-purpose register (byte) → stack
	ld.ub	%rd,%rs	General-purpose register (byte) → general-purpose register (zero-extended)
		%rd,[%rb] %rd,[%rb]+	Memory (byte) → general-purpose register (zero-extended) Postincrement possible
		%rd,[%sp+imm6]	Stack (byte) → general-purpose register (zero-extended)
	ld.h	%rd,%rs	General-purpose register (halfword) → general-purpose register (sign-extended)
		%rd,[%rb] %rd,[%rb]+	Memory (halfword) → general-purpose register (sign-extended) Postincrement possible
		%rd,[%sp+imm6]	Stack (halfword) → general-purpose register (sign-extended)
		[%rb],%rs [%rb]+,%rs	General-purpose register (halfword) → memory Postincrement possible
		[%sp+imm6],%rs	General-purpose register (halfword) → stack
	ld.uh	%rd,%rs	General-purpose register (halfword) → general-purpose register (zero-extended)
		%rd,[%rb] %rd,[%rb]+	Memory (halfword) → general-purpose register (zero-extended) Postincrement possible
		%rd,[%sp+imm6]	Stack (halfword) → general-purpose register (zero-extended)
	ld.w	%rd,%rs	General-purpose register (word) → general-purpose register
		%rd,sign6	Immediate → general-purpose register (sign-extended)
		%rd,[%rb] %rd,[%rb]+	Memory (word) → general-purpose register Postincrement possible
		%rd,[%sp+imm6]	Stack (word) → general-purpose register
		[%rb],%rs [%rb]+,%rs	General-purpose register (word) → memory Postincrement possible
		[%sp+imm6],%rs	General-purpose register (word) → stack
System control	nop		No operation
	halt		HALT
	slp		SLEEP
Immediate extension	ext	imm13	Extend operand in the following instruction
Bit manipulation	btst	[%rb],imm3	Test a specified bit in memory data
	bclr	[%rb],imm3	Clear a specified bit in memory data
	bset	[%rb],imm3	Set a specified bit in memory data
	bnot	[%rb],imm3	Invert a specified bit in memory data
Other	swap	%rd,%rs	Bytewise swap on byte boundary in word
	pushn	%rs	Push general-purpose registers %rs–%r0 onto the stack
	popn	%rd	Pop data for general-purpose registers %rd–%r0 off the stack

The symbols in the above table each have the meanings specified below.

Table 5.1.2 Symbol Meanings

Symbol	Description
%rs	General-purpose register, source
%rd	General-purpose register, destination
%ss	Special register, source
%sd	Special register, destination
[%rb]	General-purpose register, indirect addressing
[%rb]+	General-purpose register, indirect addressing with postincrement
%sp	Stack pointer
imm2, imm4, imm3, imm5, imm6, imm10, imm13	Unsigned immediate (numerals indicating bit length) However, numerals in shift instructions indicate the number of bits shifted, while those in bit manipulation indicate bit positions.
sign6, sign8	Signed immediate (numerals indicating bit length)

5.2 Function Extended Instructions

Table 5.2.1 Function Extended Instructions

Classification	Mnemonic		Function	Extended function
Logical operation	and	%rd,%rs	Logical AND between general-purpose registers	The V flag is cleared after the instruction has been executed.
		%rd,sign6	Logical AND of general-purpose register and immediate	
	or	%rd,%rs	Logical OR between general-purpose registers	
		%rd,sign6	Logical OR of general-purpose register and immediate	
	xor	%rd,%rs	Exclusive OR between general-purpose registers	
		%rd,sign6	Exclusive OR of general-purpose register and immediate	
	not	%rd,%rs	Logical inversion between general-purpose registers (1's complement)	
		%rd,sign6	Logical inversion of general-purpose register and immediate (1's complement)	
Shift and rotate	srl	%rd,%rs	Logical shift to the right (Bits 0–31 shifted as specified by the register)	For rotate/shift operation, it has been made possible to shift 9–31 bits.
		%rd,imm5	Logical shift to the right (Bits 0–31 shifted as specified by immediate)	
	sll	%rd,%rs	Logical shift to the left (Bits 0–31 shifted as specified by the register)	
		%rd,imm5	Logical shift to the left (Bits 0–31 shifted as specified by immediate)	
	sra	%rd,%rs	Arithmetic shift to the right (Bits 0–31 shifted as specified by the register)	
		%rd,imm5	Arithmetic shift to the right (Bits 0–31 shifted as specified by immediate)	
	sla	%rd,%rs	Arithmetic shift to the left (Bits 0–31 shifted as specified by the register)	
		%rd,imm5	Arithmetic shift to the left (Bits 0–31 shifted as specified by immediate)	
	rr	%rd,%rs	Rotate to the right (Bits 0–31 rotated as specified by the register)	
		%rd,imm5	Rotate to the right (Bits 0–31 rotated as specified by immediate)	
	rl	%rd,%rs	Rotate to the left (Bits 0–31 rotated as specified by the register)	
		%rd,imm5	Rotate to the left (Bits 0–31 rotated as specified by immediate)	
Data transfer	ld.w	%rd,%ss	Special register (word) → general-purpose register	The number of special registers that can be used to load data has been increased.
		%sd,%rs	General-purpose register (word) → special register	

5.3 Instructions Added to the C33 PE Core

Table 5.3.1 Instructions Added to the C33 PE Core

Classification	Mnemonic		Function
Branch	jpr	*%rb*	PC relative jump
	jpr.d		Delayed branching possible
System control	psrset	*imm5*	Set a specified bit in PSR
	psrclr	*imm5*	Clear a specified bit in PSR
Coprocessor control	ld.c	*%rd,imm4*	Load data from coprocessor
	ld.c	*imm4,%rs*	Store data in coprocessor
	do.c	*imm6*	Execute coprocessor
	ld.cf		Load C, V, Z, and N flags from coprocessor
Other	swaph	*%rd,%rs*	Bytewise swap on halfword boundary in word
	push	*%rs*	Push single general-purpose register
	pop	*%rd*	Pop single general-purpose register
	pushs	*%ss*	Push special registers *%ss*–ALR onto the stack
	pops	*%sd*	Pop data for special registers *%sd*–ALR off the stack

5.4 Instructions Removed

Table 5.4.1 Instructions Removed

Classification	Mnemonic		Function
Arithmetic operation	div0s	*%rs*	First step in signed integer division
	div0u	*%rs*	First step in unsigned integer division
	div1	*%rs*	Execution of step division
	div2s	*%rs*	Data correction for the result of signed integer division 1
	div3s		Data correction for the result of signed integer division 2
Other	mirror	*%rd,%rs*	Bitwise swap every byte in word
	mac	*%rs*	Multiply-accumulate operation 16 bits × 16 bits + 64 bits → 64 bits
	scan0	*%rd,%rs*	Search for bits whose value = 0
	scan1	*%rd,%rs*	Search for bits whose value = 1

5.5 Addressing Modes (without `ext` extension)

The instruction set of the C33 PE Core, as with the S1C33 series, has six discrete addressing modes, as described below. The processor determines the addressing mode according to the operand in each instruction before it accesses data.

(1) Immediate addressing
(2) Register direct addressing
(3) Register indirect addressing
(4) Register indirect addressing with postincrement
(5) Register indirect addressing with displacement
(6) Signed PC relative addressing

5.5.1 Immediate Addressing

The immediate included in the instruction code that is indicated as *immX* (unsigned immediate) or *signX* (signed immediate) is used as the source data. The immediate size specifiable in each instruction is indicated by a numeral in the symbol (e.g., *imm4* = unsigned 4 bits; *sign6* = signed 6 bits). For signed immediates such as *sign6*, the most significant bit is the sign bit, which is extended to 32 bits when the instruction is executed.

Example: `ld.w   %r0,0x30`

Before execution   r0 = 0x*XXXXXXXX*
After execution   r0 = 0xFFFFFFF0

The immediate *sign6* can represent values in the range of +31 to -32 (0b011111 to 0b100000).

Except in the case of shift-related and bit-manipulating instructions, immediate data can be extended to a maximum of 32 bits by a combined use of the operand value and the `ext` instruction.

Example:
```
ext    imm13   (1)
ext    imm13   (2)
ld.w   %r0,sign6
```

r0 after execution

	31 – 19	18 – 6	5 – 0
r0	*imm13* (1)	*imm13* (2)	*sign6*

5.5.2 Register Direct Addressing

The content of a specified register is used directly as the source data. Furthermore, if this addressing mode is specified as the destination for an instruction that loads the result in a register, the result is loaded in this specified register. The instructions that have the following symbols as the operand are executed in this addressing mode.

%rs *rs* is a metasymbol indicating the general-purpose register that holds the source data to be operated on or transferred. The register is actually written as `%r0`, `%r1`, ... or `%r15`.

%rd *rd* is a metasymbol indicating the general-purpose register that is the destination for the result of operation. The register is actually written as `%r0`, `%r1`, ... or `%r15`. Depending on the instruction, it will also be used as the source data.

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

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

Actual special register names are written as follows:

Processor status register	`%psr`
Stack pointer	`%sp`
Arithmetic operation low register	`%alr`
Arithmetic operation high register	`%ahr`
Trap table base register	`%ttbr`

The register names are always prefixed by "`%`" to discriminate them from symbol names, label names, and the like.

5.5.3 Register Indirect Addressing

In this mode, memory is accessed indirectly by specifying a general-purpose register that holds the address needed. This addressing mode is used only for load instructions that have `[%rb]` as the operand. Actually, this general-purpose register is written as `[%r0]`, `[%r1]`, ... or `[%r15]`, with the register name enclosed in brackets "`[]`."
The processor refers to the content of a specified register as the base address, and transfers data in the format that is determined by the type of load instruction.

Examples: Memory → Register

```
ld.b  %r0,[%r1]
ld.h  %r0,[%r1]
ld.w  %r0,[%r1]
```

Register → Memory

```
ld.b  [%r1],%r0
ld.h  [%r1],%r0
ld.w  [%r1],%r0
```

In this example, the address indicated by r1 is the memory address from or to which data is to be transferred.

In halfword and word transfers, the base address that is set in a register must be on a halfword boundary (least significant address bit = 0) or word boundary (2 low-order address bits = 0), respectively. Otherwise, an address-misaligned exception will be generated.

5.5.4 Register Indirect Addressing with Postincrement

As in register indirect addressing, the memory location to be accessed is specified indirectly by a general-purpose register. When a data transfer finishes, the base address held in a specified register is incremented* by an amount equal to the transferred data size. In this way, data can be read from or written to continuous addresses in memory only by setting the start address once at the beginning.

* Increment size

Byte transfer (`ld.b`, `ld.ub`):	$rb \rightarrow rb + 1$
Halfword transfer (`ld.h`, `ld.uh`):	$rb \rightarrow rb + 2$
Word transfer (`ld.w`):	$rb \rightarrow rb + 4$

This addressing mode is specified by enclosing the register name in brackets "`[]`," which is then suffixed by "+."
The register name is actually written as `[%r0]+`, `[%r1]+`, ... or `[%r15]+`.

5.5.5 Register Indirect Addressing with Displacement

In this mode, memory is accessed beginning with the address that is derived by adding a specified immediate (displacement) to the register content. Unless `ext` instructions are used, this addressing mode can only be used for load instructions that have `[%sp+imm6]` as the operand.

Examples: `ld.b %r0,[%sp+0x10]`

The byte data at the address derived by adding 0x10 to the content of the current SP is loaded into the R0 register. For byte data transfers, the 6-bit immediate is added directly as the displacement.

`ld.h %r0,[%sp+0x10]`

The halfword data at the address derived by adding 0x20 to the content of the current SP is loaded into the R0 register. For halfword data transfers, because halfword boundary addresses are accessed, twice the 6-bit immediate (least significant bit always 0) is the displacement.

`ld.w %r0,[%sp+0x10]`

The word data at the address derived by adding 0x40 to the content of the current SP is loaded into the R0 register. For word data transfers, because word boundary addresses are accessed, four times the 6-bit immediate (2 low-order bits always 0) is the displacement.

If `ext` instructions described in Section 5.6 are used, ordinary register indirect addressing (`[%rb]`) becomes a special addressing mode in which the immediate specified by the `ext` instruction constitutes the displacement.

Example:
```
ext   imm13
ld.b  %rd,[%rb]        The memory address to be accessed is "%rb+imm13."
```

5.5.6 Signed PC Relative Addressing

This addressing mode is used for branch instructions that have a signed 8-bit immediate (*sign8*) in their operand. When these instructions are executed, the program branches to the address derived by adding twice the *sign8* value (halfword boundary) to the current PC.

Example:
```
PC + 0   jrne   0x04     The program branches to the PC + 8 address when the jrne branch
  :         :            condition holds true.
  :         :            (PC + 0) + 0x04 * 2 → PC + 8
PC + 8
```

5.6 Addressing Modes with `ext`

The immediate specifiable in 16-bit, fixed-length instruction code is specified in a bit field of a length ranging from 4 bits to 8 bits, depending on the instruction used. The `ext` instructions are used to extend the size of this immediate.

The `ext` instructions are used in combination with data transfer or arithmetic/logic instructions, and is placed directly before the instruction whose immediate needs to be extended. The instruction is expressed in the form `ext` *`imm13`*, in which the immediate size extendable by one `ext` instruction is 13 bits and up to two `ext` instructions can be written in succession to extend the immediate further.

The `ext` instructions are effective only for the instructions for which the immediate extension written directly after `ext` is possible, and have no effect for all other instructions. When three or more `ext` instructions have been described sequentially, an undefined instruction exception (`ext` exception) occurs before executing the extension target instruction.

When an instruction, which does not support the extension in the `ext` instruction, follows an `ext`, the `ext` instruction will be executed as a `nop` instruction.

5.6.1 Extension of Immediate Addressing

Extension of *imm6*

The *imm6* immediate is extended to a 19-bit or 32-bit immediate.

Extending to a 19-bit immediate

To extend the immediate to 19-bit quantity, enter one `ext` instruction directly before the target instruction.

Example:
```
ext   imm13
add   %rd,imm6
```

Extended immediate

31–19	18–6	5–0
0 0 0 0 0 0 0 0 0 0 0 0 0	*imm13*	*imm6*

Bits 31–19 are filled with 0 (zero-extension).

Extending to a 32-bit immediate

To extend the immediate to 32-bit quantity, enter two `ext` instructions directly before the target instruction.

Example:
```
ext   imm13 (1)
ext   imm13 (2)
sub   %rd,imm6
```

Extended immediate

31–19	18–6	5–0
imm13 (1)	*imm13* (2)	*imm6*

Extension of *sign6*

The *sign6* immediate is extended to a sign-extended 19-bit or 32-bit immediate.

Extending to a 19-bit immediate

To extend the immediate to 19-bit quantity, enter one `ext` instruction directly before the target instruction.

Example:
```
ext    imm13
ld.w   %rd,sign6
```

Extended immediate

31–19	18–6	5–0
S S S S S S S S S S S S S	S *imm13*	*sign6*

The most significant bit "S" in *imm13* that has been extended by the `ext` instruction is the sign, with which bits 31–19 are extended to become signed 19-bit data. The most significant bit in *sign6* is handled as the MSB data of 6-bit data, and not as the sign.

Extending to a 32-bit immediate

To extend the immediate to 32-bit quantity, enter two `ext` instructions directly before the target instruction.

Example:
```
ext   imm13 (1)
ext   imm13 (2)
and   %rd,sign6
```

Extended immediate

31	19 18	6 5	0
S	imm13 (1)	imm13 (2)	sign6

The MSB (bit 12) in the first `ext` instruction is the sign, with the immediate extended to become signed 32-bit data.

5.6.2 Extension of Register Indirect Addressing

Adding displacement to [%rb]

Memory is accessed at the address derived by adding the immediate specified by an `ext` instruction to the address that is indirectly referenced by `[%rb]`.

Adding a 13-bit immediate

Memory is accessed at the address derived by adding the 13-bit immediate specified by *imm13* to the address specified by the *rb* register. During address calculation, *imm13* is zero-extended to 32-bit quantity.

Example:
```
ext    imm13
ld.b   %rd,[%rb]
```

	31 … 13	12 … 0
rb	Memory address pointer	
+		
Immediate	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	*imm13*

Adding a 26-bit immediate

Memory is accessed at the address derived by adding the 26-bit immediate specified by *imm26* to the address specified by the *rb* register. During address calculation, *imm26* is zero-extended to 32-bit quantity.

Example:
```
ext     imm13 (1)
ext     imm13 (2)
ld.uh   %rd,[%rb]
```

	31 … 26	25 … 13	12 … 0
rb	Memory address pointer		
+			
Immediate	0 0 0 0 0 0	*imm13* (1)	*imm13* (2)

Extending `[%sp+`*`imm6`*`]` displacement

The immediate (*imm6*) in displacement-added register indirect addressing instructions is extended. Be aware that *imm6* is handled differently in single instructions with no `ext` instructions added.
Displacement-added register indirect addressing instructions, when used singly, automatically calculate a boundary address according to the data size to be transferred by the instruction.
Example: `ld.h   %rd,[%sp+imm6]`

The address referenced in this example is the "`%sp+imm6*2`" address on a halfword boundary.

For addressing with `ext` instructions added, refer to the description below.

Extending to a 19-bit immediate

To extend the immediate to 19-bit quantity, enter one `ext` instruction directly before the target instruction. The immediate that is extended to 19-bit quantity has its low-order bits fixed to "0" or "00" according to the transferred data size. (This applies to other than byte transfers.)

Examples:
```
ext    imm13
ld.b   %rd,[%sp+imm6]

ext    imm13
ld.h   [%sp+imm6],%rs
```

Extended immediate

	31 … 19	18 … 6	5 … 0
Byte transfer	0 0 0 0 0 0 0 0 0 0 0 0 0	*imm13*	*imm6*
Halfword transfer	0 0 0 0 0 0 0 0 0 0 0 0 0	*imm13*	*imm6*[5:1] | 0
Word transfer	0 0 0 0 0 0 0 0 0 0 0 0 0	*imm13*	*imm6*[5:2] | 0 | 0

The extended data and the sp are added to comprise the source or destination address of transfer.

Extending to a 32-bit immediate

To extend the immediate to 32-bit quantity, enter two `ext` instructions directly before the target instruction. The immediate that is extended to 32-bit quantity has its low-order bits fixed to "0" or "00" according to the transferred data size. (This applies to other than byte transfers.)

Examples:
```
ext    imm13 (1)
ext    imm13 (2)
ld.b   %rd,[%sp+imm6]

ext    imm13 (1)
ext    imm13 (2)
ld.h   [%sp+imm6],%rs
```

Extended immediate

	31 … 19	18 … 6	5 … 0
Byte transfer	*imm13* (1)	*imm13* (2)	*imm6*
Halfword transfer	*imm13* (1)	*imm13* (2)	*imm6*[5:1] | 0
Word transfer	*imm13* (1)	*imm13* (2)	*imm6*[5:2] | 0 | 0

The extended data and the sp are added to comprise the source or destination address of transfer.

Extending register-to-register operation instructions

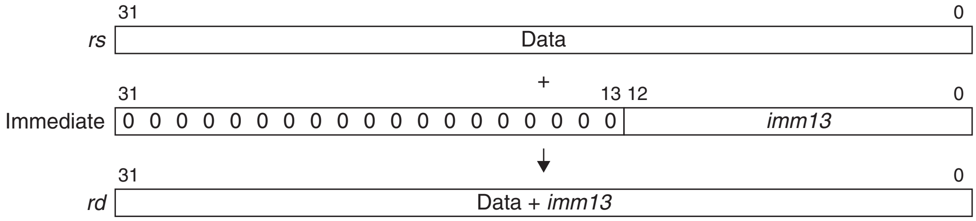
Register-to-register operation instructions are extended by one or two `ext` instructions. Unlike data transfer instructions, these instructions add or subtract the content of the *rs* register and the immediate specified by an `ext` instruction according to the arithmetic operation to be performed. They then store the result in the *rd* register. The content of the *rd* register does not affect the arithmetic operation performed. An example of how to extend for an add operation is shown below.

Extending to *rs* + *imm13*

To extend to *rs* + *imm13*, enter one `ext` instruction directly before the target instruction.

Example:
```
ext   imm13
add   %rd,%rs
```

If not extended, *rd* = *rd* + *rs*
When extended by one `ext` instruction, *rd* = *rs* + *imm13*

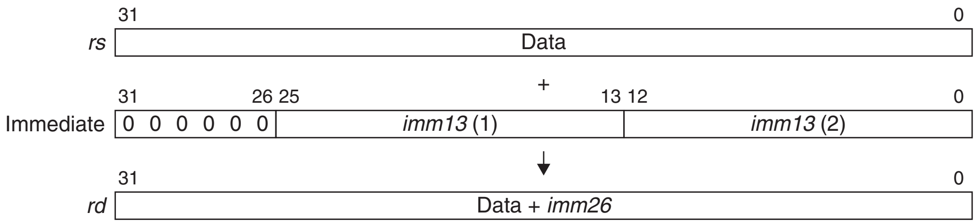


Extending to *rs* + *imm26*

To extend to *rs* + *imm26*, enter two `ext` instructions directly before the target instruction.

Example:
```
ext   imm13 (1)
ext   imm13 (2)
add   %rd,%rs
```

If not extended, *rd* = *rd* + *rs*
When extended by two `ext` instructions, *rd* = *rs* + *imm26*



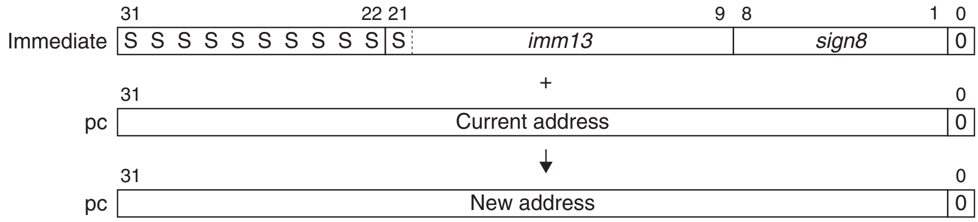
Extending the displacement of PC relative branch instructions

The *sign8* immediate in PC relative branch instructions is extended to a signed 22-bit or a signed 32-bit immediate. The *sign8* immediate in PC relative branch instructions is multiplied by 2 for conversion to a relative value for the jump address, and the derived value is then added to PC to determine the jump address. The `ext` instructions extend this relative jump address value.

Extending to a 22-bit immediate

To extend the *sign8* immediate to a 22-bit immediate, enter one `ext` instruction directly before the target instruction.

Example:
```
ext   imm13
jrgt  sign8
```

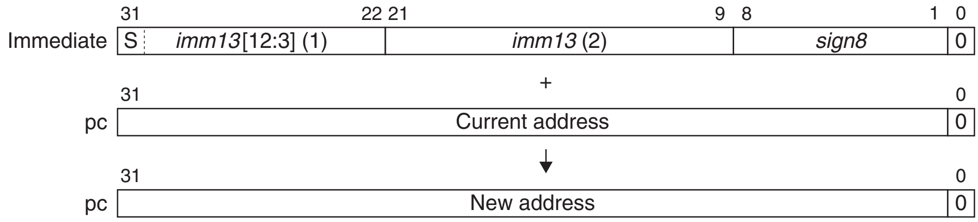


The most significant bit “S” in the immediate that has been extended by the `ext` instruction is the sign, with which bits 31–22 are extended to become signed 22-bit data. The most significant bit in *sign8* is handled as the MSB data of 8-bit data, and not as the sign.

Extending to a 32-bit immediate

To extend the *sign8* immediate to a 32-bit immediate, enter two `ext` instructions directly before the target instruction.

Example:
```
ext   imm13 (1)
ext   imm13 (2)
jrgt  sign8
```



The most significant bit “S” in the immediate that has been extended by `ext` instructions is the sign. Bits 2–0 in the first `ext` instruction are unused.

5.6.3 Exception Handling for `ext` Instructions

For exceptions associated with `ext` instructions, exception handling is started immediately for reset and debug break, but is not started for other exceptions until after the target instruction to be extended is executed. This is intended to simplify operation for the compression of `ext` instructions in prefetch. Furthermore, as the address to which the program is returned by `reti` or `retd` at the end of exception handling is the `ext` instruction, in no case will the `ext` instructions operate erratically due to exception handling. (For two `ext` instructions, control returns to the first `ext`.)

5.7 Data Transfer Instructions

The transfer instructions in the C33 PE Core support data transfer between one register and another, as well as between a register and memory. A transfer data size and data extension format can be specified in the instruction code. In mnemonics, this specification is classified as follows:

`ld.b`	Signed byte data transfer
`ld.ub`	Unsigned byte data transfer
`ld.h`	Signed halfword data transfer
`ld.uh`	Unsigned halfword data transfer
`ld.w`	Word data transfer

In signed byte or halfword transfers to registers, the source data is sign-extended to 32 bits. In unsigned byte or halfword transfers, the source data is zero-extended to 32 bits.
In transfers in which data is transferred from registers, data of a specified size on the lower side of the register is the data to be transferred.

If the destination of transfer is a general-purpose register, the register content after a transfer is as follows:

Signed byte data transfer

	31 … 24	23 … 16	15 … 8	7 … 0
rd	S S S S S S S S	S S S S S S S S	S S S S S S S S	S Byte data

Extended with the sign in bit 7 of the byte data

Unsigned byte data transfer

	31 … 24	23 … 16	15 … 8	7 … 0
rd	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Byte data

Signed halfword data transfer

	31 … 16	15 … 0
rd	S S S S S S S S S S S S S S S S	S Halfword data

Extended with the sign in bit 15 of the halfword data

Unsigned halfword data transfer

	31 … 16	15 … 0
rd	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Halfword data

5.8 Logical Operation Instructions

Four discrete logical operation instructions are available for use with the C33 PE Core.

and	Logical AND
or	Logical OR
xor	Exclusive-OR
not	Logical NOT

All logical operations are performed in a specified general-purpose register (R0–R15). The source is one of two, either 32-bit data in a specified general-purpose register or signed immediate data (6, 19, or 32 bits).

Differences from the C33 STD Core CPU

When a logical operation is performed, the V flag (bit 2) in the PSR is cleared.

5.9 Arithmetic Operation Instructions

The instruction set of the C33 PE Core supports add/subtract, compare, and multiply instructions for arithmetic operations. (The multiply instructions are described in the next section.)

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

The above arithmetic operations are performed between one general-purpose register and another (R0–R15), or between a general-purpose register and an immediate. Furthermore, the `add` and `sub` instructions can perform operations between the SP and immediate. Immediates in sizes smaller than word, except for the `cmp` instruction, are zero-extended when operation is performed.
The `cmp` instruction compares two operands, and may alter a flag, depending on the comparison result. Basically, it is used to set conditions for conditional jump instructions. If an immediate smaller than word in size is specified as the source, it is sign-extended when comparison is performed.

5.10 Multiply Instructions

The instruction set of the C33 PE Core includes four multiplication instructions.

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

The data in the specified general-purpose registers (R0–R15) is used for the multiplier and the multiplicand, respectively. For 16-bit multiplications, the 16 low-order bits in the specified register are used. The signed multiplication instructions use the MSB in the multiplier and multiplicand as the sign bit.
The result of a 16-bit × 16-bit operation is loaded into the ALR. The result of a 32-bit × 32-bit operation is loaded into the AHR and ALR, with the 32 high-order bits stored in the former and the 32 low-order bits stored in the latter.
The C33 PE Core executes 16-bit × 16-bit multiplication in five cycle and 32-bit × 32-bit multiplication in seven cycles.

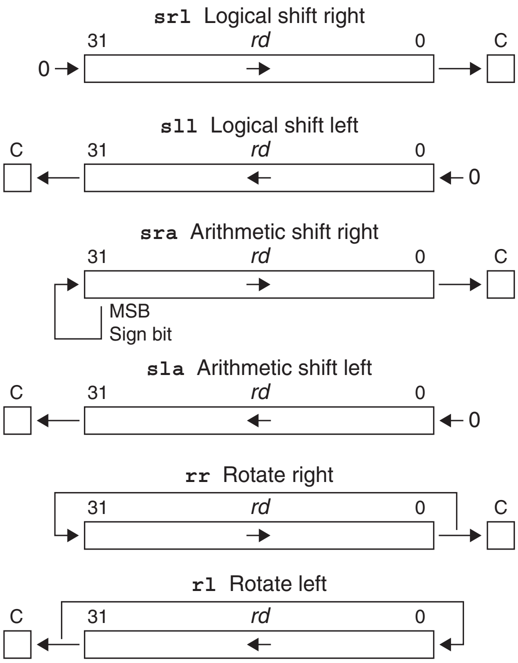
5.11 Shift and Rotate Instructions

The instruction set of the C33 PE Core supports instructions to shift or rotate the register data.

`srl`	Logical shift right
`sll`	Logical shift left
`sra`	Arithmetic shift right
`sla`	Arithmetic shift left
`rr`	Rotate right
`rl`	Rotate left

The number of bits that can be shifted has been increased from the conventional 8 bits to 32 bits. Because 32-bit shift is supported, new instructions have been added with extended functions. The number of bits to be shifted can be specified in the range of 0 to 31 using the operand *imm5* or the *rs* register.

Example: `srl %rd,imm5`   Bits 0–31 logically shifted to the right
`srl %rd,%rs`   Bits 0–31 logically shifted to the right



The table below lists the number of bits shifted as specified by the *rs* register or the operand *imm5*.

Table 5.11.1 Number of Bits Shifted as Specified by *imm5* or *rs*

imm5 *rs*[5:0]	Number of bits to be shifted	*imm5* *rs*[5:0]	Number of bits to be shifted
00000	0	10000	16
00001	1	10001	17
00010	2	10010	18
00011	3	10011	19
00100	4	10100	20
00101	5	10101	21
00110	6	10110	22
00111	7	10111	23
01000	8	11000	24
01001	9	11001	25
01010	10	11010	26
01011	11	11011	27
01100	12	11100	28
01101	13	11101	29
01110	14	11110	30
01111	15	11111	31

Bits 5–31 in the *rs* are not used.

5.12 Bit Manipulation Instructions

The following four instructions are provided for manipulating the data in memory bitwise or one bit at a time. These instructions allow the display memory or I/O map control bits to be altered directly.

btst	**[%*rb*],*imm3***	Set the Z flag if a specified bit = 0
bclr	**[%*rb*],*imm3***	Clear a specified bit to 0
bset	**[%*rb*],*imm3***	Set a specified bit to 1
bnot	**[%*rb*],*imm3***	Invert a specified bit (1 ↔ 0)

Bit manipulation is performed on the memory address specified by the *rb* (general-purpose) register. *imm3* specifies a bit number (bits 0–7) in the byte data stored in that address location.
Although the content of memory data altered by these instructions (except `btst`) is only the specified bit, the specified address is rewritten because memory is accessed bytewise. Therefore, if the addresses to be manipulated have any I/O control bits mapped whose function is enabled by a bit write operation, use of these instructions requires caution.

5.13 Push and Pop Instructions

The push and pop instructions are provided to temporarily save the contents of general-purpose or special registers to the stack, and to restore the saved register data from the stack.

Push instructions **`pushn %rs`**
`push %rs`
`pushs %ss`

The `pushn` instruction saves a range of general-purpose registers from *rs* to R0 to the stack successively. The `push` instruction saves the general-purpose register specified by *rs* to the stack singly. The `pushs` instruction saves the special registers (ALR only or AHR and ALR).

Pop instructions **`popn %rd`**
`pop %rd`
`pops %sd`

The `popn` instruction restores the saved data from the stack to the general-purpose registers R0 to *rd* successively. The `pop` instruction restores the saved data from the stack to the general-purpose register specified by *rd* singly. The `pops` instruction restores the saved data from the stack to the special registers (ALR only or ALR and AHR).

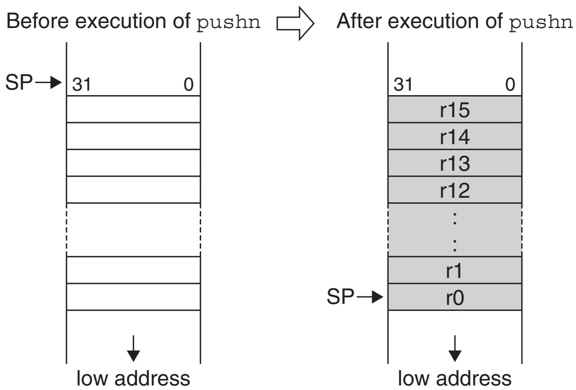
The push and pop instructions must have the same register specification in pairs. These instructions alter the SP depending on the number of pieces of data that are saved and restored. Because in addition to the push/pop instructions, load instructions are available for register indirect addressing with displacement (`[%sp+imm6]`) where the SP is the base address, individual store/load operations on each register can be performed with respect to the SP. In this case, however, the SP is not altered.

A specific register number is assigned to each register (refer to Chapter 2, "Registers"). When general-purpose or special registers are successively pushed, their data is saved to the stack in descending order of register numbers beginning with the one specified by *rs* or *ss*. In successive pop operations, conversely, the register data is restored in ascending order from R0 or ALR up to the specified register.

Differences from the C33 STD Core CPU

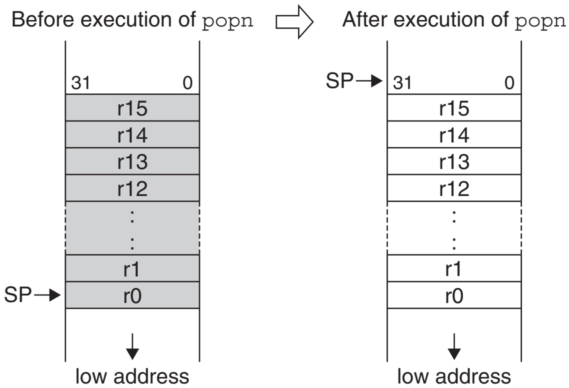
- General-purpose-register single push/pop instructions have been added.
  `push %rs` `pop %rd`
- Special-register successive push/pop instructions have been added.
  `pushs %ss` `pops %sd`

Example 1: `pushn %r15` Push all general-purpose registers onto the stack
`popn %r15` Pop all general-purpose registers off the stack



The stack pointer is updated before the register data is pushed onto the stack.
SP = SP - 4, *rs* → [SP]

Figure 5.13.1 Successive Push of General-Purpose Registers



Data is popped off the stack into the registers before the stack pointer is updated.

[SP] → *rd*, SP = SP + 4

Figure 5.13.2 Successive Pop of General-Purpose Registers

Example 2: `pushs %ahr` Push special registers onto the stack successively
`pops %ahr` Pop special registers off the stack successively

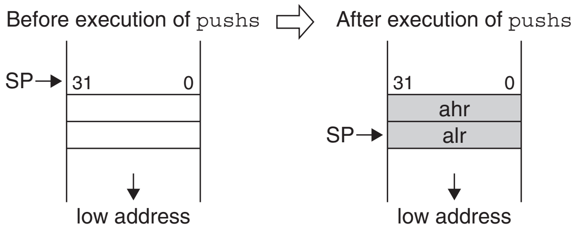


Figure 5.13.3 Successive Push of Special Registers

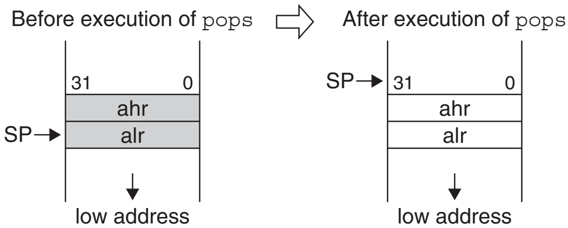


Figure 5.13.4 Successive Pop of Special Registers

Example 3: `push %rs` Push any general-purpose register onto the stack
`pop %rd` Pop any general-purpose register off the stack

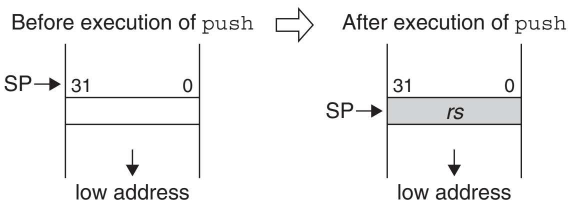


Figure 5.13.5 Single Push of a General-Purpose Register

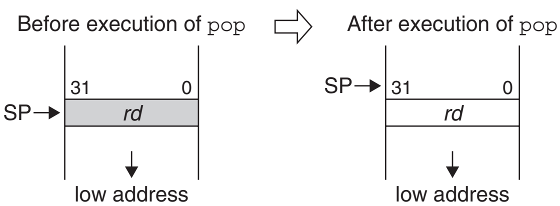


Figure 5.13.6 Single Pop of a General-Purpose Register

5.14 Branch and Delayed Branch Instructions

5.14.1 Types of Branch Instructions

(1) PC relative jump instructions

PC relative jump instructions include the following:

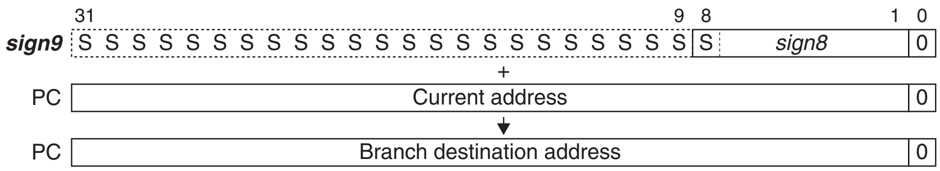
```
jr*   sign8
jp    sign8
jpr   %rb
```

PC relative jump instructions are provided for relocatable programming, so that the program branches to an address that is the same as the address indicated by the current PC (the address at which the branch instruction is located) plus a signed displacement specified by the operand.
The number of instruction steps to the jump address is specified for *sign8* or *rb*. However, since the instruction length in the C33 PE Core is fixed to 16 bits, the value of *sign8* or *rb* is doubled to become a halfword address in 16-bit units. Therefore, the displacement actually added to the PC is a signed 9-bit quantity derived by doubling *sign8* (least significant bit always 0).
The specifiable displacement can be extended by the `ext` instruction, as shown below.

For branch instructions used singly

`jp sign8` Functions as "`jp sign9`" (*sign9* = {*sign8*, 0})

For branch instructions that are used singly, a signed 8-bit displacement (*sign8*) can be specified.

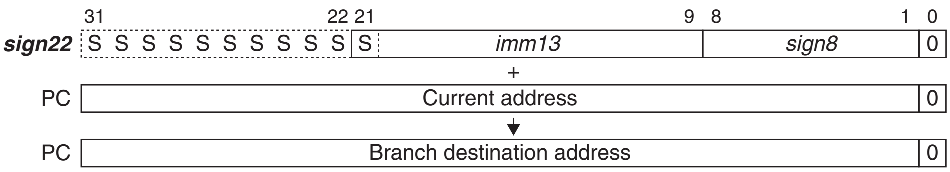


Since *sign8* is a relative value in 16-bit units, the range of addresses to which jumped is (PC - 256) to (PC + 254).

When extended by one `ext` instruction

```
ext   imm13
jp    sign8
```
Functions as "`jp sign22`" (*sign22* = {*imm13*, *sign8*, 0})

The *imm13* specified by the `ext` instruction is extended as the 13 high-order bits of *sign22*.



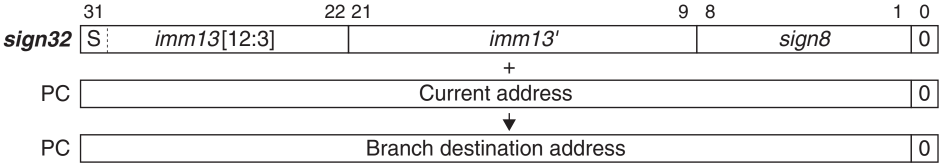
The range of addresses to which jumped is (PC - 2,097,152) to (PC + 2,097,150).

When extended by two `ext` instructions

```
ext   imm13
ext   imm13'
jp    sign8
```
`jp sign8` functions as "`jp sign32`"

The *imm13* specified by the first `ext` instruction is effective for only 10 bits, from bit 12 to bit 3 (with the 3 low-order bits ignored), so that *sign32* is configured as follows:

sign32 = {*imm13*[12:3], *imm13*', *sign8*, 0}



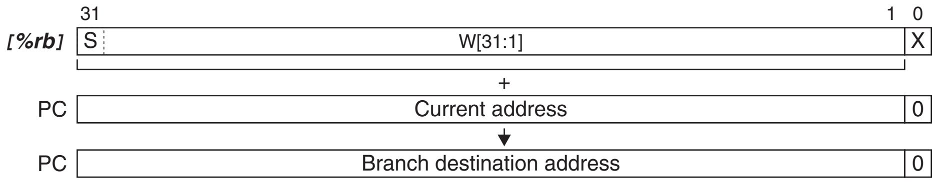
The range of addresses to which jumped is (PC - 2,147,483,648) to (PC + 2,147,483,646).
The above range of addresses to which jumped is a theoretical value, and is actually limited by the range of memory areas used.

For `jpr` branch

```
jpr   %rb
```

A signed 32-bit relative value is specified for *rb*.
The jump address is configured as follows:

{*rb*[31:1], 0}



The least significant bit in the *rb* register is always handled as 0.
The range of addresses to which jumped is (PC - 2,147,483,648) to (PC + 2,147,483,646).
The above range of addresses to which jumped is a theoretical value, and is actually limited by the range of memory areas used.

Branch conditions

The `jp` and `jpr` instructions are unconditional jump instructions that always cause the program to branch.
Instructions with names beginning with `jr` are conditional jump instructions for which the respective branch conditions are set by a combination of flags, so that only when the conditions are satisfied do they cause the program to branch to a specified address. The program does not branch unless the conditions are satisfied.
The conditional jump instructions basically use the result of the comparison of two values by the `cmp` instruction to determine whether to branch. For this reason, the name of each instruction includes a character that represents relative magnitude.
The types of conditional jump instructions and branch conditions are listed in Table 5.14.1.1.

Table 5.14.1.1 Conditional Jump Instructions and Branch Conditions

Instruction		Flag condition	Comparison of A:B	Remark
`jrgt`	Greater Than	!Z & !(N ^ V)	A > B	Used to compare signed data
`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	Used to compare unsigned data
`jruge`	Unsigned, Greater or Equal	!C	A ≥ B	
`jrult`	Unsigned, Less Than	C	A < B	
`jrule`	Unsigned, Less or Equal	Z | C	A ≤ B	
`jreq`	Equal	Z	A = B	
`jrne`	Not Equal	!Z	A ≠ B	

Comparison of A:B made when "`cmp A,B`"

(2) Absolute jump instructions

The absolute jump instruction `jp %rb` causes the program to unconditionally branch to the location indicated by the content of a specified general-purpose register (*rb*) as the absolute address. When the content of the *rb* register is loaded into the PC, its least significant bit is always made 0.

(3) PC relative call instructions

The PC relative call instruction `call sign8` is a subroutine call instruction that is useful for relocatable programming, as it causes the program to unconditionally branch to a subroutine starting from an address that is the same as the address indicated by the current PC (the address at which the branch instruction is located) plus a signed displacement specified by the operand. During branching, the program saves the address of the instruction next to the `call` instruction (for delayed branching, the address of the second instruction following call) to the stack as the return address. When the `ret` instruction is executed at the end of the subroutine, this address is loaded into the PC, and the program returns to it from the subroutine.
Note that because the instruction length is fixed to 16 bits, the least significant bit of the displacement is always handled as 0 (*sign8* doubled), causing the program to branch to an even address.
As with the PC relative jump instructions, the specifiable displacement can be extended by the `ext` instruction. For details on how to extend the displacement, refer to the "(1) PC relative jump instructions."

(4) Absolute call instructions

The absolute call instruction `call %rb` causes the program to unconditionally call a subroutine starting from the location indicated by the content of a specified general-purpose register (*rb*) as the absolute address. When the content of the *rb* register is loaded into the PC, its least significant bit is always made 0. (Refer to the "(2) Absolute jump instructions.")

(5) Software exceptions

The software exception `int imm2` is an instruction that causes the software to generate an exception, by which a specified exception handler routine can be executed. Four distinct exception handler routines can be created, with the respective vector numbers specified by *imm2*. When a software exception occurs, the processor saves the PSR and the instruction address next to `int` to the stack, and reads a specified vector from the vector table in order to execute an exception handler routine. Therefore, to return from the exception handler routine, the `reti` instruction must be used, as it restores the PSR as well as the PC from the stack. For details on the software exception, refer to Section 6.3, "Interrupts and Exceptions."

(6) Return instructions

The `ret` instruction, which is a return instruction for the `call` instruction, loads the saved return address from the stack into the PC as it terminates the subroutine. Therefore, the value of the SP when the `ret` instruction is executed must be the same as when the subroutine was executed (i.e., one that indicates the return address).

The `reti` instruction is a return instruction for the exception handler routine. Since the PSR is saved to the stack along with the return address in exception handling, the content of the PSR must be restored from the stack using the `reti` instruction. In the `reti` instruction, the PC and the PSR are read out of the stack in that order. As in the case of the `ret` instruction, the value of the SP when the `reti` instruction is executed must be the same as when the subroutine was executed.

(7) Debug exceptions

The `brk` and `retd` instructions are used to call a debug exception handler routine, and to return from that routine. Since these instructions are basically provided for the debug firmware, please do not use them in application programs. For details on the functionality of these instructions, refer to Section 6.5, "Debug Circuit."

Differences from the C33 STD Core CPU

Register indirect relative branch instructions have been added.

5.14.2 Delayed Branch Instructions

The C33 PE Core uses pipelined instruction processing, in which instructions are executed while other instructions are being fetched. In a branch instruction, because the instruction that follows it has already been fetched when it is executed, the execution cycles of the branch instruction can be reduced by one cycle by executing the prefetched instruction before the program branches. This is referred to as a delayed branch function, and the instruction executed before branching (i.e., the instruction at the address next to the branch instruction) is referred to as a delayed slot instruction.

The delayed branch function can be used in the instructions listed below, which in mnemonics is identified by the extension "`.d`" added to the branch instruction name.

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**	**jpr.d**

Delayed slot instructions

It is necessary that the delayed slot instructions satisfy all of the following conditions:

- 1-cycle instruction
- Do not access memory
- Not extended by an `ext` instruction

The instructions listed below can be used as delayed slot instructions:

```
ld.b    %rd,%rs
ld.ub   %rd,%rs
ld.h    %rd,%rs
ld.uh   %rd,%rs
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
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,imm5
sll     %rd,%rs      sll     %rd,imm5
sra     %rd,%rs      sra     %rd,imm5
sla     %rd,%rs      sla     %rd,imm5
rr      %rd,%rs      rr      %rd,imm5
rl      %rd,%rs      rl      %rd,imm5
swap    %rd,%rs      swaph   %rd,%rs
ld.c    %rd,imm4
ld.c    imm4,%rs
```

Note: Unless the above conditions are satisfied, the instruction may operate unstably. Therefore, it is prohibited to use such instructions as delayed slot instructions.

A delayed slot instruction is always executed regardless of whether the delayed branch instruction used is conditional or unconditional and whether it branches.
In "non-delayed" branch instructions (those not followed by the extension "`.d`"), the instruction at the address next to the branch instruction is not executed if the program branches; however, if it is a conditional jump and the program does not branch, the instruction at the next address is executed as the one that follows the branch instruction.

The return address saved to the stack by the `call.d` instruction becomes the address for the next instruction following the delayed slot instruction, so that the delayed slot instruction is not executed when the program returns from the subroutine.

No interrupts or exceptions occur in between a delayed branch instruction and a delayed slot instruction, as they are masked out by hardware.

Application for leaf subroutines

The following shows an example application of delayed branch instructions for achieving a fast leaf subroutine call.
Example:

```
        jp.d  SUB         ; Jumps to a subroutine by a delayed branch instruction
        ld.w  %r8,%pc     ; Loads the return address into a general-purpose register by
                          ; a delayed slot instruction
        add   %r1,%r2     ; Return address
         :     :
    SUB:
         :     :
        jp    %r8         ; Return
```

Note: The `ld.w %rd,%pc` instruction must be executed as a delayed slot instruction. If it does not follow a delayed branch instruction, the PC value that is loaded into the *rd* register may not be the next instruction address to the `ld.w` instruction.

5.15 System Control Instructions

The following three instructions are used to control the system. They do not affect the registers or memory.

`nop`	Only increments the PC, with no other operations performed
`halt`	Places the processor in HALT mode
`slp`	Places the processor in SLEEP mode

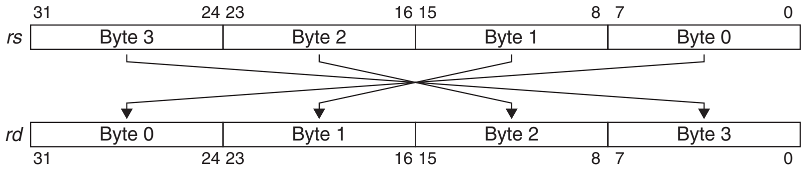
For details on HALT and SLEEP modes, refer to Section 6.4, "Power-Down Mode," and the Technical Manual for each S1C33 model.

5.16 Swap Instructions

The swap instructions replace the contents of general-purpose registers with each other, as shown below.

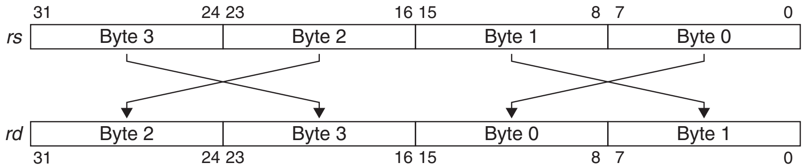
swap ***%rd,%rs***

Big and little endians are converted on a word boundary.



swaph ***%rd,%rs***

The 32-bit data in general-purpose registers has its big and little endians converted on a halfword boundary.



Differences from the C33 STD Core CPU

The `swaph` instruction has been added.

```
swaph  %rd,%rs
```

5.17 Other Instructions

Flag control instructions

The C33 PE Core has had new instructions added that enable the PSR flags to be manipulated directly. As these flag control instructions can set and clear flags bitwise, it is possible to control interrupts by enabling or disabling in one instruction.

psrset ***imm5*** Sets the PSR bit specified by *imm5*[2:0] (0–4) to 1

psrclr ***imm5*** Clears the PSR bit specified by *imm5*[2:0] (0–4) to 0
The contents of PSR are not altered when the *imm5* is 5 or more.

6 Functions

This chapter describes the processing status of the C33 PE Core and outlines the operation.

6.1 Transition of the Processor Status

The diagram below shows the transition of the operating status in the C33 PE Core.

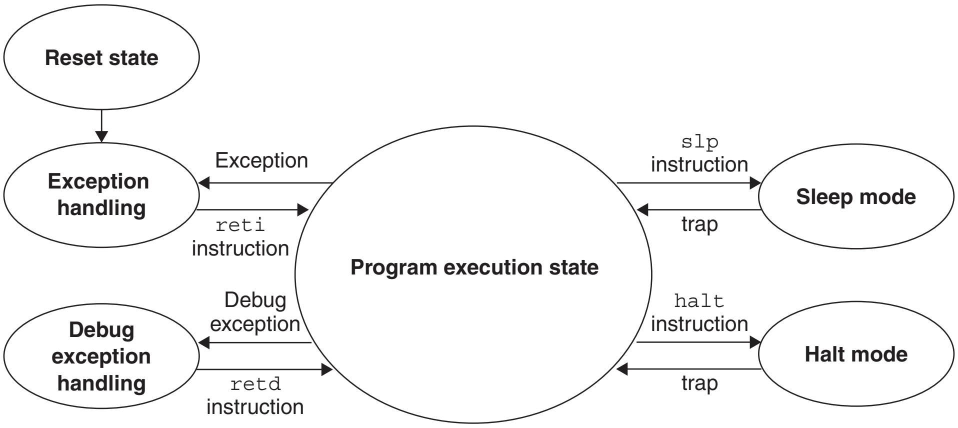


Figure 6.1.1 Processor Status Transition Diagram

6.1.1 Reset State

The processor is initialized when the reset signal is asserted, and then starts processing from the reset vector when the reset signal is deasserted.

6.1.2 Program Execution State

This is a state in which the processor executes the user program sequentially. The processor state transits to another when an exception occurs or the `slp` or `halt` instruction is executed.

6.1.3 Exception Handling

When a software or other exception occurs, the processor enters an exception handling state. The following are the possible causes of the need for exception handling:

(1) External interrupt
(2) Software exception
(3) Address misaligned exception
(4) Zero division
(5) NMI
(6) Undefined instruction exception/`ext` exception

6.1.4 Debug Exception

The C33 PE Core incorporates a debugging assistance facility to increase the efficiency of software development. To use this facility, a dedicated mode known as “debug mode” is provided. The processor can be switched from user mode to this mode by the `brk` instruction or a debug exception. The processor does not normally enter this mode.

6.1.5 HALT and SLEEP Modes

The processor is placed in HALT or SLEEP mode to reduce power consumption by executing the `halt` or `slp` instruction in the software (see Section 6.4). Normally the processor can be taken out of HALT or SLEEP mode by NMI or an external interrupt as well as initial reset.

6.2 Program Execution

Following initial reset, the processor loads the reset vector address into the PC and starts executing instructions beginning with the address that was stored in the reset vector. As the instructions in the C33 PE Core are fixed to 16 bits in length, the PC is incremented by 2 each time an instruction is fetched from the address indicated by the PC. In this way, instructions are executed successively.

When a branch instruction is executed, the processor checks the PSR flags and whether the branch conditions have been satisfied, and loads the jump address into the PC.

When an interrupt or exception occurs, the processor loads the address for the interrupt or exception handler routine from the vector table into the PC.
The vector table is a table of vectors that begin with the reset vector. Following initial reset, the vector table is located at the address "0xC00000." The exception vector table address can be determined by referencing the special register TTBR. Alternatively, any desired address can be set for the exception vector table address in the software. In this case, the addresses set in the TTBR must be aligned with the 1K-byte boundary (TTBR[9:0] = fixed to 00 0000 0000).

6.2.1 Instruction Fetch and Execution

Internally in the C33 PE Core, instructions are processed in two pipelined stages, so that data transfer between registers and general arithmetic/logic instructions can be executed in one clock cycle.
Pipelining speeds up instruction processing by executing one instruction while fetching another. In the 2-stage pipeline, each instruction is processed in two stages, with processing of instructions occurring in parallel, for faster instruction execution.

Basic instruction stages

Instruction fetch / Instruction decode	Instruction execution / Memory access / Register write

Hereinafter, each stage is represented by the following symbols:
F (for Fetch): Instruction fetch, instruction decode
E (for Execute): Instruction execution, memory access, register write

Pipelined operation

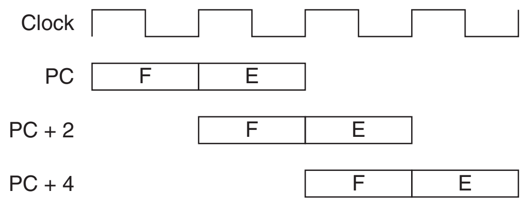


Figure 6.2.1.1 Pipelined Operation

Note: The pipelined operation shown above uses the internal memory. If external memory or low-speed external devices are used, one or more wait cycles may be inserted depending on the devices used, with the E stage kept waiting.

6.2.2 Execution Cycles and Flags

The instructions in the C33 PE Core are processed in parallel at two pipelined stages as described above, so most instructions are executed in one clock cycle. This comprises the basic execution cycle in the C33 PE Core.
Although instructions to transfer data between registers as in register direct addressing are executed in one clock cycle, one or more wait cycles are inserted for accesses to external memory and low-speed external peripheral circuits. These include clock cycles spent for the arbitration by the bus control unit, and wait cycles inherent in the external devices connected to the chip. Note, however, that accesses to the internal RAM and caches are completed in one clock cycle.

The number of clock cycles required for accesses to the internal RAM and caches, as well as flag changes that occur pursuant to memory accesses, are given below.

C33 STD Core CPU compatible instructions

Table 6.2.2.1 Number of Instruction Execution Cycles and Flag Status (C33 STD Compatible Instructions)

Classification	Mnemonic		Cycle	Flag				Remark
				C	V	Z	N	
Arithmetic operation	add	%rd,%rs	1	↔	↔	↔	↔	
		%rd,imm6	1	↔	↔	↔	↔	
		%sp,imm10	1	–	–	–	–	
	adc	%rd,%rs	1	↔	↔	↔	↔	
	sub	%rd,%rs	1	↔	↔	↔	↔	
		%rd,imm6	1	↔	↔	↔	↔	
		%sp,imm10	1	–	–	–	–	
	sbc	%rd,%rs	1	↔	↔	↔	↔	
	cmp	%rd,%rs	1	↔	↔	↔	↔	
		%rd,sign6	1	↔	↔	↔	↔	
	mlt.h	%rd,%rs	5	–	–	–	–	
	mltu.h	%rd,%rs	5	–	–	–	–	
	mlt.w	%rd,%rs	7	–	–	–	–	
	mltu.w	%rd,%rs	7	–	–	–	–	
Branch	jrgt jrgt.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrge jrge.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrlt jrlt.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrle jrle.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrugt jrugt.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jruge jruge.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrult jrult.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrule jrule.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jreq jreq.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jrne jrne.d	sign8	2–3 (∗1, ∗3)	–	–	–	–	
	jp	sign8	2–3 (∗3)	–	–	–	–	
	jp.d	%rb	2–3 (∗3)	–	–	–	–	
	call	sign8	3–4 (∗3)	–	–	–	–	
	call.d	%rb	3–4 (∗3)	–	–	–	–	
	ret ret.d		3–4 (∗3)	–	–	–	–	
	reti		5	↔	↔	↔	↔	PSR change
	retd		5	–	–	–	–	
	int	imm2	7	–	–	–	–	IE = 0
	brk		9	–	–	–	–	IE no change

Classification	Mnemonic		Cycle	Flag				Remark
				C	V	Z	N	
Data transfer	ld.b	%rd,%rs	1	–	–	–	–	
		%rd,[%rb]	1–2 (*4)	–	–	–	–	
		%rd,[%rb]+	2	–	–	–	–	
		%rd,[%sp+imm6]	2	–	–	–	–	
		[%rb],%rs	1–2 (*4)	–	–	–	–	
		[%rb]+,%rs	2	–	–	–	–	
		[%sp+imm6],%rs	2	–	–	–	–	
	ld.ub	%rd,%rs	1	–	–	–	–	
		%rd,[%rb]	1–2 (*4)	–	–	–	–	
		%rd,[%rb]+	2	–	–	–	–	
		%rd,[%sp+imm6]	2	–	–	–	–	
	ld.h	%rd,%rs	1	–	–	–	–	
		%rd,[%rb]	1–2 (*4)	–	–	–	–	
		%rd,[%rb]+	2	–	–	–	–	
		%rd,[%sp+imm6]	2	–	–	–	–	
		[%rb],%rs	1–2 (*4)	–	–	–	–	
		[%rb]+,%rs	2	–	–	–	–	
		[%sp+imm6],%rs	2	–	–	–	–	
	ld.uh	%rd,%rs	1	–	–	–	–	
		%rd,[%rb]	1–2 (*4)	–	–	–	–	
		%rd,[%rb]+	2	–	–	–	–	
		%rd,[%sp+imm6]	2	–	–	–	–	
	ld.w	%rd,%rs	1	–	–	–	–	
		%rd,sign6	1	–	–	–	–	
		%rd,[%rb]	1–2 (*4)	–	–	–	–	
		%rd,[%rb]+	2	–	–	–	–	
		%rd,[%sp+imm6]	2	–	–	–	–	
		[%rb],%rs	1–2 (*4)	–	–	–	–	
		[%rb]+,%rs	2	–	–	–	–	
		[%sp+imm6],%rs	2	–	–	–	–	
System control	nop		1	–	–	–	–	
	halt		5	–	–	–	–	
	slp		5	–	–	–	–	
Immediate extension	ext	imm13	0–1 (*2)	–	–	–	–	
Bit manipulation	btst	[%rb],imm3	2–3 (*4)	–	–	↔	–	
	bclr	[%rb],imm3	3–4 (*4)	–	–	–	–	
	bset	[%rb],imm3	3–4 (*4)	–	–	–	–	
	bnot	[%rb],imm3	3–4 (*4)	–	–	–	–	
Other	swap	%rd,%rs	1	–	–	–	–	
	pushn	%rs	N+1	–	–	–	–	
	popn	%rd	N+1	–	–	–	–	

Function-extended instructions

Table 6.2.2.2 Number of Instruction Execution Cycles and Flag Status (Function-Extended Instructions)

Classification	Mnemonic		Cycle	Flag				Remark
				C	V	Z	N	
Logical operation	and	%rd,%rs	1	–	0	↔	↔	
		%rd,sign6	1	–	0	↔	↔	
	or	%rd,%rs	1	–	0	↔	↔	
		%rd,sign6	1	–	0	↔	↔	
	xor	%rd,%rs	1	–	0	↔	↔	
		%rd,sign6	1	–	0	↔	↔	
	not	%rd,%rs	1	–	0	↔	↔	
		%rd,sign6	1	–	0	↔	↔	
Shift and rotate	srl	%rd,%rs	1	–	–	↔	↔	
		%rd,imm5	1	–	–	↔	↔	
	sll	%rd,%rs	1	–	–	↔	↔	
		%rd,imm5	1	–	–	↔	↔	
	sra	%rd,%rs	1	–	–	↔	↔	
		%rd,imm5	1	–	–	↔	↔	
	sla	%rd,%rs	1	–	–	↔	↔	
		%rd,imm5	1	–	–	↔	↔	
	rr	%rd,%rs	1	–	–	↔	↔	
		%rd,imm5	1	–	–	↔	↔	
	rl	%rd,%rs	1	–	–	↔	↔	
		%rd,imm5	1	–	–	↔	↔	
Data transfer	ld.w	%rd,%ss	1	–	–	–	–	
		%sd,%rs	1–3 (*5)	–	–	–	–	

Added instructions

Table 6.2.2.3 Number of Instruction Execution Cycles and Flag Status (Added Instructions)

Classification	Mnemonic		Cycle	Flag				Remark
				C	V	Z	N	
Branch	jpr jpr.d	%rb	2–3 (∗3)	–	–	–	–	
System control	psrset	imm5	3	↔	↔	↔	↔	
	psrclr	imm5	3	↔	↔	↔	↔	
Coprocessor control	ld.c	%rd,imm4	1	–	–	–	–	
	ld.c	imm4,%rs	1	–	–	–	–	
	do.c	imm6	1	–	–	–	–	
	ld.cf		3	↔	↔	↔	↔	
Other	swaph	%rd,%rs	1	–	–	–	–	
	push	%rs	2	–	–	–	–	
	pop	%rd	1	–	–	–	–	
	pushs	%ss	2–3 (∗6)	–	–	–	–	
	pops	%sd	2–3 (∗6)	–	–	–	–	

∗1 Three cycles when the branch conditions are satisfied and the instruction is not a delayed branch instruction
∗2 Zero cycles when lookahead decoding is possible
∗3 When a branch instruction does not involve a delayed branch (not accompanied by the extension ".d"), a 1-instruction equivalent blank time occurs, as no instructions are executed during a branch; therefore, apparently +1 cycle.
∗4 +1 cycle when `ext` is used
∗5 Three cycles when `%psr` is specified
∗6 Two cycles when `%alr` is specified or three cycles when `%ahr` is specified

In the C33 PE Core, no interlock cycle is generated.

6.3 Interrupts and Exceptions

When an external interrupt or exception occurs during program execution, the processor enters an exception handling state. The exception handling state is a process by which the processor branches to the corresponding user's service routine for the interrupt or exception that occurred. The processor returns after branching and starts executing the program from where it left off.

6.3.1 Priority of Exceptions

The following exception handlings are supported by the C33 PE Core:

(1) Reset, internal exceptions of the processor, and external interrupts for which the processor branches to the relevant exception handler routine by referencing the vector table
(2) Debug exceptions such as breaks that are provided to support debugging by the user

The priority of these exceptions is listed in the table below.

Table 6.3.1.1 Vector Address and Priority of Exceptions

Exception	Vector address (Hex)	Priority
Reset	TTBR + 0x00	High
Address misaligned exception	TTBR + 0x18	↑
Undefined instruction exception	TTBR + 0x0C	
`ext` exception	TTBR + 0x08	
Debug exception	0x00060000	
NMI	TTBR + 0x1C	
Software exception	TTBR + 0x30 to TTBR + 0x3C	↓
Maskable external interrupt	TTBR + 0x40 to TTBR + 0x3FC	Low

When two or more exceptions occur simultaneously, they are processed in order of priority beginning with the one that has the highest priority.

When an exception occurs, the processor disables interrupts that would occur thereafter and performs exception handling. To support multiple interrupts (or another interrupt from within an interrupt), set the IE flag in the PSR to 1 in the exception handler routine to enable interrupts during exception handling. Basically, even when multiple interrupts are enabled, interrupts and exceptions whose priorities are below the one set by the IL[3:0] bits in the PSR are not accepted.

The debug exception has its vector located at the specific addresses, and the vector table is not referenced for this exception. Nor is the stack used for the PC, and the PC is saved in a specific area along with R0.
The table below shows the addresses that are referenced when a debug exception occurs.

Table 6.3.1.2 Debug Exception Vector Address and PC/R0 Save Area

Address	Content
0x00060000	Debug exception handler vector
0x00060008	PC save area
0x0006000C	R0 save area

During debug exception handling, neither other exceptions nor multiple debug exceptions are accepted. They are kept pending until the debug exception handling currently underway finishes.

6.3.2 Vector Table

Vector table in the C33 PE Core

The table below lists the exceptions and interrupts for which the vector table is referenced during exception handling. The priorities of these exceptions and interrupts are managed by the interrupt controller (ITC).

Table 6.3.2.1 Vector List

Exception	Vector No.	Synchronous/ asynchronous	Classification	Vector address
Reset	0	Asynchronous	Interrupt	TTBR + 0x00
reserved	1	–	–	–
`ext` exception	2	Synchronous	Exception	TTBR + 0x08
Undefined instruction exception	3	Synchronous	Exception	TTBR + 0x0C
reserved	4–5	–	–	–
Address misaligned exception	6	Synchronous	Exception	TTBR + 0x18
NMI	7	Asynchronous	Interrupt	TTBR + 0x1C
reserved	8–11	–	–	–
Software exception 0	12	Synchronous	Exception	TTBR + 0x30
Software exception 1	13	Synchronous	Exception	TTBR + 0x34
Software exception 2	14	Synchronous	Exception	TTBR + 0x38
Software exception 3	15	Synchronous	Exception	TTBR + 0x3C
Maskable external interrupt 0	16	Asynchronous	Interrupt	TTBR + 0x40
:	:	:	:	:
Maskable external interrupt 239	255	Asynchronous	Interrupt	TTBR + 0x3FC

The sources of exceptions in the C33 PE Core are shown in Table 6.3.2.1.

The Synchronous/Asynchronous column of the table indicates whether the relevant exception is generated synchronously or asynchronously with the program execution. Those that occur synchronously with the program execution are classified as "exceptions," and those that occur asynchronously are classified as "interrupts." In this manual, the internal processing performed by the processor for interrupts and exceptions that occurred is referred to collectively as "exception handling."

The vector address is one that contains a vector (or the jump address) for the user's exception handler routine that is provided for each exception and is executed when the relevant exception occurs. Because an address value is stored, each vector address is located at a word boundary. The memory area in which these vectors are stored is referred to as the "vector table." The "TTBR" in the Vector Address column represents the base (start) address of the vector table.

In the C33 PE Core, the TTBR is provided as a special register, and because this register can be written to in the software, the vector table can be mapped into any desired area in the RAM.

TTBR (Trap Table Base Register)

	31–10	9–0
TTBR	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0
	1K-byte boundary address (R/W)	Fixed (R only)

The initial value of the TTBR, or the value to which the TTBR is initialized when cold reset, is "0x00C00000."

Referenced vector-table addresses

When an exception occurs, the vector table is referenced from the TTBR value and a 10-bit vector code that is assigned to each exception source. As only bits 31–10 in the TTBR are referenced, the vector table must be located in a 1K-byte boundary RAM area.

TTBR[31:10]	+	Vector code (10 bits)

Vector code is generated by the processor.

6.3.3 Exception Handling

When an interrupt or exception occurs, the processor starts exception handling. (This exception handling does not apply for reset and debug exceptions.)

The exception handling performed by the processor is outlined below.

(1) Suspends the instruction currently being executed.
An interrupt or exception is generated synchronously with the rising edge of the system clock at the end of the cycle of the currently executed instruction.

(2) Saves the contents of the PC and PSR to the stack (SP), in that order.

(3) Clears the IE (interrupt enable) bit in the PSR to disable maskable interrupts that would occur thereafter. If the generated exception is a maskable interrupt, the IL (interrupt level) in the PSR is rewritten to that of the generated interrupt.

(4) Reads the vector for the generated exception from the vector table, and sets it in the PC. The processor thereby branches to the user's exception handler routine.

After branching to the user's exception handler routine, when the `reti` instruction is executed at the end of exception handling, the saved data is restored from the stack in order of the PC and PSR, and the processing returns to the suspended instruction.

6.3.4 Reset

The processor is reset by applying a low-level pulse to its #RESET pin. All bits of the PSR are thereby cleared to 0, and the contents of other registers become indeterminate.
The processor starts operating at the rising edge of the #RESET pulse to perform a reset sequence. In this reset sequence, the reset vector is read out from the top of the vector table and set in the PC. The processor thereby branches to the user's initialization routine, in which it starts executing the program. The reset sequence has priority over all other processing.

6.3.5 Address Misaligned Exception

The load instructions that access memory or I/O areas are characteristic in that the data size to be transferred is predetermined for each instruction used, and that the accessed addresses must be aligned with the respective data-size boundaries.

Instruction	Transfer data size	Address
`ld.b`/`ld.ub`	Byte (8 bits)	Byte boundary (applies to all addresses)
`ld.h`/`ld.uh`	Halfword (16 bits)	Halfword boundary (least significant address bit = 0)
`ld.w`	Word (32 bits)	Word boundary (two least significant address bits = 00)

If the specified address in a load instruction does not satisfy this condition, the processor assumes an address misaligned exception and performs exception handling. In this case, the load instruction is not executed. The PC value saved to the stack in exception handling is the address of the load instruction that caused the exception.

In the load instructions that use the SP as the base address, no address misaligned exceptions will occur, as the addresses are aligned properly according to the data size.

Nor does this exception occur in the instructions that involve branching of the program flow (e.g., `call %rb` or `jp %rb`), as the least significant bit of the PC is always fixed to 0. The same applies to the vector for exception handling.

6.3.6 NMI

An NMI is generated when the #NMI input on the processor is asserted low. When an NMI occurs, the processor performs exception handling after it has finished executing the instruction currently underway. The PC value saved to the stack in exception handling is the address of the instruction that was being executed.
During an NMI exception, other new NMI exceptions are disabled and not accepted (multiple NMI exceptions prohibited). To prevent another NMI from being serviced during a current NMI exception, the processor masks NMIs before it starts executing the NMI exception handler routine. NMIs are unmasked by executing the `reti` instruction, so that it is possible that if another exception occurs in an NMI handler routine and `reti` is executed in that routine, NMIs will be unmasked. In such a case, the NMI handler routine may not be executed correctly. Therefore, make sure that no other exceptions will occur during an NMI handler routine.
NMIs are nonmaskable interrupts, but because if an NMI occurs before SP is set after the processor is reset (either cold start or hot start), the program may run out of control, the #NMI input on the processor is therefore masked in the hardware until the SP is set by the `ld.w %sp,%rs` instruction.

6.3.7 Software Exceptions

A software exception is generated by executing the `int imm2` instruction. The PC value saved to the stack in this exception handling is the address of the next instruction. The operand *imm2* in the `int` instruction specifies the vector address for one of four distinct software exceptions. The processor reads the vector for the exception from the address that is equal to TTBR + 48 (vector address for software exception 0) plus 4 × *imm2*, before branching to the handler routine.

6.3.8 Maskable External Interrupts

The C33 PE Core can accept up to 240 types of maskable external interrupts. It is only when the IE (interrupt enable) flag in the PSR is set that the processor accepts a maskable external interrupt. Furthermore, their acceptable interrupt levels are limited by the IL (interrupt level) field in the PSR. The interrupt levels (0–15) in the IL field dictate the interrupt levels that can be accepted by the processor, and only interrupts with priority levels higher than that are accepted.
The IE flag and the IL field can be set in the software. When an exception occurs, the IE flag is cleared to 0 (interrupts disabled) after the PSR is saved to the stack, and the maskable interrupts remain disabled until the IE flag is set in the handler routine or the handler routine is terminated by the `reti` instruction that restores the PSR from the stack. The IL field is set to the priority level of the interrupt that occurred.
Multiple interrupts or the ability to accept another interrupt during exception handling if its priority is higher than that of the currently serviced interrupt can easily be realized by setting the IE flag in the interrupt handler routine.
When the processor is reset, the PSR is initialized to 0 and the maskable interrupts are therefore disabled, and the interrupt level is set to 0 (interrupts with priority levels 1–15 enabled).

The following describes how the maskable interrupts are accepted and processed by the processor.

(1) Suspends the instruction currently being executed.
The interrupt is accepted synchronously with the rising edge of the system clock at the end of the cycle of the currently executed instruction.

(2) Saves the contents of the PC and PSR to the stack (SP), in that order.

(3) Clears the IE flag in the PSR and copy the priority level of the accepted interrupt to the IL field.

(4) Reads the vector for the interrupt from the vector address in the vector table, and sets it in the PC. The processor then branches to the interrupt handler routine.

In the interrupt handler routine, the `reti` instruction should be executed at the end of processing. In the `reti` instruction, the saved data is restored from the stack in order of the PC and PSR, and the processing returns to the suspended instruction.

6.3.9 Undefined Instruction Exception

When an instruction, which does not exist in the C33 PE instruction set, is executed, an undefined instruction exception occurs. The object code is loaded into the 16 low-order bits of the IDIR register and is processed similar to the `nop` instruction. In this case, the PC value that is saved into the stack by the exception processing is the instruction address that follows the undefined instruction executed.
Address TTBR + 12 is used to store the undefined instruction exception vector.

6.3.10 `ext` Exception

If three or more `ext` instructions are described sequentially, an `ext` exception occurs when the third `ext` instruction is detected. In this case, the PC value that is saved into the stack by the exception processing is the first `ext` instruction address.
Address TTBR + 8 is used to store the `ext` exception vector.
When an instruction, which does not support the extension in the `ext` instruction, follows an `ext`, the `ext` instruction will be executed as a `nop` instruction.

6.4 Power-Down Mode

The C33 PE Core supports two power-down modes: HALT and SLEEP modes.

HALT mode

Program execution is halted at the same time that the C33 PE Core executes the `halt` instruction, and the processor enters HALT mode.
HALT mode commonly turns off only the C33 PE Core operation, note, however that modules to be turned off depend on the implementation of the clock control circuit outside the core. Refer to the technical manual of each model for details.

SLEEP mode

Program execution is halted at the same time the C33 PE Core executes the `slp` instruction, and the processor enters SLEEP mode.
SLEEP mode commonly turns off the C33 PE Core and on-chip peripheral circuit operations, thereby it significantly reduces the current consumption in comparison to the HALT mode. However, modules to be turned off depend on the implementation of the clock control circuit outside the core. Refer to the technical manual of each model for details.

Canceling HALT or SLEEP mode

Initial reset is one cause that can be bring the processor out of HALT or SLEEP mode. Other causes depend on the implementation of the clock control circuit outside the C33 PE Core.
Initial reset, maskable external interrupts, NMI, and debug exceptions are commonly used for canceling HALT and SLEEP modes.

The interrupt enable/disable status set in the processor does not affect the cancellation of HALT or SLEEP modes even if an interrupt signal is used as the cancellation. In other words, interrupt signals are able to cancel HALT and SLEEP modes even if the IE flag in PSR or the interrupt enable bits in the interrupt controller (depending on the implementation) are set to disable interrupts.
When the processor is taken out of HALT or SLEEP mode using an interrupt that has been enabled (by the interrupt controller and IE flag), the corresponding interrupt handler routine is executed. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the processor returns to the instruction next to `halt` or `slp`.
When the interrupt has been disabled, the processor restarts the program from the instruction next to `halt` or `slp` after the processor is taken out of HALT or SLEEP mode.

6.5 Debug Circuit

The C33 PE Core has a debug circuit to assist in software development by the user.
The debug circuit provides the following functions:

- Instruction break
  A debug exception is generated before the set instruction address is executed. An instruction break can be set at three addresses.
- Data break
  A debug exception is generated when the set address is accessed for read or write. A data break can be set at only one address.
- Single step
  A debug exception is generated every instruction executed.
- Forcible break
  A debug exception is generated by an external input signal.
- Software break
  A debug exception is generated when the `brk` instruction is executed.
- PC trace
  The status of instruction execution by the processor is traced.

When a debug exception occurs, the processor performs the following processing:

(1) Suspends the instruction currently being executed.
A debug exception is generated at the end of the E stage of the currently executed instruction, and is accepted at the next rise of the system clock.

(2) Saves the contents of the PC and R0, in that order, to the addresses specified below.
PC → 0x00060008
R0 → 0x0006000C

(3) Loads the debug exception vector located at the address 0x00060000 to PC and branches to the debug exception handler routine.

In the exception handler routine, the `retd` instruction should be executed at the end of processing to return to the suspended instruction. When returning from the exception by the `retd` instruction, the processor restores the saved data in order of the R0 and the PC.
Neither hardware interrupts nor NMI interrupts are accepted during a debug exception.

6.6 Coprocessor Interface

The C33 PE Core incorporates a coprocessor interface. This interface has dedicated coprocessor instructions available for use, allowing various data processors such as an FPU or DSP to be connected to the chip, and is configured as a simple interface (consisting of only a 16-bit instruction bus and 32-bit input and output data buses).

Dedicated coprocessor instructions

`ld.c`	***`%rd,imm4`***	Transfer data from the coprocessor
`ld.c`	***`imm4,%rs`***	Transfer data to the coprocessor
`do.c`	***`imm6`***	Execute the coprocessor
`ld.cf`		Transfer C, V, Z, and N flags from the coprocessor

The concrete commands and status of the coprocessor vary with each coprocessor connected to the chip. Please refer to the user's manual for the coprocessor used.

7 Details of Instructions

This section explains all the instructions in alphabetical order.

Symbols in the instruction reference

`%rd`, *rd*	General-purpose registers (R0–R15) or their contents used as the destination
`%rs`, *rs*	General-purpose registers (R0–R15) or their contents used as the source
`%rb`, *rb*	General-purpose registers (R0–R15) or their contents that hold the base address to be accessed in register indirect addressing
`%sd`, *sd*	Special registers or their contents used as the destination
`%ss`, *ss*	Special registers or their contents used as the source
`%sp`, sp	Stack pointer (SP) or its content

The register field (*rd*, *rs*, *sd*, or *ss*) in the code contains a 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, TTBR = 0b1000, IDIR = 0b1010, DBBR = 0b1011, PC = 0b1111

immX	Unsigned immediate *X* bits in length. The *X* contains a number representing the bit length of the immediate.
signX	Signed immediate *X* bits in length. The *X* contains a number representing the bit length of the immediate. Furthermore, the most significant bit is handled as the sign bit.
IL[3:0]	Interrupt level field
IE	Interrupt enable flag
C	Carry flag
V	Overflow flag
Z	Zero flag
N	Negative flag
–	Indicates that the bit is not changed by instruction execution
↔	Indicates that the bit is set (= 1) or reset (= 0) by instruction execution
0	Indicates that the bit is reset (= 0) by instruction execution

adc %rd, %rs

Function

Addition with carry

Standard) *rd* ← *rd* + *rs* + C

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0
1	0	1	1	1	0	0	0	*rs*	*rd*

0xB8__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
adc   %rd,%rs          ; rd ← rd + rs + C
```

The content of the *rs* register and C (carry) flag are added to the *rd* register.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

(1)
```
adc   %r0,%r1          ; r0 = r0 + r1 + C
```

(2) 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

15–12	11–8	7–4	3–0	
0 0 1 0	0 0 1 0	rs	rd	0x22__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

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

The content of the *rs* register is added to the *rd* register.

(2) Extension 1

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

The 13-bit immediate *imm13* is added to the content of the *rs* register after being zero-extended, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(3) Extension 2

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

The 26-bit immediate *imm26* is added to the content of the *rs* register after being zero-extended, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
add   %r0,%r0          ; r0 = r0 + r0
```

(2)
```
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

15			12	11	10	9–4	3–0	
0	1	1	0	0	0	*imm6*	*rd*	0x60__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Immediate data (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

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

The 6-bit immediate *imm6* is added to the *rd* register after being zero-extended.

(2) Extension 1

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

The 19-bit immediate *imm19* is added to the *rd* register after being zero-extended.

(3) Extension 2

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

The 32-bit immediate *imm32* is added to the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
add   %r0,0x3f          ; r0 = r0 + 0x3f
```

(2)
```
ext   0x1fff
ext   0x1fff
add   %r1,0x3f          ; r1 = r1 + 0xffffffff
```

add %sp, *imm10*

Function

Addition

Standard) sp ← sp + *imm10* × 4
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11	10	9 … 0	
1	0	0	0	0	0	*imm10*	0x80__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate data (unsigned)
Dst: Register direct (SP)

CLK

One cycle

Description

(1) Standard

Quadruples the 10-bit immediate *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 may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

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

and %rd, %rs

Function

Logical AND

Standard) *rd* ← *rd* & *rs*

Extension 1) *rd* ← *rs* & *imm13*

Extension 2) *rd* ← *rs* & *imm26*

Code

15–12	11–8	7–4	3–0
0 0 1 1	0 0 1 0	*rs*	*rd*

0x32__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

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

The content of the *rs* register and that of the *rd* register are logically AND'ed, and the result is loaded into the *rd* register.

(2) Extension 1

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

The content of the *rs* register and the zero-extended 13-bit immediate *imm13* are logically AND'ed, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(3) Extension 2

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

The content of the *rs* register and the zero-extended 26-bit immediate *imm26* are logically AND'ed, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
and   %r0,%r0          ; r0 = r0 & r0
```

(2)
```
ext   0x1
ext   0x1fff
and   %r1,%r2          ; r1 = r2 & 0x00003fff
```

and %rd, sign6

Function

Logical AND

Standard) *rd* ← *rd* & *sign6*

Extension 1) *rd* ← *rd* & *sign19*

Extension 2) *rd* ← *rd* & *sign32*

Code

15			12	11	10	9–4	3–0
0	1	1	1	0	0	*sign6*	*rd*

0x70__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Immediate data (signed)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

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

The content of the *rd* register and the sign-extended 6-bit immediate *sign6* are logically AND'ed, and the result is loaded into the *rd* register.

(2) Extension 1

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

The content of the *rd* register and the sign-extended 19-bit immediate *sign19* are logically AND'ed, and the result is loaded into the *rd* register.

(3) Extension 2

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

The content of the *rd* register and the 32-bit immediate *sign32* are logically AND'ed, and the result is loaded into the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
and   %r0,0x3e         ; r0 = r0 & 0xfffffffe
```

(2)
```
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

15	14	13	12	11	10	9	8	7–4	3	2–0
1	0	1	0	1	1	0	0	*rb*	0	*imm3*

0xAC__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate data (unsigned)
Dst: Register indirect %*rb* = %r0 to %r15

CLK

Three cycles (four cycles when `ext` is used)

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 *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 *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 *imm26* to the contents of the *rb* register. It does not change the contents of the *rb* register.

Example

(1)
```
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.
```

(2)
```
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

15–12	11–8	7–4	3	2–0
1 0 1 1	0 1 0 0	*rb*	0	*imm3*

0xB4__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate data (unsigned)

Dst: Register indirect `%rb` = `%r0` to `%r15`

CLK

Three cycles (four cycles when `ext` is used)

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 *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 *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 *imm26* to the contents of the *rb* register. It does not change the contents of the *rb* register.

Example

(1)
```
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.
```

(2)
```
ext    0x1
bnot   [%r0],0x7      ; Reverses Bit 7 of data in the following
                      ; address.
```

brk

Function

Debugging exception

Standard) W[0x60008] ← pc + 2, W[0x6000C] ← r0, pc ← W[0x60000]

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

0x0400

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

–

CLK

Nine 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 debug-handler routine from the debug-vector address (0x0060000) and sets it to the PC. Thus the program branches to the debug-handler routine. Furthermore the processor enters the debug mode.
The `retd` instruction must be used for return from the debug-handler routine.
This instruction is provided for debug firmware. Do not use it in general programs.

Example

```
brk             ; Executes the debug-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			12	11			8	7			4	3	2		0
1	0	1	1	0	0	0	0	*r b*				0	*imm3*		

0xB0__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate data (unsigned)

Dst: Register indirect `%rb` = `%r0` to `%r15`

CLK

Three cycles (four cycles when `ext` is used)

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 *imm3* specifies the bit number to be cleared (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 *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 *imm26* to the contents of the *rb* register. It does not change the contents of the *rb* register.

Example

(1)
```
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.
```

(2)
```
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			12	11			8	7–4	3	2–0
1	0	1	0	1	0	0	0	*rb*	0	*imm3*

0xA8__

Flag

IE	C	V	Z	N
–	–	–	↔	–

Mode

Src: Immediate data (unsigned)
Dst: Register indirect `%rb` = `%r0` to `%r15`

CLK

Two cycles (three cycles when `ext` is used)

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 *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 *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 *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) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	1	1	d	0	0	0	0	*rb*			

0x060_, 0x070_

`call` *%rb* when d bit (bit 8) = 0
`call.d` *%rb* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Register direct *%rb* = `%r0` to `%r15`

CLK

`call` Four cycles
`call.d` Three 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.
```

Caution

When the `call.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7 … 0
0	0	0	1	1	1	0	d	*sign8*

0x1C__, 0x1D__

`call` *sign8* when d bit (bit 8) = 0
`call.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`call` Four cycles
`call.d` Three 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 *sign8* and adds it to the PC for calling the subroutine that starts from the address. The *sign8* specifies a halfword 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 *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    ; = sign32(21:9)
call   sign8    ; = "call sign32", sign8 = sign32(8:1), sign32(0) = 0
```

The `ext` instructions extend the displacement into 32 bits using their two 13-bit immediates (*imm13* × 2). 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.
```

Caution

When the `call.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

cmp %rd, %rs

Function

Comparison

Standard) *rd - rs*

Extension 1) *rs - imm13*

Extension 2) *rs - imm26*

Code

15			12	11			8	7–4	3–0	
0	0	1	0	1	0	1	0	rs	rd	0x2A__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One 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 *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 *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 may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
cmp   %r0,%r1  ; Changes the flags according to the results of
               ; r0 - r1.
```

(2)
```
ext   0x1
ext   0x1fff
cmp   %r1,%r2  ; Changes the flags according to the results of
               ; r2 - 0x3fff.
```

cmp %rd, sign6

Function

Comparison

Standard) *rd - sign6*

Extension 1) *rd - sign19*

Extension 2) *rd - sign32*

Code

15	14	13	12	11	10	9–4	3–0
0	1	1	0	1	0	*sign6*	*rd*

0x68__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Immediate data (signed)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

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

Subtracts the signed 6-bit immediate *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 *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, sign6 = sign32(5:0)
```

Subtracts the signed 32-bit immediate *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 may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
cmp   %r0,0x3f ; Changes the flags according to the results of
               ; r0 - 0x3f.
```

(2)
```
ext   0x1fff
ext   0x1fff
cmp   %r1,0x3f ; Changes the flags according to the results of
               ; r1 - 0xffffffff.
```

do.c *imm6*

Function

Coprocessor execution

Standard) W[CA(*imm6*)]

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7	6	5–0	
1	0	1	1	1	1	1	1	0	0	*imm6*	0xBF0_

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Immediate (unsigned)

CLK

One cycle

Description

The command specified by *imm6* is issued to the coprocessor. *imm6* is output to the dedicated coprocessor address bus.

Example

```
do.c   0x1a              ; coprocessor execute command 1A
```

ext *imm13*

Function

Immediate extension

Standard) Extends the immediate data/operand of the following instruction
Extension 1) Unusable
Extension 2) Unusable

Code

15	14	13	12–0	
1	1	0	*imm13*	0xC0__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Immediate data (unsigned)

CLK

Zero or One cycle (depending on the instruction queue status)

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 imm3` 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 imm13` instructions are described sequentially, an undefined instruction exception (`ext` exception) will occur.
See descriptions of each instruction for the extension contents and the usage.

Exceptions for the `ext` instruction (not including reset and debug break) are masked in the hardware, and exception handling is determined when the target instruction to be extended is executed. In this case, the return address from exception handling is the beginning of the `ext` instruction.

Example

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

Caution

When a load instruction that transfers data between memory and a register follows the `ext` instruction, an address misaligned 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 misaligned exception occurs, the trap handling 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 processor to HALT mode
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0	
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0x0080

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

–

CLK

Five cycles

Description

Sets the processor to HALT mode for power saving.
Program execution is halted at the same time that the C33 PE Core executes the `halt` instruction, and the processor enters HALT mode.
HALT mode commonly turns off only the C33 PE Core operation, note, however that modules to be turned off depend on the implementation of the clock control circuit outside the core.

Initial reset is one cause that can bring the processor out of HALT mode. Other causes depend on the implementation of the clock control circuit outside the C33 PE Core.
Initial reset, maskable external interrupts, NMI, and debug exceptions are commonly used for canceling HALT mode.
The interrupt enable/disable status set in the processor does not affect the cancellation of HALT mode even if an interrupt signal is used as the cancellation. In other words, interrupt signals are able to cancel HALT mode even if the IE flag in PSR or the interrupt enable bits in the interrupt controller (depending on the implementation) are set to disable interrupts.
When the processor is taken out of HALT mode using an interrupt that has been enabled (by the interrupt controller and IE flag), the corresponding interrupt handler routine is executed. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the processor returns to the instruction next to `halt`.
When the interrupt has been disabled, the processor restarts the program from the instruction next to `halt` after the processor is taken out of HALT mode.

Refer to the technical manual of each model for details of HALT mode.

Example

```
halt              ; Sets the processor 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) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3	2	1 0	
0	0	0	0	0	1	0	0	1	0	0	0	0	0	*imm2*	0x048_

Flag

IE	C	V	Z	N
0	–	–	–	–

Mode

Immediate data (unsigned)

CLK

Seven 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 C33 PE supports four types of software exceptions and the software exception number (0 to 3) is specified by the 2-bit immediate *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 set in the TTBR register (default: 0xC00000).
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) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	1	1	d	1	0	0	0	*rb*			

0x068_, 0x078_

`jp %rb` when d bit (bit 8) = 0

`jp.d %rb` when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Register direct `%rb` = `%r0` to `%r15`

CLK

`jp` Three cycles

`jp.d` Two cycles

Description

(1) Standard

`jp %rb`

The content of the *rb* register is loaded to the PC, and the program branches to that address. The LSB of the *rb* register is ignored and is always handled as 0.

(2) Delayed branch (d bit = 1)

`jp.d %rb`

For the `jp.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jp.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jp.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7 … 0
0	0	0	1	1	1	1	d	*sign8*

0x1E__, 0x1F__

`jp` *sign8* when d bit (bit 8) = 0

`jp.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jp` Three cycles

`jp.d` Two cycles

Description

(1) Standard

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

Doubles the signed 8-bit immediate *sign8* and adds it to the PC. The program flow branches to the address. The *sign8* specifies a halfword 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 immediates (*imm13* × 2). 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
```

For the `jp.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jp.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jp.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

jpr *%rb* / jpr.d *%rb*

Function

Unconditional PC relative jump

Standard) pc ← pc + *rb*
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	1	d	1	1	0	0	*rb*			

0x02C_, 0x03C_

`jpr` `%rb` when d bit (bit 8) = 0
`jpr.d` `%rb` when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Register direct `%rb` = `%r0` to `%r15`

CLK

`jpr` Three cycles
`jpr.d` Two cycles

Description

(1) Standard

```
jpr   %rb
```

The content of the *rb* register is added to the PC, and the program branches to that address.

(2) Delayed branch (d bit = 1)

```
jpr.d   %rb
```

For the `jpr.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jpr.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

```
jpr   %r0          ; PC ← PC + R0
```

Caution

When the `jpr.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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

15			12	11			8	7–0
0	0	0	1	1	0	0	d	*sign8*

0x18__, 0x19__

`jreq` *sign8* when d bit (bit 8) = 0
`jreq.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jreq` Two cycles (when not branched), Three cycles (when branched)
`jreq.d` Two cycles

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 *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jreq.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jreq.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jreq.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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

15–12	11–8	7–0
0 0 0 0	1 0 1 d	*sign8*

0x0A__, 0x0B__

`jrge` *sign8* when d bit (bit 8) = 0
`jrge.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrge` Two cycles (when not branched), Three cycles (when branched)
`jrge.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrge.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrge.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jrge.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7 ... 0
0	0	0	0	1	0	0	d	*sign8*

0x08__, 0x09__

`jrgt` *sign8* when d bit (bit 8) = 0
`jrgt.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrgt` Two cycles (when not branched), Three cycles (when branched)
`jrgt.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrgt.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrgt.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jrgt.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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

15–12	11–8	7–0
0 0 0 0	1 1 1 d	*sign8*

0x0E__, 0x0F__

`jrle` *sign8* when d bit (bit 8) = 0
`jrle.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrle` Two cycles (when not branched), Three cycles (when branched)
`jrle.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrle.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrle.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jrle.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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

15–12	11–8	7–0
0 0 0 0	1 1 0 d	*sign8*

0x0C__, 0x0D__

`jrlt` *sign8* when d bit (bit 8) = 0
`jrlt.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrlt` Two cycles (when not branched), Three cycles (when branched)
`jrlt.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrlt.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrlt.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jrlt.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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

15–12				11–8				7–0
0	0	0	1	1	0	1	d	*sign8*

0x1A__, 0x1B__

`jrne` *sign8* when d bit (bit 8) = 0
`jrne.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrne` Two cycles (when not branched), Three cycles (when branched)
`jrne.d` Two cycles

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 *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrne.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrne.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

```
cmp    %r0,%r1
jrne   0x2        ; Skips the next instruction if r0 ≠ r1.
```

Caution

When the `jrne.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7 … 0
0	0	0	1	0	0	1	d	*sign8*

0x12__, 0x13__

`jruge` *sign8* when d bit (bit 8) = 0
`jruge.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jruge` Two cycles (when not branched), Three cycles (when branched)
`jruge.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jruge.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jruge.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

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

Caution

When the `jruge.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7 … 0
0	0	0	1	0	0	0	d	*sign8*

0x10__, 0x11__

`jrugt` *sign8* when d bit (bit 8) = 0
`jrugt.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrugt` Two cycles (when not branched), Three cycles (when branched)
`jrugt.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrugt.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrugt.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

```
cmp    %r0,%r1   ; r0 and r1 contain unsigned data.
jrugt  0x2       ; Skips the next instruction if r0 > r1.
```

Caution

When the `jrugt.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7–0
0	0	0	1	0	1	1	d	*sign8*

0x16__, 0x17__

`jrule` *sign8* when d bit (bit 8) = 0
`jrule.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrule` Two cycles (when not branched), Three cycles (when branched)
`jrule.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrule.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrule.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

```
cmp    %r0,%r1   ; r0 and r1 contain unsigned data.
jrule  0x2       ; Skips the next instruction if r0 ≤ r1.
```

Caution

When the `jrule.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

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			12	11			8	7 … 0
0	0	0	1	0	1	0	d	*sign8*

0x14__, 0x15__

`jrult` *sign8* when d bit (bit 8) = 0
`jrult.d` *sign8* when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Signed PC relative

CLK

`jrult` Two cycles (when not branched), Three cycles (when branched)
`jrult.d` Two cycles

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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 halfword 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 immediates (*imm13* × 2). 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
```

For the `jrult.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program branches. Exceptions are masked in intervals between the `jrult.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

```
cmp    %r0,%r1  ; r0 and r1 contain unsigned data.
jrult  0x2      ; Skips the next instruction if r0 < r1.
```

Caution

When the `jrult.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

ld.b *%rd, %rs*

Function

Signed byte data transfer

Standard) *rd*(7:0) ← *rs*(7:0), *rd*(31:8) ← *rs*(7)

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0
1	0	1	0	0	0	0	1	*rs*	*rd*

0xA1__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The 8 low-order bits of the *rs* register are transferred to the *rd* register after being sign-extended to 32 bits.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

```
ld.b  %r0,%r1   ; r0 ← r1(7:0) sign-extended
```

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			12	11			8	7–4	3–0	
0	0	1	0	0	0	0	0	*rb*	*rd*	0x20__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.b   %rd,[%rb]        ; memory address = rb
```

The byte data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext    imm13
ld.b   %rd,[%rb]        ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the content of the *rb* register with the 13-bit immediate *imm13* added comprises the memory address, the byte data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

(3) Extension 2

```
ext    imm13            ; = imm26(25:13)
ext    imm13            ; = imm26(12:0)
ld.b   %rd,[%rb]        ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the content of the *rb* register with the 26-bit immediate *imm26* added comprises the memory address, the byte data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

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) Unusable
Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0
0	0	1	0	0	0	0	1	*rb*	*rd*

0x21__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with post-increment `%rb` = `%r0` to `%r15`
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

The byte data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 1.

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			12	11	10	9 … 4	3 … 0	
0	1	0	0	0	0	*imm6*	*rd*	0x40__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with displacement

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

(1) Standard

```
ld.b   %rd,[%sp + imm6]       ; memory address = sp + imm6
```

The byte data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The content of the current SP with the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed.

(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 to a 19-bit quantity. As a result, the content of the SP with the 19-bit immediate *imm19* added comprises the memory address, the byte data in which is transferred 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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the SP with the 32-bit immediate *imm32* added comprises the memory address, the byte data in which is transferred to the *rd* register.

Example

```
ext    0x1
ld.b   %r0,[%sp + 0x1]   ; r0 ← [sp + 0x41] sign-extended
```

ld.b [%rb], %rs

Function

Signed 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

15	14	13	12	11	10	9	8	7–4	3–0	
0	0	1	1	0	1	0	0	*rb*	*rs*	0x34__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`
Dst: Register indirect `%rb` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.b   [%rb],%rs      ; memory address = rb
```

The 8 low-order bits of the *rs* register are transferred to the specified memory location. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext    imm13
ld.b   [%rb],%rs      ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the 8 low-order bits of the *rs* register are transferred to the address indicated by the content of the *rb* register with the 13-bit immediate *imm13* added. The content of the *rb* register is not altered.

(3) Extension 2

```
ext    imm13          ; = imm26(25:13)
ext    imm13          ; = imm26(12:0)
ld.b   [%rb],%rs      ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the 8 low-order bits of the *rs* register are transferred to the address indicated by the content of the *rb* register with the 26-bit immediate *imm26* added. The content of the *rb* register is not altered.

ld.b [%*rb*]+, %*rs*

Function

Signed byte data transfer

Standard) B[*rb*] ← *rs*(7:0), *rb* ← *rb* + 1

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0
0	0	1	1	0	1	0	1	*rb*	*rs*

0x35__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register indirect with post-increment `%rb` = `%r0` to `%r15`

CLK

Two cycles

Description

The 8 low-order bits of the *rs* register are transferred to the specified memory location. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 1.

ld.b [%sp + *imm6*], %rs

Function

Signed 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

15			12	11	10	9–4	3–0
0	1	0	1	0	1	*imm6*	*rs*

0x54__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register indirect with displacement

CLK

Two cycle

Description

(1) Standard

```
ld.b   [%sp + imm6],%rs      ; memory address = sp + imm6
```

The 8 low-order bits of the *rs* register are transferred to the specified memory location. The content of the current SP with the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed.

(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 to a 19-bit quantity. As a result, The 8 low-order bits of the *rs* register are transferred to the address indicated by the content of the SP with the 19-bit immediate *imm19* added.

(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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, The 8 low-order bits of the *rs* register are transferred to the address indicated by the content of the SP with the 32-bit immediate *imm32* added.

Example

```
ext    0x1
ld.b   [%sp + 0x1],%r0  ; B[sp + 0x41] ← 8 low-order bits of r0
```

ld.c *%rd, imm4*

Function

Transfer data from the coprocessor

Standard) *rd*(7:0) ← W[CA(*imm4*)]

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	1	0	0	0	1	*imm4*	*rd*	0xB1__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The contents of the coprocessor register specified by *imm4* is transferred to the general-purpose register *rd*. *imm4* is output to the dedicated coprocessor address bus.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit.

Example

```
ld.c   %r1,0x3    ; r1 ← coprocessor reg3
```

ld.c *imm4*, *%rs*

Function

Transfer data to the coprocessor

Standard) W[CA(*imm4*)] ← *rs*(7:0)

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7 … 4	3 … 0
1	0	1	1	0	1	0	1	*imm4*	*rs*

0xB5__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Immediate (unsigned)

CLK

One cycle

Description

(1) Standard

The contents of the general-purpose register *rs* is transferred to the coprocessor register specified by *imm4*. *imm4* is output to the dedicated coprocessor address bus.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit.

Example

```
ld.c  0x5,%r2   ; coprocessor reg5 ← r2
```

ld.cf

Function
Transfer C, V, Z, and N flags from the coprocessor

Standard) PSR(3:0) ← coprocessor flag
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0

0x01D0

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode –

CLK Three cycles

Description The C, V, Z, and N flags are transferred from the coprocessor to the PSR(3:0).

Example

```
ld.cf            ; copy coprocessor flag
```

ld.h *%rd, %rs*

Function

Signed halfword data transfer

Standard) *rd*(15:0) ← *rs*(15:0), *rd*(31:16) ← *rs*(15)
Extension 1) Unusable
Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0
1	0	1	0	1	0	0	1	*rs*	*rd*

0xA9__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The 16 low-order bits of the *rs* register are transferred to the *rd* register after being sign-extended to 32 bits.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

```
ld.h  %r0,%r1    ; r0 ← r1(15:0) sign-extended
```

ld.h *%rd*, [*%rb*]

Function

Signed halfword 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			12	11			8	7–4	3–0	
0	0	1	0	1	0	0	0	*rb*	*rd*	0x28__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.h   %rd,[%rb]      ; memory address = rb
```

The halfword data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext    imm13
ld.h   %rd,[%rb]      ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the content of the *rb* register with the 13-bit immediate *imm13* added comprises the memory address, the halfword data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

(3) Extension 2

```
ext    imm13          ; = imm26(25:13)
ext    imm13          ; = imm26(12:0)
ld.h   %rd,[%rb]      ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the content of the *rb* register with the 26-bit immediate *imm26* added comprises the memory address, the halfword data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

Caution

The *rb* register and the displacement must specify a halfword boundary address (least significant bit = 0). Specifying an odd address causes an address misaligned exception.

ld.h *%rd*, [*%rb*]+

Function

Signed halfword data transfer

Standard) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15), *rb* ← *rb* + 2
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	1	0	1	0	0	1		*rb*				*rd*		

0x29__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with post-increment `%rb` = `%r0` to `%r15`
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

The halfword data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 2.

Caution

(1) The *rb* register must specify a halfword boundary address (least significant bit = 0). Specifying an odd address causes an address misaligned exception.

(2) 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 halfword data transfer

Standard) $rd(15{:}0) \leftarrow H[sp + imm6 \times 2]$, $rd(31{:}16) \leftarrow H[sp + imm6 \times 2](15)$

Extension 1) $rd(15{:}0) \leftarrow H[sp + imm19]$, $rd(31{:}16) \leftarrow H[sp + imm19](15)$

Extension 2) $rd(15{:}0) \leftarrow H[sp + imm32]$, $rd(31{:}16) \leftarrow H[sp + imm32](15)$

Code

15	14	13	12	11	10	9–4	3–0
0	1	0	0	1	0	*imm6*	*rd*

0x48__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with displacement
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

(1) Standard

```
ld.h  %rd,[%sp + imm6]      ; memory address = sp + imm6 × 2
```

The halfword data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The content of the current SP with twice the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed. The least significant bit of the displacement is always 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 to a 19-bit quantity. As a result, the content of the SP with the 19-bit immediate *imm19* added comprises the memory address, the halfword data in which is transferred to the *rd* register. Make sure the *imm6* specified here resides on a halfword boundary (least significant bit = 0).

(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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the SP with the 32-bit immediate *imm32* added comprises the memory address, the halfword data in which is transferred to the *rd* register. Make sure the *imm6* specified here resides on a halfword boundary (least significant bit = 0).

Example

```
ext   0x1
ld.h  %r0,[%sp + 0x2]  ; r0 ← [sp + 0x42] sign-extended
```

ld.h [%rb], %rs

Function

Signed halfword 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			12	11			8	7			4	3			0
0	0	1	1	1	0	0	0	rb				rs			

0x38__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register indirect `%rb` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.h   [%rb],%rs       ; memory address = rb
```

The 16 low-order bits of the *rs* register are transferred to the specified memory location. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext    imm13
ld.h   [%rb],%rs       ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the 16 low-order bits of the *rs* register are transferred to the address indicated by the content of the *rb* register with the 13-bit immediate *imm13* added. The content of the *rb* register is not altered.

(3) Extension 2

```
ext    imm13           ; = imm26(25:13)
ext    imm13           ; = imm26(12:0)
ld.h   [%rb],%rs       ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the 16 low-order bits of the *rs* register are transferred to the address indicated by the content of the *rb* register with the 26-bit immediate *imm26* added. The content of the *rb* register is not altered.

Caution

The *rb* register and the displacement must specify a halfword boundary address (least significant bit = 0). Specifying an odd address causes an address misaligned exception.

ld.h [%*rb*]+, %*rs*

Function

Signed halfword data transfer

Standard) H[*rb*] ← *rs*(15:0), *rb* ← *rb* + 2

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0
0	0	1	1	1	0	0	1	*rb*	*rs*

0x39__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register indirect with post-increment `%rb` = `%r0` to `%r15`

CLK

Two cycles

Description

The 16 low-order bits of the *rs* register are transferred to the specified memory location. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 2.

Caution

The *rb* register and the displacement must specify a halfword boundary address (least significant bit = 0). Specifying an odd address causes an address misaligned exception.

ld.h [%sp + *imm6*], %*rs*

Function

Signed halfword 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	14	13	12	11	10	9–4	3–0
0	1	0	1	1	0	*imm6*	*rs*

0x58__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct %*rs* = %r0 to %r15

Dst: Register indirect with displacement

CLK

Two cycles

Description

(1) Standard

```
ld.h  [%sp + imm6],%rs     ; memory address = sp + imm6 × 2
```

The 16 low-order bits of the *rs* register are transferred to the specified memory location. The content of the current SP with twice the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed. The least significant bit of the displacement is always 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 to a 19-bit quantity. As a result, the 16 low-order bits of the *rs* register are transferred to the address indicated by the content of the SP with the 19-bit immediate *imm19* added. Make sure the *imm6* specified here resides on a halfword boundary (least significant bit = 0).

(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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the 16 low-order bits of the *rs* register are transferred to the address indicated by the content of the SP with the 32-bit immediate *imm32* added. Make sure the *imm6* specified here resides on a halfword boundary (least significant bit = 0).

Example

```
ext   0x1
ld.h  [%sp + 0x2],%r0  ; H[sp + 0x42] ← 16 low-order bits of r0
```

ld.ub *%rd, %rs*

Function

Unsigned byte data transfer

Standard) *rd*(7:0) ← *rs*(7:0), *rd*(31:8) ← 0

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0	
1 0 1 0	0 1 0 1	*rs*	*rd*	0xA5__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The 8 low-order bits of the *rs* register are transferred to the *rd* register after being zero-extended to 32 bits.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

```
ld.ub   %r0,%r1   ; r0 ← r1(7:0) zero-extended
```

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			12	11			8	7–4	3–0
0	0	1	0	0	1	0	0	*rb*	*rd*

0x24__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.ub   %rd,[%rb]     ; memory address = rb
```

The byte data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext     imm13
ld.ub   %rd,[%rb]     ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the content of the *rb* register with the 13-bit immediate *imm13* added comprises the memory address, the byte data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

(3) Extension 2

```
ext     imm13         ; = imm26(25:13)
ext     imm13         ; = imm26(12:0)
ld.ub   %rd,[%rb]     ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the content of the *rb* register with the 26-bit immediate *imm26* added comprises the memory address, the byte data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

ld.ub *%rd*, [*%rb*]+

Function

Unsigned byte data transfer

Standard) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0, *rb* ← *rb* + 1

Extension 1) Unusable

Extension 2) Unusable

Code

15–12				11–8				7–4	3–0	
0	0	1	0	0	1	0	1	*rb*	*rd*	0x25__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with post-increment `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

The byte data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 1.

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			12	11	10	9–4	3–0	
0	1	0	0	0	1	*imm6*	*rd*	0x44__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with displacement
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

(1) Standard

```
ld.ub   %rd,[%sp + imm6]     ; memory address = sp + imm6
```

The byte data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The content of the current SP with the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed.

(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 to a 19-bit quantity. As a result, the content of the SP with the 19-bit immediate *imm19* added comprises the memory address, the byte data in which is transferred 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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the SP with the 32-bit immediate *imm32* added comprises the memory address, the byte data in which is transferred to the *rd* register.

Example

```
ext    0x1
ld.ub %r0,[%sp + 0x1]  ; r0 ← [sp + 0x41] zero-extended
```

ld.uh *%rd, %rs*

Function

Unsigned halfword data transfer

Standard) *rd*(15:0) ← *rs*(15:0), *rd*(31:16) ← 0

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	1	0	1	1	0	1	*rs*	*rd*	0xAD__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The 16 low-order bits of the *rs* register are transferred to the *rd* register after being zero-extended to 32 bits.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

```
ld.uh   %r0,%r1   ; r0 ← r1(15:0) zero-extended
```

ld.uh %rd, [%rb]

Function

Unsigned halfword 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			12	11			8	7–4	3–0	
0	0	1	0	1	1	0	0	*rb*	*rd*	0x2C__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.uh    %rd,[%rb]      ; memory address = rb
```

The halfword data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext      imm13
ld.uh    %rd,[%rb]      ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the content of the *rb* register with the 13-bit immediate *imm13* added comprises the memory address, the halfword data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

(3) Extension 2

```
ext      imm13          ; = imm26(25:13)
ext      imm13          ; = imm26(12:0)
ld.uh    %rd,[%rb]      ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the content of the *rb* register with the 26-bit immediate *imm26* added comprises the memory address, the halfword data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

Caution

The *rb* register and the displacement must specify a halfword boundary address (least significant bit = 0). Specifying an odd address causes an address misaligned exception.

ld.uh *%rd*, [*%rb*]+

Function

Unsigned halfword data transfer

Standard) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0, *rb* ← *rb* + 2

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0	
0	0	1	0	1	1	0	1	*rb*	*rd*	0x2D__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with post-increment `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

The halfword data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 2.

Caution

(1) The *rb* register must specify a halfword boundary address (least significant bit = 0). Specifying an odd address causes an address misaligned exception.

(2) 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 halfword data transfer

Standard) $rd(15:0) \leftarrow$ H[sp + $imm6 \times 2$], $rd(31:16) \leftarrow 0$

Extension 1) $rd(15:0) \leftarrow$ H[sp + $imm19$], $rd(31:16) \leftarrow 0$

Extension 2) $rd(15:0) \leftarrow$ H[sp + $imm32$], $rd(31:16) \leftarrow 0$

Code

15	14	13	12	11	10	9–4	3–0	
0	1	0	0	1	1	*imm6*	*rd*	0x4C__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with displacement

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

(1) Standard

```
ld.uh   %rd,[%sp + imm6]     ; memory address = sp + imm6 × 2
```

The halfword data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The content of the current SP with twice the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed. The least significant bit of the displacement is always 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 to a 19-bit quantity. As a result, the content of the SP with the 19-bit immediate *imm19* added comprises the memory address, the halfword data in which is transferred to the *rd* register. Make sure the *imm6* specified here resides on a halfword boundary (least significant bit = 0).

(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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the SP with the 32-bit immediate *imm32* added comprises the memory address, the halfword data in which is transferred to the *rd* register. Make sure the *imm6* specified here resides on a halfword boundary (least significant bit = 0).

Example

```
ext     0x1
ld.uh   %r0,[%sp + 0x2] ; r0 ← [sp + 0x42] zero-extended
```

ld.w *%rd*, *%rs*

Function

Word data transfer

Standard) *rd* ← *rs*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0	
0	0	1	0	1	1	1	0	*rs*	*rd*	0x2E__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The content of the *rs* register (word data) is transferred to the *rd* register.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit.

Example

```
ld.w  %r0,%r1    ; r0 ← r1
```

ld.w *%rd, %ss*

Function

Word data transfer

Standard) *rd* ← *ss*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7	4	3	0
1	0	1	0	0	1	0	0	*ss*		*rd*	

0xA4__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%ss` = `%psr`, `%sp`, `%alr`, `%ahr`, `%ttbr`, `%idir`, `%dbbr`, `%pc`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

The content of a special register (word data) is transferred to the *rd* register.

Example

```
ld.w  %r0,%psr   ; r0 ← psr
```

Caution

(1) When a `ld.w %rd,%pc` instruction is executed, a value equal to the PC of this `ld.w` instruction plus 2 is loaded into the register. This instruction must be executed as a delayed slot instruction. If it does not follow a delayed branch instruction, the PC value that is loaded into the *rd* register may not be the next instruction address to the `ld.w` instruction.

(2) When a special register other than the source registers listed above is specified as `%ss`, the `ld.w` instruction will be executed as a `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			12	11			8	7–4	3–0	
0	0	1	1	0	0	0	0	*rb*	*rd*	0x30__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.w   %rd,[%rb]        ; memory address = rb
```

The word data in the specified memory location is transferred to the *rd* register. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext    imm13
ld.w   %rd,[%rb]        ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the content of the *rb* register with the 13-bit immediate *imm13* added comprises the memory address, the word data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

(3) Extension 2

```
ext    imm13            ; = imm26(25:13)
ext    imm13            ; = imm26(12:0)
ld.w   %rd,[%rb]        ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the content of the *rb* register with the 26-bit immediate *imm26* added comprises the memory address, the word data in which is transferred to the *rd* register. The content of the *rb* register is not altered.

Caution

The *rb* register and the displacement must specify a word boundary address (two least significant bits = 0). Specifying other addresses causes an address misaligned exception.

ld.w *%rd*, [*%rb*]+

Function

Word data transfer

Standard) *rd* ← W[*rb*], *rb* ← *rb* + 4

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0
0	0	1	1	0	0	0	1	*rb*	*rd*

0x31__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with post-increment `%rb` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

The word data in the specified memory location is transferred to the *rd* register. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 4.

Caution

(1) The *rb* register and the displacement must specify a word boundary address (two least significant bits = 0). Specifying other addresses causes an address misaligned exception.

(2) 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

15–12	11–10	9–4	3–0	
0 1 0 1	0 0	*imm6*	*rd*	0x50__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register indirect with displacement

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Two cycles

Description

(1) Standard

```
ld.w   %rd,[%sp + imm6]     ; memory address = sp + imm6 × 4
```

The word data in the specified memory location is transferred to the *rd* register. The content of the current SP with 4 times the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed. The two least significant bits of the displacement are always 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 to a 19-bit quantity. As a result, the content of the SP with the 19-bit immediate *imm19* added comprises the memory address, the word data in which is transferred to the *rd* register. Make sure the *imm6* specified here resides on a word boundary (two least significant bits = 0).

(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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the SP with the 32-bit immediate *imm32* added comprises the memory address, the word data in which is transferred to the *rd* register. Make sure the *imm6* specified here resides on a word boundary (two least significant bits = 0).

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	14	13	12	11	10	9–4	3–0	
0	1	1	0	1	1	*sign6*	*rd*	0x6C__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate data (signed)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
ld.w   %rd,sign6       ; rd ← sign6 (sign-extended)
```

The 6-bit immediate *sign6* is loaded to the *rd* register after being sign-extended.

(2) Extension 1

```
ext    imm13           ; = sign19(18:6)
ld.w   %rd,sign6       ; rd ← sign19 (sign-extended),
                       ; sign6 = sign19(5:0)
```

The immediate data is extended into a 19-bit quantity by the `ext` instruction and it is loaded to the *rd* register after being sign-extended.

(3) Extension 2

```
ext    imm13           ; = sign32(31:19)
ext    imm13           ; = sign32(18:6)
ld.w   %rd,sign6       ; rd ← sign32, sign6 = sign32(5:0)
```

The immediate data is extended into a 32-bit quantity by the `ext` instruction and it is loaded to the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

```
ld.w  %r0,0x3f   ; r0 ← 0xffffffff
```

ld.w *%sd, %rs*

Function

Word data transfer

Standard) *sd* ← *rs*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	0	0	0	0	0	*rs*	*sd*	0xA0__

Flag

IE	C	V	Z	N
–	–	–	–	–

If *sd* is the PSR, the content of *rs* is copied.

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%sd` = `%psr`, `%sp`, `%alr`, `%ahr`, `%ttbr`, `%pc`

CLK

One cycle (three cycles when `%sd` = `%psr`)

Description

The content of the *rs* register (word data) is transferred to a special register.

Example

```
ld.w   %sp,%r0    ; sp ← r0
```

Caution

When a special register other than the destination registers listed above is specified as `%sd`, the `ld.w` instruction will be executed as a `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

15–12	11–8	7–4	3–0	
0 0 1 1	1 1 0 0	*rb*	*rs*	0x3C__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register indirect `%rb` = `%r0` to `%r15`

CLK

One cycle (two cycles when `ext` is used)

Description

(1) Standard

```
ld.w   [%rb],%rs      ; memory address = rb
```

The content of the *rs* register (word data) is transferred to the specified memory location. The *rb* register contains the memory address to be accessed.

(2) Extension 1

```
ext    imm13
ld.w   [%rb],%rs      ; memory address = rb + imm13
```

The `ext` instruction changes the addressing mode to register indirect addressing with displacement. As a result, the content of the *rs* register is transferred to the address indicated by the content of the *rb* register with the 13-bit immediate *imm13* added. The content of the *rb* register is not altered.

(3) Extension 2

```
ext    imm13          ; = imm26(25:13)
ext    imm13          ; = imm26(12:0)
ld.w   [%rb],%rs      ; memory address = rb + imm26
```

The addressing mode changes to register indirect addressing with displacement, so the content of the *rs* register is transferred to the address indicated by the content of the *rb* register with the 26-bit immediate *imm26* added. The content of the *rb* register is not altered.

Caution

The *rb* register and the displacement must specify a word boundary address (two least significant bits = 0). Specifying an odd address causes an address misaligned exception.

ld.w [%rb]+, %rs

Function

Word data transfer

Standard) W[*rb*] ← *rs*, *rb* ← *rb* + 4

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0	
0 0 1 1	1 1 0 1	*rb*	*rs*	0x3D__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register indirect with post-increment `%rb` = `%r0` to `%r15`

CLK

Two cycles

Description

The content of the *rs* register (word data) is transferred to the specified memory location. The *rb* register contains the memory address to be accessed. Following data transfer, the address in the *rb* register is incremented by 4.

Caution

The *rb* register and the displacement must specify a word boundary address (two least significant bits = 0). Specifying an odd address causes an address misaligned exception.

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	14	13	12	11	10	9–4	3–0
0	1	0	1	1	1	*imm6*	*rs*

0x5C__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct %*rs* = %r0 to %r15
Dst: Register indirect with displacement

CLK

Two cycle

Description

(1) Standard

```
ld.w  [%sp + imm6],%rs     ; memory address = sp + imm6 × 4
```

The content of the *rs* register is transferred to the specified memory location. The content of the current SP with four times the 6-bit immediate *imm6* added as displacement comprises the memory address to be accessed. The two least significant bits of the displacement are always 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 to a 19-bit quantity. As a result, the content of the *rs* register is transferred to the address indicated by the content of the SP with the 19-bit immediate *imm19* added. Make sure the *imm6* specified here resides on a word boundary (two least significant bits = 0).

(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 two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the *rs* register is transferred to the address indicated by the content of the SP with the 32-bit immediate *imm32* added. Make sure the *imm6* specified here resides on a word boundary (two least significant bits = 0).

mlt.h *%rd, %rs*

Function

Signed 16-bit × 16-bit multiplication

Standard) alr ← *rd*(15:0) × *rs*(15:0)

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0
1	0	1	0	0	0	1	0	*rs*	*rd*

0xA2__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Five cycles

Description

The 16 low-order bits of the *rd* register and the 16 low-order bits of the *rs* register are multiplied together with the signs, and the 32-bit product resulting from the operation is loaded into the ALR register.

Example

```
mlt.h  %r0,%r1          ; alr ← r0(15:0) × r1(15:0)
                        ; signed multiplication
```

mlt.w *%rd, %rs*

Function

Signed 32-bit × 32-bit multiplication

Standard) {ahr, alr} ← *rd* × *rs*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0
1	0	1	0	1	0	1	0	*rs*	*rd*

0xAA__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Seven cycles

Description

The content of the *rd* register and the content of the *rs* register are multiplied together with the signs, and the 64-bit product resulting from the operation is loaded into the AHR and ALR register pair.

Example

```
mlt.w  %r0,%r1          ; {ahr,alr} ← r0 × r1 signed multiplication
```

mltu.h *%rd, %rs*

Function

Unsigned 16-bit × 16-bit multiplication

Standard)	alr ← *rd*(15:0) × *rs*(15:0)
Extension 1)	Unusable
Extension 2)	Unusable

Code

15–12				11–8				7–4	3–0
1	0	1	0	0	1	1	0	*rs*	*rd*

0xA6__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Five cycles

Description

The 16 low-order bits of the *rd* register and the 16 low-order bits of the *rs* register are multiplied together without signs, and the 32-bit product resulting from the operation is loaded into the ALR register.

Example

```
mltu.h  %r0,%r1        ; alr ← r0(15:0) × r1(15:0)
                       ; unsigned multiplication
```

mltu.w *%rd, %rs*

Function

Unsigned 32-bit × 32-bit multiplication

Standard) {ahr, alr} ← *rd* × *rs*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0
1	0	1	0	1	1	1	0	*rs*	*rd*

0xAE__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

Seven cycles

Description

The content of the *rd* register and the content of the *rs* register are multiplied together without signs, and the 64-bit product resulting from the operation is loaded into the AHR and ALR register pair.

Example

```
mltu.w  %r0,%r1  ; {ahr,alr} ← r0 × r1 unsigned multiplication
```

nop

Function No operation

Standard) No operation
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0x0000

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode –

CLK One 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) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0
0	0	1	1	1	1	1	0	*rs*	*rd*

0x3E__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

All the bits of the *rs* register are reversed, and the result is loaded into the *rd* register.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit.

Example

When r1 = 0x55555555

```
not   %r0,%r1     ; r0 = 0xAAAAAAAA
```

not %rd, sign6

Function

Logical negation

Standard) *rd* ← !*sign6*
Extension 1) *rd* ← !*sign19*
Extension 2) *rd* ← !*sign32*

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	1	1	1	*sign6*	*rd*	0x7C__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Immediate data (signed)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
not    %rd,sign6        ; rd ← !sign6
```

All the bits of the sign-extended 6-bit immediate *sign6* are reversed, and the result is loaded into the *rd* register.

(2) Extension 1

```
ext    imm13            ; = sign19(18:6)
not    %rd,sign6        ; rd ← !sign19, sign6 = sign19(5:0)
```

All the bits of the sign-extended 19-bit immediate *sign19* are reversed, and the result is loaded into 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)
```

All the bits of the sign-extended 32-bit immediate *sign32* are reversed, and the result is loaded into the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
not    %r0,0x1f         ; r0 = 0xffffffe0
```

(2)
```
ext    0x7ff
not    %r1,0x3f         ; r1 = 0xfffe0000
```

or %rd, %rs

Function

Logical OR

Standard) *rd* ← *rd* | *rs*

Extension 1) *rd* ← *rs* | *imm13*

Extension 2) *rd* ← *rs* | *imm26*

Code

15			12	11			8	7–4	3–0
0	0	1	1	0	1	1	0	*rs*	*rd*

0x36__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
or   %rd,%rs         ; rd ← rd | rs
```

The content of the *rs* register and that of the *rd* register are logically OR'ed, and the result is loaded into the *rd* register.

(2) Extension 1

```
ext  imm13
or   %rd,%rs         ; rd ← rs | imm13
```

The content of the *rs* register and the zero-extended 13-bit immediate *imm13* are logically OR'ed, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(3) Extension 2

```
ext  imm13           ; = imm26(25:13)
ext  imm13           ; = imm26(12:0)
or   %rd,%rs         ; rd ← rs | imm26
```

The content of the *rs* register and the zero-extended 26-bit immediate *imm26* are logically OR'ed, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
or   %r0,%r0         ; r0 = r0 | r0
```

(2)
```
ext  0x1
ext  0x1fff
or   %r1,%r2         ; r1 = r2 | 0x00003fff
```

or *%rd, sign6*

Function

Logical OR

Standard) *rd* ← *rd* | *sign6*

Extension 1) *rd* ← *rd* | *sign19*

Extension 2) *rd* ← *rd* | *sign32*

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	1	0	1	*sign6*	*rd*	0x74__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Immediate data (signed)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
or    %rd,sign6        ; rd ← rd | sign6
```

The content of the *rd* register and the sign-extended 6-bit immediate *sign6* are logically OR'ed, and the result is loaded into the *rd* register.

(2) Extension 1

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

The content of the *rd* register and the sign-extended 19-bit immediate *sign19* are logically OR'ed, and the result is loaded into 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)
```

The content of the *rd* register and the sign-extended 32-bit immediate *sign32* are logically OR'ed, and the result is loaded into the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
or    %r0,0x3e         ; r0 = r0 | 0xfffffffe
```

(2)
```
ext   0x7ff
or    %r1,0x3f         ; r1 = r1 | 0x0001ffff
```

pop %rd

Function

Pop

Standard) *rd* ← W[sp], sp ← sp + 4
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3 – 0
0	0	0	0	0	0	0	0	0	1	0	1	*rd*

0x005_

Flag

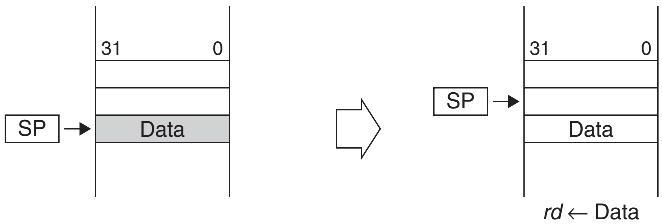
IE	C	V	Z	N
–	–	–	–	–

Mode Register direct `%rd` = `%r0` to `%r15`

CLK One cycle

Description The data of a general-purpose register that has been saved to the stack by a `push` instruction is restored from the stack. The `pop` instruction restores word data from the stack with an address indicated by the current SP to the *rd* register, and increments the SP by an amount equivalent to 1 word (4 bytes).

Stack operation when `pop %rd` is executed



Example

```
pop   %r3          ; r3 ← W[sp], sp ← sp + 4
```

popn *%rd*

Function

Pop

Standard) "*rN* ← W[sp], sp ← sp + 4" repeated for *rN* = r0 to *rd*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	1	0	0	1	0	0	*rd*			

0x024_

Flag

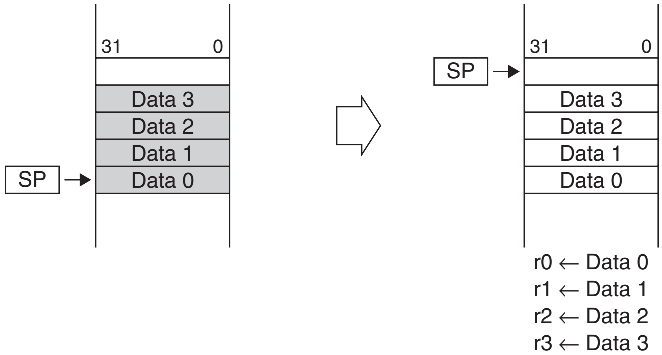
IE	C	V	Z	N
–	–	–	–	–

Mode Register direct `%rd` = `%r0` to `%r15`

CLK N + 1 cycles, where N = number of registers to be restored

Description The data of general-purpose registers that have been saved to the stack by a `pushn` instruction is restored from the stack. The `popn` instruction restores word data from the stack with its address indicated by the current SP to the r0 register, and increments the SP by an amount equivalent to 1 word (4 bytes). This operation is repeated until a register that matches *rd* is reached. The *rd* must be the same register as specified in the corresponding `pushn` instruction.

Stack operation when `popn %rd` (where `%rd` = `%r3`) is executed



Example

```
popn  %r3        ; r0, r1, r2, and r3 are restored
```

pops %sd

Function

Pop

Standard) When *sd* = ahr: alr ← W[sp], sp ← sp + 4, ahr ← W[sp], sp ← sp + 4
When *sd* = alr: alr ← W[sp], sp ← sp + 4

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	0	1	1	0	1	*sd*			

0x00D_

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode Register direct `%sd` = `%alr` or `%ahr`

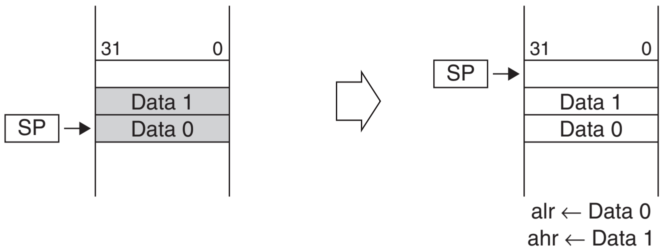
CLK Two cycles (when *sd* = alr), Three cycles (when *sd* = ahr)

Description This instruction restores the data of special registers that have been saved to the stack by a `pushs` instruction back to each register.

(1) When the *sd* register is the ALR register
The word data at the address indicated by the current SP is restored to the ALR register, and the SP is incremented by an amount equivalent to 1 word (4 bytes).

(2) When the *sd* register is the AHR register
The word data at the address indicated by the current SP is restored to the ALR register, and the SP is incremented by an amount equivalent to 1 word (4 bytes). Next, the word data at the address indicated by the current SP is restored to the AHR register, and the SP is incremented by an amount equivalent to 1 word (4 bytes). The *sd* must be the same register as specified in the corresponding `pushs` instruction.

Stack operation when `pops %sd` (where `%sd` = `%ahr`) is executed



Example

```
(1) pops  %alr   ; alr is restored singly
(2) pops  %ahr   ; registers are restored in order of alr and ahr
```

Caution When a register other than ALR or AHR is specified as the *sd* register, the `pops` instruction does not pop data from the stack.

psrclr *imm5*

Function

Clear PSR bit

Standard) psr ← psr & !*imm5*

Extension 1) Unusable

Extension 2) Unusable

Code

15	14	13	12	11	10	9	8	7	6	5	4–0	
1	0	1	1	1	1	1	1	1	0	0	*imm5*	0xBF8_

Flag

IE	C	V	Z	N
↔	↔	↔	↔	↔

Mode

Immediate

CLK

Three cycles

Description

Clear the bit in the PSR specified by the immediate *imm5* to 0. The value of *imm5* indicates a bit number, with values 0, 1, 2, 3, and 4 representing bits 0 (N), 1 (Z), 2 (V), 3 (C), and 4 (IE), respectively. An *imm5* of more than 4 is not effective and does not alter the contents of PSR.

Example

```
psrclr  2        ; V ← 0 (V flag cleared)
```

psrset *imm5*

Function Set PSR bit

Standard) psr ← psr | *imm5*
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7		5	4				0	
1	0	1	1	1	1	1	1	0	1	0	*imm5*					0xBF4_

Flag

IE	C	V	Z	N
↔	↔	↔	↔	↔

Mode Immediate

CLK Three cycles

Description Set the bit in the PSR specified by the immediate *imm5* to 1. The value of *imm5* indicates a bit number, with values 0, 1, 2, 3, and 4 representing bits 0 (N), 1 (Z), 2 (V), 3 (C), and 4 (IE), respectively. An *imm5* of more than 4 is not effective and does not alter the contents of PSR.

Example

```
psrset  2        ; V ← 1 (V flag set)
```

push %*rs*

Function

Push

Standard) sp ← sp - 4, W[sp] ← *rs*

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3			0	
0	0	0	0	0	0	0	0	0	0	0	1	*rs*				0x001_

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Register direct `%rs` = `%r0` to `%r15`

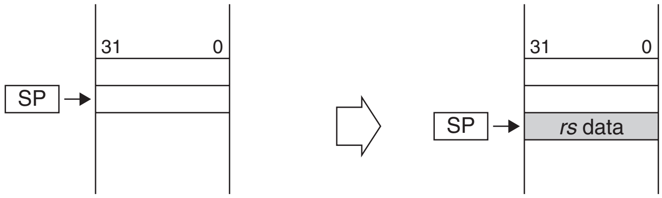
CLK

Two cycles

Description

Save the data of a general-purpose register to the stack.
The `push` instruction first decrements the current SP by an amount equivalent to 1 word (4 bytes), and saves the content of the *rs* register to that address.

Stack operation when `push %rs` is executed



Example

```
push   %r3         ; sp ← sp - 4, W[sp] ← r3
```

pushn %rs

Function

Push

Standard) "sp ← sp - 4, W[sp] ← *rN*" repeated for *rN* = *rs* to r0
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3–0
0	0	0	0	0	0	1	0	0	0	0	0	*rs*

0x020_

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Register direct `%rs` = `%r0` to `%r15`

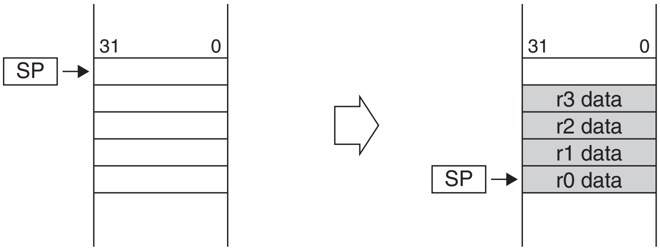
CLK

N + 1 cycles, where N = number of registers to be saved

Description

Save the data of general-purpose registers to the stack.
The `pushn` instruction first decrements the current SP by an amount equivalent to 1 word (4 bytes), and saves the content of the *rs* register to that address. This operation is repeated successively until the r0 register is reached.

Stack operation when `pushn %rs` (where `%rs` = `%r3`) is executed



Example

```
pushn  %r3      ; r3, r2, r1, and r0 are saved
```

pushs *%ss*

Function

Push

Standard) When *ss* = ahr: sp ← sp - 4, W[sp] ← ahr, sp ← sp - 4, W[sp] ← alr
When *ss* = alr: sp ← sp - 4, W[sp] ← alr

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0	
0 0 0 0	0 0 0 0	1 0 0 1	*ss*	0x009_

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode Register direct `%ss` = `%alr` or `%ahr`

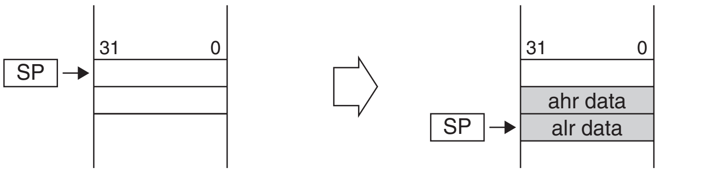
CLK Two cycles (when *ss* = alr), Three cycles (when *ss* = ahr)

Description Save the data of special registers to the stack.

(1) When the *ss* register is the ALR register
The current SP is decremented by an amount equivalent to 1 word (4 bytes), and the content of the ALR register is saved to that address.

(2) When the *ss* register is the AHR register
The current SP is decremented by an amount equivalent to 1 word (4 bytes), and the content of the AHR register is saved to that address. Next, SP is decremented by an amount equivalent to 1 word (4 bytes), and the content of the ALR register is saved to that address.

Stack operation when `pushs %ss` (where `%ss` = `%ahr`) is executed



The ahr and alr registers are saved

Example

(1) `pushs %alr ; alr is saved singly`

(2) `pushs %ahr ; registers are saved in order of ahr and alr`

Caution When a register other than ALR or AHR is specified as the *ss* register, the `pushs` instruction does not save the register data to the stack.

ret / ret.d

Function

Return from subroutine

Standard) pc ← W[sp], sp ← sp + 4
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	1	1	d	0	1	0	0	0	0	0	0

0x0640, 0x0740

`ret` when d bit (bit 8) = 0
`ret.d` when d bit (bit 8) = 1

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

–

CLK

`ret` Four cycles
`ret.d` Three cycles

Description

(1) Standard

`ret`

Restores 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`

For the `ret.d` instruction, the next instruction becomes a delayed instruction. A delayed instruction is executed before the program returns from the subroutine. Exceptions are masked in intervals between the `ret.d` instruction and the next instruction, so no interrupts or exceptions occur.

Example

```
ret.d
add   %r0,%r1    ; Executed before return from the subroutine
```

Caution

When the `ret.d` instruction (delayed branch) is used, be careful to ensure that the next instruction is limited to those that can be used as a delayed instruction. If any other instruction is executed, the program may operate indeterminately. For the usable instructions, refer to the instruction list in the Appendix.

retd

Function

Return from a debug-exception handler routine

Standard) r0 ← W[0x6000C], pc ← W[0x60008]

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3			0	
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0x0440

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

–

CLK

Five cycles

Description

Restore the contents of the R0 and PC that were saved to the debug exception memory space when an debug exception occurred to the respective registers, and return from the debug exception handler routine.

Example

```
retd            ; Return from a debug exception handler routine
```

reti

Function

Return from trap handler routine

Standard) pc ← W[sp + 4], psr ← W[sp], sp ← sp + 8
Extension 1) Unusable
Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0

0x04C0

Flag

IE	C	V	Z	N
↔	↔	↔	↔	↔

Mode

–

CLK

Five cycles

Description

Restore the contents of the PC and PSR that were saved to the stack when an exception or interrupt occurred to the respective registers, and return from the trap handler routine. The SP is incremented by an amount equivalent to 2 words.

Example

```
reti            ; Return from a trap handler routine
```

rl *%rd, %rs*

Function

Rotate to the left

Standard) Rotate the content of *rd* to the left as many bits as specified by *rs* (0 to 31), LSB ← MSB

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0
1 0 0 1	1 1 0 1	rs	rd

0x9D__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

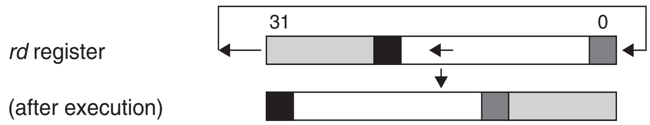
CLK

One cycle

Description

(1) Standard

The *rd* register is rotated as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. The value in the most significant bit of the *rd* register is placed in the least significant bit.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

rl *%rd, imm5*

Function

Rotate to the left

Standard) Rotate the content of *rd* to the left as many bits as specified by *imm5* (0 to 31), LSB ← MSB

Extension 1) Unusable

Extension 2) Unusable

Code

When *imm5*(4) = 0, rotated to the left by 0 to 15 bits

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	0	1	1	1	0	0	*imm5*(3:0)	*r d*	0x9C__

When *imm5*(4) = 1, rotated to the left by 16 to 31 bits

15	14	13	12	11	10	9	8	7–4	3–0	
0	0	1	1	0	1	1	1	*imm5*(3:0)	*r d*	0x37__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Immediate (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

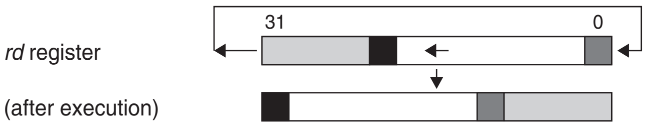
CLK

One cycle

Description

(1) Standard

The *rd* register is rotated as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. The value in the most significant bit of the *rd* register is placed in the least significant bit.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

rr %rd, %rs

Function

Rotate to the right

Standard) Rotate the content of *rd* to the right as many bits as specified by *rs* (0 to 31), MSB ← LSB

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7	4	3	0
1	0	0	1	1	0	0	1	rs		rd	

0x99__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

The *rd* register is rotated as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. The value in the least significant bit of the *rd* register is placed in the most significant bit.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

rr %rd, imm5

Function

Rotate to the right

Standard) Rotate the content of *rd* to the right as many bits as specified by *imm5* (0 to 31), MSB ← LSB

Extension 1) Unusable

Extension 2) Unusable

Code

When *imm5*(4) = 0, rotated to the right by 0 to 15 bits

15–12	11–8	7–4	3–0	
1 0 0 1	1 0 0 0	*imm5*(3:0)	*r d*	0x98__

When *imm5*(4) = 1, rotated to the right by 16 to 31 bits

15–12	11–8	7–4	3–0	
0 0 1 1	0 0 1 1	*imm5*(3:0)	*r d*	0x33__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Immediate (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

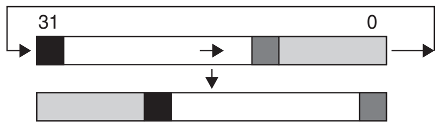
Description

(1) Standard

The *rd* register is rotated as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. The value in the least significant bit of the *rd* register is placed in the most significant bit.

rd register

(after execution)



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit included.

sbc %rd, %rs

Function

Subtraction with borrow

Standard) *rd* ← *rd* - *rs* - C

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0	
1 0 1 1	1 1 0 0	rs	rd	0xBC__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
sbc   %rd,%rs          ; rd ← rd - rs - C
```

The content of the *rs* register and C (carry) flag are subtracted from the *rd* register.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

(1)
```
sbc   %r0,%r1          ; r0 = r0 - r1 - C
```

(2) 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
```

sla %rd, %rs

Function

Arithmetic shift to the left

Standard) Shift the content of *rd* to left as many bits as specified by *rs* (0 to 31), LSB ← 0

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7–4	3–0	
1	0	0	1	0	1	0	1	rs	rd	0x95__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Register direct %rs = %r0 to %r15

Dst: Register direct %rd = %r0 to %r15

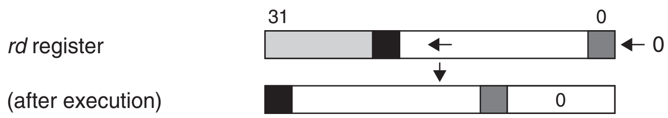
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. Data "0" is placed in the least significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

sla *%rd, imm5*

Function

Arithmetic shift to the left

Standard) Shift the content of *rd* to left as many bits as specified by *imm5* (0 to 31), LSB ← 0
Extension 1) Unusable
Extension 2) Unusable

Code

When *imm5*(4) = 0, arithmetic shift to the left by 0 to 15 bits

15–12	11–8	7–4	3–0	
1 0 0 1	0 1 0 0	*imm5*(3:0)	*r d*	0x94__

When *imm5*(4) = 1, arithmetic shift to the left by 16 to 31 bits

15–12	11–8	7–4	3–0	
0 0 1 0	1 1 1 1	*imm5*(3:0)	*r d*	0x2F__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

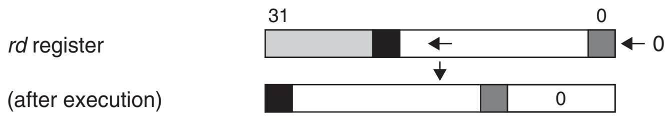
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. Data "0" is placed in the least significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

sll %rd, %rs

Function

Logical shift to the left

Standard) Shift the content of *rd* to left as many bits as specified by *rs* (0 to 31), LSB ← 0

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0
1 0 0 0	1 1 0 1	rs	rd

0x8D__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

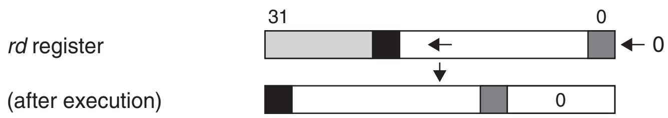
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. Data "0" is placed in the least significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

sll *%rd, imm5*

Function

Logical shift to the left

Standard) Shift the content of *rd* to left as many bits as specified by *imm5* (0 to 31), LSB ← 0

Extension 1) Unusable

Extension 2) Unusable

Code

When *imm5*(4) = 0, logical shift to the left by 0 to 15 bits

15–12	11–8	7–4	3–0	
1 0 0 0	1 1 0 0	*imm5*(3:0)	*rd*	0x8C__

When *imm5*(4) = 1, logical shift to the left by 16 to 31 bits

15–12	11–8	7–4	3–0	
0 0 1 0	0 1 1 1	*imm5*(3:0)	*rd*	0x27__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Immediate (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

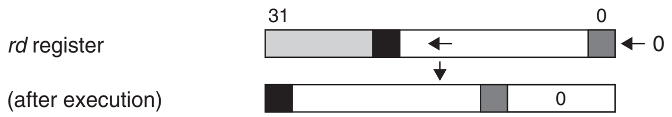
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. Data "0" is placed in the least significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

slp

Function

SLEEP

Standard) Place the processor in SLEEP mode

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

0x0040

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

–

CLK

Five cycles

Description

Places the processor in SLEEP mode for power saving.
Program execution is halted at the same time that the C33 PE Core executes the `slp` instruction, and the processor enters SLEEP mode.
SLEEP mode commonly turns off the C33 PE Core and on-chip peripheral circuit operations, thereby it significantly reduces the current consumption in comparison to the HALT mode.

Initial reset is one cause that can bring the processor out of SLEEP mode. Other causes depend on the implementation of the clock control circuit outside the C33 PE Core.
Initial reset, maskable external interrupts, NMI, and debug exceptions are commonly used for canceling SLEEP mode.
The interrupt enable/disable status set in the processor does not affect the cancellation of SLEEP mode even if an interrupt signal is used as the cancellation. In other words, interrupt signals are able to cancel SLEEP mode even if the IE flag in PSR or the interrupt enable bits in the interrupt controller (depending on the implementation) are set to disable interrupts.
When the processor is taken out of SLEEP mode using an interrupt that has been enabled (by the interrupt controller and IE flag), the corresponding interrupt handler routine is executed. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the processor returns to the instruction next to `slp`.
When the interrupt has been disabled, the processor restarts the program from the instruction next to `slp` after the processor is taken out of SLEEP mode.

Refer to the technical manual of each model for details of SLEEP mode.

Example

```
slp       ; The processor is placed in SLEEP mode.
```

sra %rd, %rs

Function

Arithmetic shift to the right

Standard) Shift the content of *rd* to right as many bits as specified by *rs* (0 to 31), MSB ← MSB
Extension 1) Unusable
Extension 2) Unusable

Code

15–12	11–8	7–4	3–0	
1 0 0 1	0 0 0 1	*rs*	*rd*	0x91__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`
Dst: Register direct `%rd` = `%r0` to `%r15`

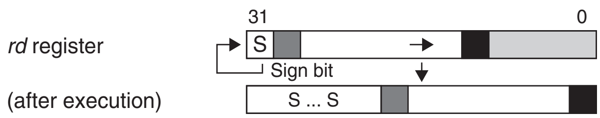
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. The sign bit is copied to the most significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

sra %rd, imm5

Function

Arithmetic shift to the right

Standard) Shift the content of *rd* to right as many bits as specified by *imm5* (0 to 31),
MSB ← MSB

Extension 1) Unusable

Extension 2) Unusable

Code

When *imm5*(4) = 0, arithmetic shift to the right by 0 to 15 bits

15–12	11–8	7–4	3–0	
1 0 0 1	0 0 0 0	*imm5*(3:0)	*r d*	0x90__

When *imm5*(4) = 1, arithmetic shift to the right by 16 to 31 bits

15–12	11–8	7–4	3–0	
0 0 1 0	1 0 1 1	*imm5*(3:0)	*r d*	0x2B__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Immediate (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

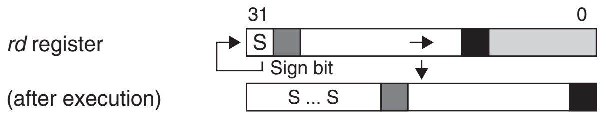
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. The sign bit is copied to the most significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit included.

srl *%rd, %rs*

Function

Logical shift to the right

Standard) Shift the content of *rd* to right as many bits as specified by *rs* (0 to 31), MSB ← 0
Extension 1) Unusable
Extension 2) Unusable

Code

15–12	11–8	7–4	3–0	
1 0 0 0	1 0 0 1	*rs*	*rd*	0x89__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`
Dst: Register direct `%rd` = `%r0` to `%r15`

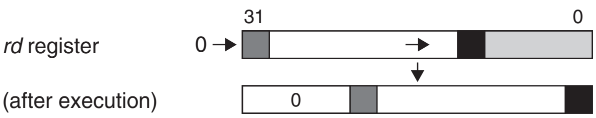
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. Data "0" is placed in the most significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

srl *%rd, imm5*

Function

Logical shift to the right

Standard) Shift the content of *rd* to right as many bits as specified by *imm5* (0 to 31), MSB ← 0
Extension 1) Unusable
Extension 2) Unusable

Code

When *imm5*(4) = 0, logical shift to the right by 0 to 15 bits

15–12	11–8	7–4	3–0	
1 0 0 0	1 0 0 0	*imm5*(3:0)	*r d*	0x88__

When *imm5*(4) = 1, logical shift to the right by 16 to 31 bits

15–12	11–8	7–4	3–0	
0 0 1 0	0 0 1 1	*imm5*(3:0)	*r d*	0x23__

Flag

IE	C	V	Z	N
–	–	–	↔	↔

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

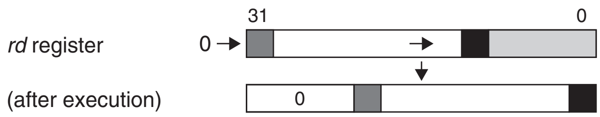
CLK

One cycle

Description

(1) Standard

The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. Data "0" is placed in the most significant bit of the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit included.

sub *%rd, %rs*

Function

Subtraction

Standard) *rd* ← *rd* - *rs*

Extension 1) *rd* ← *rs* - *imm13*

Extension 2) *rd* ← *rs* - *imm26*

Code

15			12	11			8	7–4	3–0
0	0	1	0	0	1	1	0	*rs*	*rd*

0x26__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
sub   %rd,%rs          ; rd ← rd - rs
```

The content of the *rs* register is subtracted from the *rd* register.

(2) Extension 1

```
ext   imm13
sub   %rd,%rs          ; rd ← rs - imm13
```

The 13-bit immediate *imm13* is subtracted from the content of the *rs* register after being zero-extended, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(3) Extension 2

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

The 26-bit immediate *imm26* is subtracted from the content of the *rs* register after being zero-extended, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
sub   %r0,%r0          ; r0 = r0 - r0
```

(2)
```
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			12	11	10	9–4	3–0
0	1	1	0	0	1	*imm6*	*rd*

0x64__

Flag

IE	C	V	Z	N
–	↔	↔	↔	↔

Mode

Src: Immediate data (unsigned)

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
sub   %rd,imm6          ; rd ← rd - imm6
```

The 6-bit immediate *imm6* is subtracted from the *rd* register after being zero-extended.

(2) Extension 1

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

The 19-bit immediate *imm19* is subtracted from the *rd* register after being zero-extended.

(3) Extension 2

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

The 32-bit immediate *imm32* is subtracted from the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
sub   %r0,0x3f          ; r0 = r0 - 0x3f
```

(2)
```
ext   0x1fff
ext   0x1fff
sub   %r1,0x3f          ; r1 = r1 - 0xffffffff
```

sub %sp, *imm10*

Function

Subtraction

Standard) sp ← sp - *imm10* × 4

Extension 1) Unusable

Extension 2) Unusable

Code

15			12	11	10	9–0	
1	0	0	0	0	1	*imm10*	0x84__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Immediate data (unsigned)

Dst: Register direct (SP)

CLK

One cycle

Description

(1) Standard

Quadruples the 10-bit immediate *imm10* and subtracts it from the stack pointer SP. The *imm10* is zero-extended into 32 bits prior to the operation.

(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the “d” bit.

Example

```
sub  %sp,0x100         ; sp = sp - 0x400
```

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) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0
1 0 0 1	0 0 1 0	*rs*	*rd*

0x92__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

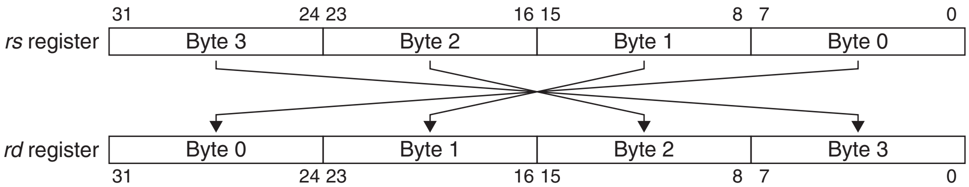
CLK

One cycle

Description

(1) Standard

Swaps the byte order of the *rs* register high and low and loads the results to the *rd* register.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

When r1 = 0x87654321

```
swap  %r0,%r1   ; r0 ← 0x21436587
```

swaph *%rd, %rs*

Function

Swap

Standard) *rd*(31:24) ← *rs*(23:16), *rd*(23:16) ← *rs*(31:24), *rd*(15:8) ← *rs*(7:0), *rd*(7:0) ← *rs*(15:8)

Extension 1) Unusable

Extension 2) Unusable

Code

15–12	11–8	7–4	3–0
1 0 0 1	1 0 1 0	*rs*	*rd*

0x9A__

Flag

IE	C	V	Z	N
–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

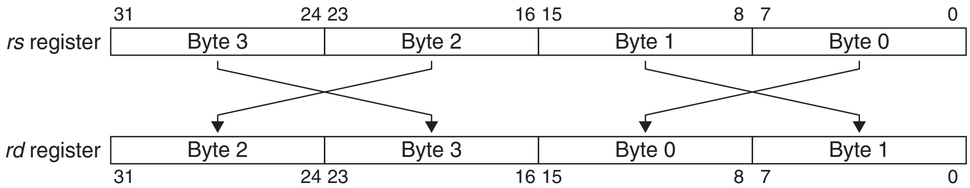
CLK

One cycle

Description

(1) Standard

Converts the 32-bit data in a general-purpose register between big and little endians at halfword boundaries.



(2) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit.

Example

When r1 = 0x12345678

```
swaph  %r2,%r1          ; 0x34127856 → r2
```

xor %rd, %rs

Function

Exclusive OR

Standard) *rd* ← *rd* ^ *rs*

Extension 1) *rd* ← *rs* ^ *imm13*

Extension 2) *rd* ← *rs* ^ *imm26*

Code

15			12	11			8	7–4	3–0	
0	0	1	1	1	0	1	0	rs	rd	0x3A__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Register direct `%rs` = `%r0` to `%r15`

Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
xor    %rd,%rs          ; rd ← rd ^ rs
```

The content of the *rs* register and that of the *rd* register are exclusively OR'ed, and the result is loaded into the *rd* register.

(2) Extension 1

```
ext    imm13
xor    %rd,%rs          ; rd ← rs ^ imm13
```

The content of the *rs* register and the zero-extended 13-bit immediate *imm13* are exclusively OR'ed, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(3) Extension 2

```
ext    imm13            ; = imm26(25:13)
ext    imm13            ; = imm26(12:0)
xor    %rd,%rs          ; rd ← rs ^ imm26
```

The content of the *rs* register and the zero-extended 26-bit immediate *imm26* are exclusively OR'ed, and the result is loaded into the *rd* register. The content of the *rs* register is not altered.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
xor   %r0,%r0          ; r0 = r0 ^ r0
```

(2)
```
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

15			12	11	10	9–4	3–0
0	1	1	1	1	0	*sign6*	*rd*

0x78__

Flag

IE	C	V	Z	N
–	–	0	↔	↔

Mode

Src: Immediate data (signed)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

```
xor   %rd,sign6        ; rd ← rd ^ sign6
```

The content of the *rd* register and the sign-extended 6-bit immediate *sign6* are exclusively OR'ed, and the result is loaded into the *rd* register.

(2) Extension 1

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

The content of the *rd* register and the sign-extended 19-bit immediate *sign19* are exclusively OR'ed, and the result is loaded into 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)
```

The content of the *rd* register and the sign-extended 32-bit immediate *sign32* are exclusively OR'ed, and the result is loaded into the *rd* register.

(4) Delayed instruction

This instruction may be executed as a delayed instruction by writing it directly after a branch instruction with the "d" bit. In this case, extension of the immediate by the `ext` instruction cannot be performed.

Example

(1)
```
xor   %r0,0x3e         ; r0 = r0 ^ 0xfffffffe
```

(2)
```
ext   0x7ff
xor   %r1,0x3f         ; r1 = r1 ^ 0x0001ffff
```

Appendix Instruction Code List (in Order of Codes)

Class 0 (1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S
Class			op1				d	op2		0	0	*imm2,rd,rs,rb*							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	nop	1	×	×
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	slp	5	×	×
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	halt	5	×	×
0	0	0	0	0	0	1	0	0	0	0	0	*rs*				pushn *%rs*	N+1	×	×
0	0	0	0	0	0	1	0	0	1	0	0	*rd*				popn *%rd*	N+1	×	×
0	0	0	0	0	0	1	0	1	1	0	0	*rb*				**jpr *%rb***	3	×	×
0	0	0	0	0	0	1	1	1	1	0	0	*rb*				**jpr.d *%rb***	2	×	×
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	brk	9	×	×
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	retd	5	×	×
0	0	0	0	0	1	0	0	1	0	0	0	0	0	*imm2*		int *imm2*	7	×	×
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	reti	5	×	×
0	0	0	0	0	1	1	0	0	0	0	0	*rb*				call *%rb*	4	×	×
0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	ret	4	×	×
0	0	0	0	0	1	1	0	1	0	0	0	*rb*				jp *%rb*	3	×	×
0	0	0	0	0	1	1	1	0	0	0	0	*rb*				call.d *%rb*	3	×	×
0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	ret.d	3	×	×
0	0	0	0	0	1	1	1	1	0	0	0	*rb*				jp.d *%rb*	2	×	×

Class 0 (2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S
Class			op1				op2			0	1	*rs,rd,ss,sd*							
0	0	0	0	0	0	0	0	0	0	0	1	*rs*				**push *%rs***	2	×	×
0	0	0	0	0	0	0	0	0	1	0	1	*rd*				**pop *%rd***	1	×	×
0	0	0	0	0	0	0	0	1	0	0	1	*ss*				**pushs *%ss***	2(alr),3(ahr)	×	×
0	0	0	0	0	0	0	0	1	1	0	1	*sd*				**pops *%sd***	2(alr),3(ahr)	×	×
0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	**ld.cf**	3	×	×

Class 0 (3)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S
Class			op1				d	*sign8*											
0	0	0	0	1	0	0	0	*sign8*								jrgt *sign8*	3	○	×
0	0	0	0	1	0	0	1	*sign8*								jrgt.d *sign8*	2	○	×
0	0	0	0	1	0	1	0	*sign8*								jrge *sign8*	3	○	×
0	0	0	0	1	0	1	1	*sign8*								jrge.d *sign8*	2	○	×
0	0	0	0	1	1	0	0	*sign8*								jrlt *sign8*	3	○	×
0	0	0	0	1	1	0	1	*sign8*								jrlt.d *sign8*	2	○	×
0	0	0	0	1	1	1	0	*sign8*								jrle *sign8*	3	○	×
0	0	0	0	1	1	1	1	*sign8*								jrle.d *sign8*	2	○	×
0	0	0	1	0	0	0	0	*sign8*								jrugt *sign8*	3	○	×
0	0	0	1	0	0	0	1	*sign8*								jrugt.d *sign8*	2	○	×
0	0	0	1	0	0	1	0	*sign8*								jruge *sign8*	3	○	×
0	0	0	1	0	0	1	1	*sign8*								jruge.d *sign8*	2	○	×
0	0	0	1	0	1	0	0	*sign8*								jrult *sign8*	3	○	×
0	0	0	1	0	1	0	1	*sign8*								jrult.d *sign8*	2	○	×
0	0	0	1	0	1	1	0	*sign8*								jrule *sign8*	3	○	×
0	0	0	1	0	1	1	1	*sign8*								jrule.d *sign8*	2	○	×
0	0	0	1	1	0	0	0	*sign8*								jreq *sign8*	3	○	×
0	0	0	1	1	0	0	1	*sign8*								jreq.d *sign8*	2	○	×
0	0	0	1	1	0	1	0	*sign8*								jrne *sign8*	3	○	×
0	0	0	1	1	0	1	1	*sign8*								jrne.d *sign8*	2	○	×
0	0	0	1	1	1	0	0	*sign8*								call *sign8*	4	○	×
0	0	0	1	1	1	0	1	*sign8*								call.d *sign8*	3	○	×
0	0	0	1	1	1	1	0	*sign8*								jp *sign8*	3	○	×
0	0	0	1	1	1	1	1	*sign8*								jp.d *sign8*	2	○	×

Class 1

15	14	13	12	11	10	9	8	7 6 5 4	3 2 1 0	Mnemonic	Cycle	Extension	Delayed S
Class			op1			op2		*imm5,rb,rs*	*rs,rd*				
0	0	1	0	0	0	0	0	*rb*	*rd*	ld.b %*rd*,[%*rb*]	1,2(ext)	○	×
0	0	1	0	0	0	0	1	*rb*	*rd*	ld.b %*rd*,[%*rb*]+	2	×	×
0	0	1	0	0	0	1	0	*rs*	*rd*	add %*rd*,%*rs*	1	○	○
0	0	1	0	0	0	1	1	*imm5*(3:0)	*rd*	**srl %*rd*,*imm5***	1	×	○
0	0	1	0	0	1	0	0	*rb*	*rd*	ld.ub %*rd*,[%*rb*]	1,2(ext)	○	×
0	0	1	0	0	1	0	1	*rb*	*rd*	ld.ub %*rd*,[%*rb*]+	2	×	×
0	0	1	0	0	1	1	0	*rs*	*rd*	sub %*rd*,%*rs*	1	○	○
0	0	1	0	0	1	1	1	*imm5*(3:0)	*rd*	**sll %*rd*,*imm5***	1	×	○
0	0	1	0	1	0	0	0	*rb*	*rd*	ld.h %*rd*,[%rb]	1,2(ext)	○	×
0	0	1	0	1	0	0	1	*rb*	*rd*	ld.h %*rd*,[%*rb*]+	2	×	×
0	0	1	0	1	0	1	0	*rs*	*rd*	cmp %*rd*,%*rs*	1	○	○
0	0	1	0	1	0	1	1	*imm5*(3:0)	*rd*	**sra %*rd*,*imm5***	1	×	○
0	0	1	0	1	1	0	0	*rb*	*rd*	ld.uh %*rd*,[%*rb*]	1,2(ext)	○	×
0	0	1	0	1	1	0	1	*rb*	*rd*	ld.uh %*rd*,[%*rb*]+	2	×	×
0	0	1	0	1	1	1	0	*rs*	*rd*	ld.w %*rd*,%*rs*	1	×	○
0	0	1	0	1	1	1	1	*imm5*(3:0)	*rd*	**sla %*rd*,*imm5***	1	×	○
0	0	1	1	0	0	0	0	*rb*	*rd*	ld.w %*rd*,[%*rb*]	1,2(ext)	○	×
0	0	1	1	0	0	0	1	*rb*	*rd*	ld.w %*rd*,[%*rb*]+	2	×	×
0	0	1	1	0	0	1	0	*rs*	*rd*	and %*rd*,%*rs*	1	○	○
0	0	1	1	0	0	1	1	*imm5*(3:0)	*rd*	**rr %*rd*,*imm5***	1	×	○
0	0	1	1	0	1	0	0	*rb*	*rs*	ld.b [%*rb*],%*rs*	1,2(ext)	○	×
0	0	1	1	0	1	0	1	*rb*	*rs*	ld.b [%*rb*]+,%*rs*	2	×	×
0	0	1	1	0	1	1	0	*rs*	*rd*	or %*rd*,%*rs*	1	○	○
0	0	1	1	0	1	1	1	*imm5*(3:0)	*rd*	**rl %*rd*,*imm5***	1	×	○
0	0	1	1	1	0	0	0	*rb*	*rs*	ld.h [%*rb*],%*rs*	1,2(ext)	○	×
0	0	1	1	1	0	0	1	*rb*	*rs*	ld.h [%*rb*]+,%*rs*	2	×	×
0	0	1	1	1	0	1	0	*rs*	*rd*	xor %*rd*,%*rs*	1	○	○
0	0	1	1	1	1	0	0	*rb*	*rs*	ld.w [%*rb*],%*rs*	1,2(ext)	○	×
0	0	1	1	1	1	0	1	*rb*	*rs*	ld.w [%*rb*]+,%*rs*	2	×	×
0	0	1	1	1	1	1	0	*rs*	*rd*	not %*rd*,%*rs*	1	×	○

Class 2

15	14	13	12	11	10	9 8 7 6 5 4	3 2 1 0	Mnemonic	Cycle	Extension	Delayed S
Class			op1			*imm6*	*rs,rd*				
0	1	0	0	0	0	*imm6*	*rd*	ld.b %*rd*,[%sp+*imm6*]	2	○	×
0	1	0	0	0	1	*imm6*	*rd*	ld.ub %*rd*,[%sp+*imm6*]	2	○	×
0	1	0	0	1	0	*imm6*	*rd*	ld.h %*rd*,[%sp+*imm6*]	2	○	×
0	1	0	0	1	1	*imm6*	*rd*	ld.uh %*rd*,[%sp+*imm6*]	2	○	×
0	1	0	1	0	0	*imm6*	*rd*	ld.w %*rd*,[%sp+*imm6*]	2	○	×
0	1	0	1	0	1	*imm6*	*rs*	ld.b [%sp+*imm6*],%*rs*	2	○	×
0	1	0	1	1	0	*imm6*	*rs*	ld.h [%sp+*imm6*],%*rs*	2	○	×
0	1	0	1	1	1	*imm6*	*rs*	ld.w [%sp+*imm6*],%*rs*	2	○	×

Class 3

15	14	13	12	11	10	9 8 7 6 5 4	3 2 1 0	Mnemonic	Cycle	Extension	Delayed S
Class			op1			*imm6,sign6*	*rd*				
0	1	1	0	0	0	*imm6*	*rd*	add %*rd*,*imm6*	1	○	○
0	1	1	0	0	1	*imm6*	*rd*	sub %*rd*,*imm6*	1	○	○
0	1	1	0	1	0	*sign6*	*rd*	cmp %*rd*,*sign6*	1	○	○
0	1	1	0	1	1	*sign6*	*rd*	ld.w %*rd*,*sign6*	1	○	○
0	1	1	1	0	0	*sign6*	*rd*	**and %*rd*,*sign6***	1	○	○
0	1	1	1	0	1	*sign6*	*rd*	**or %*rd*,*sign6***	1	○	○
0	1	1	1	1	0	*sign6*	*rd*	**xor %*rd*,*sign6***	1	○	○
0	1	1	1	1	1	*sign6*	*rd*	**not %*rd*,*sign6***	1	○	○

Class 4 (1)

Class (15–13)	op1 (12–10)	imm10 (9–0)	Mnemonic	Cycle	Extension	Delayed S
1 0 0	0 0 0	*imm10*	`add %sp,imm10`	1	×	○
1 0 0	0 0 1	*imm10*	`sub %sp,imm10`	1	×	○

Class 4 (2)

Class (15–13)	op1 (12–10)	op2 (9–8)	imm5,rs (7–4)	rd (3–0)	Mnemonic	Cycle	Extension	Delayed S
1 0 0	0 1 0	0 0	*imm5*(3:0)	*rd*	**`srl %rd,imm5`**	1	×	○
1 0 0	0 1 0	0 1	*rs*	*rd*	**`srl %rd,%rs`**	1	×	○
1 0 0	0 1 1	0 0	*imm5*(3:0)	*rd*	**`sll %rd,imm5`**	1	×	○
1 0 0	0 1 1	0 1	*rs*	*rd*	**`sll %rd,%rs`**	1	×	○
1 0 0	1 0 0	0 0	*imm5*(3:0)	*rd*	**`sra %rd,imm5`**	1	×	○
1 0 0	1 0 0	0 1	*rs*	*rd*	**`sra %rd,%rs`**	1	×	○
1 0 0	1 0 0	1 0	*rs*	*rd*	`swap %rd,%rs`	1	×	○
1 0 0	1 0 1	0 0	*imm5*(3:0)	*rd*	**`sla %rd,imm5`**	1	×	○
1 0 0	1 0 1	0 1	*rs*	*rd*	**`sla %rd,%rs`**	1	×	○
1 0 0	1 1 0	0 0	*imm5*(3:0)	*rd*	**`rr %rd,imm5`**	1	×	○
1 0 0	1 1 0	0 1	*rs*	*rd*	**`rr %rd,%rs`**	1	×	○
1 0 0	1 1 0	1 0	*rs*	*rd*	**`swaph %rd,%rs`**	1	×	○
1 0 0	1 1 1	0 0	*imm5*(3:0)	*rd*	**`rl %rd,imm5`**	1	×	○
1 0 0	1 1 1	0 1	*rs*	*rd*	**`rl %rd,%rs`**	1	×	○

Class 5 (1)

Class (15–13)	op1 (12–10)	op2 (9–8)	imm4,r,s (7–4)	imm3,r,s (3–0)	Mnemonic	Cycle	Extension	Delayed S
1 0 1	0 0 0	0 0	*rs*	*sd*	**`ld.w %sd,%rs`**	1,3(psr)	×	×
1 0 1	0 0 0	0 1	*rs*	*rd*	`ld.b %rd,%rs`	1	×	○
1 0 1	0 0 0	1 0	*rs*	*rd*	`mlt.h %rd,%rs`	5	×	×
1 0 1	0 0 1	0 0	*ss*	*rd*	**`ld.w %rd,%ss`** *1	1	×	×
1 0 1	0 0 1	0 1	*rs*	*rd*	`ld.ub %rd,%rs`	1	×	○
1 0 1	0 0 1	1 0	*rs*	*rd*	`mltu.h %rd,%rs`	5	×	×
1 0 1	0 1 0	0 0	*rb*	0 *imm3*	`btst [%rb],imm3`	2,3(ext)	○	×
1 0 1	0 1 0	0 1	*rs*	*rd*	`ld.h %rd,%rs`	1	×	○
1 0 1	0 1 0	1 0	*rs*	*rd*	`mlt.w %rd,%rs`	7	×	×
1 0 1	0 1 1	0 0	*rb*	0 *imm3*	`bclr [%rb],imm3`	3,4(ext)	○	×
1 0 1	0 1 1	0 1	*rs*	*rd*	`ld.uh %rd,%rs`	1	×	○
1 0 1	0 1 1	1 0	*rs*	*rd*	`mltu.w %rd,%rs`	7	×	×
1 0 1	1 0 0	0 0	*rb*	0 *imm3*	`bset [%rb],imm3`	3,4(ext)	○	×
1 0 1	1 0 0	0 1	*imm4*	*rd*	**`ld.c %rd,imm4`**	1	×	○
1 0 1	1 0 1	0 0	*rb*	0 *imm3*	`bnot [%rb],imm3`	3,4(ext)	○	×
1 0 1	1 0 1	0 1	*imm4*	*rs*	**`ld.c imm4,%rs`**	1	×	○
1 0 1	1 1 0	0 0	*rs*	*rd*	`adc %rd,%rs`	1	×	○
1 0 1	1 1 1	0 0	*rs*	*rd*	`sbc %rd,%rs`	1	×	○

Class 5 (2)

Class (15–13)	op1 (12–10)	op2 (9–8)	op3 (7–6)	imm5,imm6 (5–0)	Mnemonic	Cycle	Extension	Delayed S
1 0 1	1 1 1	1 1	0 0	*imm6*	**`do.c imm6`**	1	×	×
1 0 1	1 1 1	1 1	0 1	0 *imm5*	**`psrset imm5`**	3	×	×
1 0 1	1 1 1	1 1	1 0	0 *imm5*	**`psrclr imm5`**	3	×	×

Class 6

Class (15–13)	imm13 (12–0)	Mnemonic	Cycle	Extension	Delayed S
1 1 0	*imm13*	**`ext imm13`**	0,1	×	×

`Inst` Function-Extended Instructions

`Inst` (shaded) Added Instructions

*1 The `ld.w %rd,%pc` instruction must be executed as a delayed slot instruction. If it does not follow a delayed branch instruction, the PC value that is loaded into the *rd* register may not be the next instruction address to the `ld.w` instruction.

EPSON

International Sales Operations

AMERICA

EPSON ELECTRONICS AMERICA, INC.

HEADQUARTERS
150 River Oaks Parkway
San Jose, CA 95134, U.S.A.
Phone: +1-800-228-3964 Fax: +1-408-922-0238

SALES OFFICE
Northeast
301 Edgewater Place, Suite 210
Wakefield, MA 01880, U.S.A.
Phone: +1-800-922-7667 Fax: +1-781-246-5443

EUROPE

EPSON EUROPE ELECTRONICS GmbH

HEADQUARTERS
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-89-14005-0 Fax: +49-89-14005-110

DÜSSELDORF BRANCH OFFICE
Altstadtstrasse 176
51379 Leverkusen, GERMANY
Phone: +49-2171-5045-0 Fax: +49-2171-5045-10

FRENCH BRANCH OFFICE
1 Avenue de l' Atlantique, LP 915 Les Conquerants
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE
Phone: +33-1-64862350 Fax: +33-1-64862355

UK & IRELAND BRANCH OFFICE
8 The Square, Stockley Park, Uxbridge
Middx UB11 1FW, UNITED KINGDOM
Phone: +44-1295-750-216/+44-1342-824451
Fax: +44-89-14005 446/447

Scotland Design Center
Integration House, The Alba Campus
Livingston West Lothian, EH54 7EG, SCOTLAND
Phone: +44-1506-605040 Fax: +44-1506-605041

ASIA

EPSON (CHINA) CO., LTD.
23F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: +86-10-6410-6655 Fax: +86-10-6410-7320

SHANGHAI BRANCH
7F, High-Tech Bldg., 900, Yishan Road
Shanghai 200233, CHINA
Phone: +86-21-5423-5522 Fax: +86-21-5423-5512

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

EPSON Electronic Technology Development (Shenzhen) LTD.
12/F, Dawning Mansion, Keji South 12th Road
Hi- Tech Park, Shenzhen
Phone: +86-755-2699-3828 Fax: +86-755-2699-3838

EPSON TAIWAN TECHNOLOGY & TRADING LTD.
14F, No. 7, Song Ren Road
Taipei 110
Phone: +886-2-8786-6688 Fax: +886-2-8786-6677

EPSON SINGAPORE PTE., LTD.
1 HarbourFront Place
#03-02 HarbourFront Tower One, Singapore 098633
Phone: +65-6586-5500 Fax: +65-6271-3182

SEIKO EPSON CORPORATION
KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: +82-2-784-6027 Fax: +82-2-767-3677

GUMI OFFICE
2F, Grand B/D, 457-4 Songjeong-dong
Gumi-City, KOREA
Phone: +82-54-454-6027 Fax: +82-54-454-6093

SEIKO EPSON CORPORATION
SEMICONDUCTOR OPERATIONS DIVISION

IC Sales Dept.
IC International Sales Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-42-587-5814 Fax: +81-42-587-5117

S1C33 Family C33 PE
Core Manual

SEIKO EPSON CORPORATION
SEMICONDUCTOR OPERATIONS DIVISION

■ **EPSON Electronic Devices Website**

http://www.epsondevice.com

Document code: 410755500
Issue July, 2006
Printed in Japan Ⓛ A