EPSON

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER

S1C33 Family C33 ADV Core CPU Manual

SEIKO EPSON CORPORATION

Configuration of product number

Devices

S1 C 33209 F 00E1 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	

Specification

Package
[D: die form; F: QFP]

Model number

Model name
[C: microcomputer, digital products]

Product classification
[S1: semiconductor]

Development tools

S5U1 C 33000 H2 1 00

Packing specifications
[00: standard packing]

Version
[1: Version 1]

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

Corresponding model number
[33L01: for S1C33L01]

Tool classification
[C: microcomputer use]

Product classification
[S5U1: development tool for semiconductor products]

– Contents –

1 Summary

The C33 ADV Core CPU is a high-end RISC computer in the S1C33 series of Seiko Epson 32-bit microcomputers featuring the extended functionality of the instruction set, with new instructions added, low power consumption, and high processing speed. The C33 ADV Core CPU adopts an input instruction queue and 5-stage pipelined processing, providing a computer architecture that saves more resources and is more efficient than ever.
Furthermore, the C33 ADV Core CPU includes as standard features the multiplier and the multiply-accumulate instructions that conventionally were available as options in the S1C33 series. Combined with the newly added instructions, they will help to accomplish multimedia-related processing easily.
As the C33 ADV Core CPU is upward object-code compatible with the C33 STD Core CPU, the software assets of the user that have been amassed in the past can be effectively utilized.
What's more, when the C33 ADV Core CPU is combined with a Memory Management Unit (MMU) and a Cache Control Unit (CCU) to configure the CPU core, even faster and more advanced processing will be made possible.

1.1 Features

Processor type

- Seiko Epson original 32-bit RISC CPU
- 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)

Instruction set

- Instruction set useful for multimedia processing

• Code length	16-bit fixed length
• Number of instructions	164
• Execution cycle	Main instructions executed in one cycle Two to three instructions (including immediate-extended instructions) can be executed in one clock cycle
• Extended immediate instructions	Immediate extended up to 32 bits

Multimedia features

• Multiplication instructions	Multiplications for 16 × 16, 32 × 16, and 32 × 32 bits supported
• Multiply-accumulate instructions	Step/continuous multiply-accumulate operations for 16 × 16, 32 × 16, and 32 × 32 bits supported
• Loop instruction	Specified range executed repeatedly
• Repeat instruction	One instruction executed repeatedly
• Saturation instruction	Rounded to minimum/maximum values
• Postshift function	ALU instruction execution with postshift supported

Register set

- 32-bit general-purpose registers
- 32-bit special registers
- 32-bit multiply-accumulate operation registers

Memory space and external bus

- Instruction, data, and I/O coexisting linear space
- Up to 4G bytes of memory space
- Little endian format (can be switched to big endian)

Interrupts

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

Reset

- Cold reset (all internal circuits reset)
- Hot reset (bus and port statuses retained)

Power-down mode

- HALT mode (only the CPU core turned off)
- SLEEP mode (CPU core and oscillator circuit turned off)

Others

- MMU supported
- Caches supported

1.2 Summary of Added/Changed Functions of the C33 ADV

The functions below have been added to or changed for the C33 ADV Core CPU, 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 instruction set of the C33 ADV Core CPU is upward compatible with the C33 STD Core CPU. For functional enhancement, however, the functionality of some existing instructions has been extended and new instructions added as described below.

Function-extended instructions

The C33 ADV Core CPU 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 postincrement instructions ([*%rb*]+) have been modified to support address extension by ext.

ext	*offset*	
ld.*t*	[*%rb*]+,*%rs*	Used for ext extended instruction, *t* = b, h, w
ext	*offset*	
ld.*t*	*%rd*,[*%rb*]+	Used for ext extended instruction, *t* = b, ub, h, uh, w

3. 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

4. The number of bits scanned by the scan instruction has been increased from 8 to 32.

scan0	*%rd*,*%rs*	Number of scan bits extended to 32 bits
scan1	*%rd*,*%rs*	Number of scan bits extended to 32 bits

Added instructions

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

1. The ext instructions that support three operands in the target instruction, control of whether to execute depending on flag status, and post-shift in arithmetic instructions (up to 3 bits) have been added.

ext	*%rs*	Expands to 3 operands
ext	*cond*	Conditional execution
ext	*op*,*imm2*	Postshift
ext	*%rs*,*op*,*imm2*	Expands to 3 operands + postshift

2. Since the C33 ADV Core CPU now comes standard with a 32-bit × 16-bit multiplier, multiply-accumulate instructions have been enhanced accordingly.
   - 16 bits × 16 bits, 32 bits × 16 bits, and 32 bits × 32 bits are supported.
   - Register-based stepping instructions have been added for multiply-accumulate processing.
   - Instructions to initialize the arithmetic operation registers have been added.

mlt.hw	*%rd*,*%rs*	Multiplication, 32 bits × 16 bits → 48 bits
mac.hw	*%rs*	Multiply-accumulate, 32 bits × 16 bits + 64 bits → 64 bits
mac.w	*%rs*	Multiply-accumulate, 32 bits × 32 bits + 64 bits → 64 bits
mac1.h	*%rd*,*%rs*	Single-operation multiply-accumulate, 16 bits × 16 bits + 64 bits → 64 bits
mac1.hw	*%rd*,*%rs*	Single-operation multiply-accumulate, 32 bits × 16 bits + 64 bits → 64 bits
mac1.w	*%rd*,*%rs*	Single-operation multiply-accumulate, 32 bits × 32 bits + 64 bits → 64 bits
macclr		Clears AHR and ALR registers and MO flag

3. Loop and repeat instructions to execute high-speed repeat processing have been added. A set of instructions can now be executed consecutively at high speed without branching.

`loop`	*%rc*,*%ra*	Loops specified range
`loop`	*%rc*,*imm4*	Loops specified range
`loop`	*imm4*,*imm4*	Loops specified range
`repeat`	*%rb*	Repeat
`repeat`	*imm4*	Repeat

4. DP relative addressed memory access instructions have been added.
By setting the start address of a data area in the DP register, it is now possible to access desired memory locations using fewer instructions through relative addressing.

`ld.`*t*	*%rd*,[%dp+*imm6*]	DP register indirect load, *t* = b, ub, h, uh, w
`ld.`*t*	[%dp+*imm6*],*%rs*	DP register indirect store, *t* = b, h, w
`add`	*%rd*,%dp	Addition, DP register is added as an operand

5. 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

6. 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

7. Other special instructions have been added.

`sat.`*t*	*%rd*	Saturation, *t* = b, ub, h, uh, w, uw
`div.w`		Signed division, 32 bits / 32 bits → 16 bits ... 16 bits
`divu.w`		Unsigned division, 32 bits / 32 bits → 16 bits ... 16 bits
`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
`retm`		Returns from the MMU exeception handler routine

Note: Depending on how instructions are used or the PSR register is set, not all existing instructions can be guaranteed to be fully compatible with the predecessor.

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 significantly 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.

Stack pointer

Stack pointers SSP and USP have been added.
The C33 ADV Core CPU has two operation modes: Supervisor Mode and User Mode. Therefore, it has separate stack pointers for use in each mode. SSP is used in supervisor mode; USP is used in user mode. The existing SP is no longer a physical register, and access to the SP by an instruction with operand "`%sp`" is made to SSP or USP in the existing mode.

Data pointer

A data pointer (DP) has been added.
It is designed to enable the efficient access of memory locations using fewer instructions.
Set the start address of a data area in the DP and specify the address in "DP + offset" format in an instruction as conventionally used to specify "SP + offset" to load or save data to and from memory. This helps reduce the number of `ext` instructions used.

Trap table base register

A trap table base register (TTBR) has been added.
In the C33 STD Core CPU, TTBR was mapped to address 0x48134, with its initial value set to 0xC00000.
In the C33 ADV Core CPU, TTBR operates as an internal special register of the CPU, and its initial value (boot address) has been changed to 0x20000000.

Shift-out register

A shift-out register (SOR) has been added.
This register is used to accommodate the bits shifted out from the specified register by a shift instruction. This register should prove useful when more than 32 bits must be shifted, as in floating-point arithmetic or image processing.

Loop-related registers

To support loop and repeat instructions, three registers have been added. These include the Loop Start Address (LSA) register, Loop End Address (LEA) register, and Loop Counter (LCO).
These registers contain the address range in which a loop or repeat instruction is to be executed, as well as the number of times the instruction is to be executed.

Arithmetic operation registers

ALR and AHR are the same as in the C33 STD Core CPU.
In the C33 ADV Core CPU, modification has been made to load results of multiply, divide, and multiply-accumulate operations that are written to ALR and AHR into the R4 (= ALR) and R5 (= AHR) registers at the same time by setting the PSR bits (note, however, that ALR and AHR cannot be accessed through R4 and R5). Therefore, the result of a multiply/divide or multiply-accumulate operation can be referenced directly in a data transfer or arithmetic operation instruction.

CPU identification register

A CPU 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 bits have been added to the Processor Status Register (PSR):

HE	Enables the use of `halt` and `slp` instructions in user mode.
RM	Indicates whether a repeat instruction is being executed.
LM	Indicates whether a loop instruction is being executed.
PM	Indicates whether a consecutive push/pop is being executed.
RC	Contains the register number whose content is being consecutively pushed or popped.
SW	Switches between 8-bit and 32-bit scans.
OC	Switches over V flag processing in logical operation.
SE	Switches over C and V flag processing in shift operation.
LC	Selects a map of ALR to R4.
HC	Selects a map of AHR to R5.
S	Indicates whether saturation occurred when executing a saturation instruction.
DE	Indicates the status of debug exceptions generated.
ME	Indicates the status of MMU exceptions generated.
SV	Selects between supervisor and user modes.

1.2.3 Address Space, Modes, and Other

Address space

The C33 ADV Core CPU supports a 4G-byte space based on a 32-bit address bus.
With the HBCU and MMU supported, the space accessed by the CPU and the space to which actual memory and other devices are mapped can now be handled as two different spaces: "logical address space" and "physical address space."
The logical address space (4GB) is equally divided by the HBCU into eight 0.5GB blocks where MMU and ASID-related settings can be made independently. For blocks managed by the MMU, their logical addresses are translated into physical addresses in 4KB or 64KB units. Moreover, 64 process × 64MB multiple virtual spaces can be realized by using the ASID.
The physical address space is divided by the #CE signal into 22 areas where memory and other devices can actually be located.

Supervisor and user modes

Two distinct access modes have been created: supervisor mode (in which all resources can be accessed) and user mode (in which accessible resources are limited).

Limitations on user mode

1. Memory access limitations
   Setting up the MMU as required can disable access to supervisor space or any pages in user mode.
   When using the ASID, a limitation is imposed whereby any process (ASID) cannot access other processes beyond the 64KB area.
   The HBCU can be set to forcibly use the MMU and ASID in all memory spaces during user mode.
2. Register access limitations
   Some special registers can only be accessed in supervisor mode.
3. Instruction execution limitations
   Setting up the PSR as required can disable execution of the `slp` and `halt` instructions from user mode.
4. Interrupt mode limitations
   Since the CPU enters supervisor mode whenever an exception (i.e., normal interrupt, MMU exception, debug exception) occurs, exceptions cannot be directly managed in user mode.

Other

1. MMU support
   The C33 ADV Core CPU supports MMU exceptions that may occur when fetching instructions or accessing data via the MMU. Moreover, the `retm` instruction to return from MMU exceptions has been added.
2. Interrupt processing
   The Trap Table Base Register (TTBR) now serves as an internal special register of the CPU, with the boot address at cold reset changed to 0x20000000. Although normal interrupts, software interrupts, and exceptions are all processed the same way as in the C33 STD Core CPU, the difference is that the CPU enters supervisor mode whenever an exception occurs.
   Multiple occurrences of NMI exceptions are disabled internally in the CPU. NMI request signal trigger mode can now also be selected in software.
3. Pipeline
   The 3-stage pipeline in the C33 STD Core CPU has been upgraded to a 5-stage pipeline in the C33 ADV Core CPU (consisting of fetch, decode, execute, access, and write back) for supporting instruction processing at even higher clock frequencies.
   Moreover, the `ext` instruction is processed in parallel with a normal other instruction (i.e., target instruction) so that up to two `ext` instructions for the respective extended instructions can, in effect, be executed with zero clock cycles (requiring only clock cycles of the target instruction).

2 Registers

The C33 ADV Core CPU contains 16 general-purpose registers and 15 special registers.

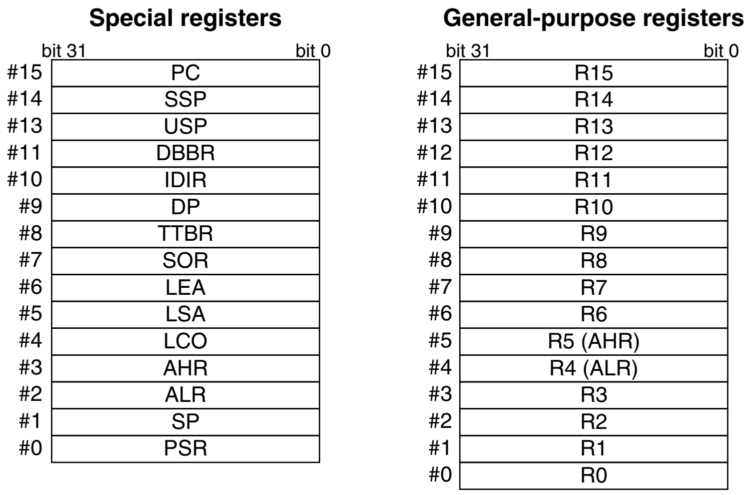


Figure 2.1 Registers

Table 2.1 Register Access Rights

Register symbol	Name	Supervisor mode	User mode
PC	Program counter	R	R
SSP *1	Supervisor stack pointer	R/W	R
USP *1	User stack pointer	R/W	R/W
DBBR *1	Debug base register	R	R
IDIR *1	CPU identification register	R	R
DP *1	Data pointer	R/W	R/W
TTBR *1	Trap table base register	R/W	R
SOR *1	Shift out register	R/W	R/W
LEA *1	Loop end address register	R/W	R/W
LSA *1	Loop start address register	R/W	R/W
LCO *1	Loop count register	R/W	R/W
AHR	Arithmetic-operation high register	R/W	R/W
ALR	Arithmetic-operation low register	R/W	R/W
SP *2	Stack pointer	R/W	R/W
PSR	Processor status register	R/W	R/W *3

∗1 New registers added to the C33 ADV Core CPU

∗2 When the SP register is referenced, either SSP or USP is referenced.

∗3 Some bits in the PSR cannot be accessed for writing in user mode.

2.1 General-Purpose Registers (R0–R15)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
R0–R15	General-Purpose Register	32 bits	R/W	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 ADV Core CPU, 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	Supervisor mode	User mode	Initial value
PC	Program Counter	32 bits	R	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 ADV Core CPU are fixed at 16 bits in length, the low-order one bit of the PC (bit 0) is always 0.
Although the C33 ADV Core CPU 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 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 CPU starts executing a program from the address indicated by the PC.
During cold reset, TTBR is initialized to "0x20000000," so that the address written at the address "0x20000000" is the start address of the program.

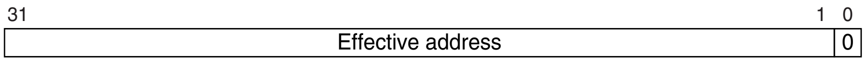


Figure 2.2.1 Program Counter (PC)

2.3 Processor Status Register (PSR)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
PSR	Processor Status Register	32 bits	R/W *1	R/W *1	0x00000000

*1 Some bits in the PSR cannot be accessed for writing. This limitation differs between supervisor mode and user mode.

The Processor Status Register (hereinafter referred to as the "PSR") is a 32-bit register for storing the internal status of the CPU.
The PSR stores the internal status of the CPU 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. Although the PSR can be altered by a program, some bits in it cannot be accessed for writing. Refer to Figure 2.3.1.
As the PSR affects program execution, whenever an interrupt or exception occurs, the PSR is saved to the stack, except for MMU and debug exceptions, to maintain the PSR value. The specific bits in it—RM (bit 30), LM (bit 29), PM (bit 28), and IE (bit 4)—are 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	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	HE	RM	LM	PM	RC[3:0]				–	SW	OC	SE	–	–	LC	HC
Initial value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Supervisor mode	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R	R/W	R/W
User mode	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R	R/W	R/W

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	S	DE	ME	SV	IL[3:0]				MO	DS	–	IE	C	V	Z	N
Initial value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Supervisor mode	R/W	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
User mode	R/W	R	R	R	R	R	R	R	R/W	R/W	R	R	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.

HE (bit 31): Halt and Sleep Enable

When this flag = 1, the `halt` and `slp` instructions are enabled, even in user mode. The CPU status is unaffected by this flag in supervisor mode. This flag cannot be altered in user mode.

RM (bit 30): Repeat Mode Enable

This bit is set to 1 when the `repeat` instruction is executed. While this flag remains set, the CPU successively executes the instructions that follow the `repeat` instruction. If an interrupt or exception is accepted during execution of the `repeat` instruction, the PSR is saved to the stack and this bit is cleared to 0. However, the PSR is not saved to the stack for MMU and debug exceptions, nor is this bit cleared to 0. Therefore, the PSR must be protected in an exception handler routine.

LM (bit 29): Loop Mode Enable

This bit is set to 1 when the `loop` instruction is executed. While this flag remains set, the CPU repeatedly executes a range of instructions from the LSA to the LEA registers, as many times as specified by the LCO register. If an interrupt or exception is accepted during execution of the `loop` instruction, the PSR is saved to the stack and this bit is cleared to 0. However, the PSR is not saved to the stack for MMU and debug exceptions, nor is this bit cleared to 0. Therefore, the PSR must be protected in an exception handler routine.

PM (bit 28): Push/Pop Mode

This bit is set to 1 when the `pushn`, `popn`, `pushs`, or `pops` instruction is executed, and remains set until the last register is saved to or restored from the stack. If the `pushn`, `popn`, `pushs`, or `pops` instruction is executed when this bit = 1, the CPU starts a push/pop operation on registers beginning with that set in RC[3:0] (bits 27–24), and continues operating until the last register is saved to or restored from the stack. If an interrupt or exception is accepted during execution of the `pushn`, `popn`, `pushs`, or `pops` instruction, the PSR is saved to the stack and this bit is cleared to 0. However, the PSR is not saved to the stack for MMU and debug exceptions, nor is this bit cleared to 0. Therefore, the PSR must be protected in an exception handler routine.

RC[3:0] (bits 27–24): Register Counter

If an interrupt or exception occurs during execution of the `pushn`, `popn`, `pushs`, or `pops` instruction, the register number on which the CPU is executing a push/pop is stored in this counter. If the `pushn`, `popn`, `pushs`, or `pops` instruction is executed when PM (bit 28) = 1, the CPU starts saving or restoring registers beginning with that stored here.

SW (bit 22): Scan Word Enable

If the `scan` instruction is executed when this flag = 0, bits are scanned in 8-bit scan mode. If the `scan` instruction is executed when this flag = 1, bits are scanned in 32-bit scan mode.

OC (bit 21): Overflow Clear Enable

If a logical instruction is executed when this flag = 1, the V flag (bit 2) in the PSR is cleared.

SE (bit 20): Shift with Carry Enable

When this flag = 1, the bit shifted out in a shift instruction is stored in the C flag (bit 3) in the PSR. Furthermore, the V flag also changes state depending on the shift result.

LC (bit 17): ALR Change Enable

When this flag = 1, the data written to the ALR register in one of the following target instructions is also written to R4. These target instructions are the same as those for the HC (bit 16) flag.

Target instructions

```
mlt.h   %rd,%rs       mltu.h  %rd,%rs
mlt.hw  %rd,%rs
mlt.w   %rd,%rs       mltu.w  %rd,%rs
div0s   %rs           div0u   %rs
div1    %rs           div2s   %rs           div3s
div.w   %rs           divu.w  %rs
mac     %rs           mac.hw  %rs           mac.w   %rs
mac1.h  %rd,%rs       mac1.hw %rd,%rs       mac1.w  %rd,%rs
macclr
```

HC (bit 16): AHR Change Enable

When this flag = 1, the data written to the AHR register in one of the following target instructions is also written to R5. These target instructions are the same as those for the LC (bit 17) flag.

Target instructions

```
mlt.h   %rd,%rs       mltu.h  %rd,%rs
mlt.hw  %rd,%rs
mlt.w   %rd,%rs       mltu.w  %rd,%rs
div0s   %rs           div0u   %rs
div1    %rs           div2s   %rs           div3s
div.w   %rs           divu.w  %rs
mac     %rs           mac.hw  %rs           mac.w   %rs
mac1.h  %rd,%rs       mac1.hw %rd,%rs       mac1.w  %rd,%rs
macclr
```

S (bit 15): Saturation

This bit is set to 1 if saturation occurs during execution of a saturation instruction. Once set, the S flag remains set unless it is cleared in a program.

DE (bit 14): Debug Exception

This bit is set to 1 when a debug exception occurs. This DE flag is cleared to 0 by the `retd` instruction. This is a read-only flag.

ME (bit 13): MMU Exception

This bit is set to 1 when an MMU exception occurs. Once the ME flag is set, all of the subsequent exceptions are disabled and the MMU is placed in an idle state. Only the physical addresses are accessed during MMU exception handling. This ME flag is cleared to 0 by the `retm` instruction. This is a read-only flag.

SV (bit 12): SuperVisor Mode

When an interrupt or exception occurs, the CPU is placed in supervisor mode and the SV flag is cleared to 0. In supervisor mode, all resources can be accessed. When the SV flag = 1, the CPU is placed in user mode, in which writing to some registers is restricted. This flag cannot be altered in user mode. During an MMU exception, the CPU is always in supervisor mode irrespective of the SV flag.

IL[3:0] (bits 11–8): Interrupt Level

These bits indicate the priority levels of the CPU 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. This flag cannot be altered in user mode.

MO (bit 7): Mac Overflow

This bit indicates an overflow. More specifically, this bit is set to 1 when an intermediate result of a multiply-accumulate operation being executed exceeded the effective range of values representable by signed 64 bits. As the operation is executed until it finishes regardless of whether it overflowed, the MO bit should be read out after completion of the operation to determine whether the result is valid. Once the MO flag is set, it remains set until the PSR is initialized or the bit is explicitly reset by a program.

DS (bit 6): <u>D</u>ivide <u>S</u>ign

The sign bit of the dividend of a step division being executed is set in this DS flag, which affects the execution of the division.

Note: The proper DS value may not be obtained if PSR is read using the `ld.w` instruction immediately after the `div0s` or `div0u` instruction has been executed. To avoid this erroneous reading, insert two or more instructions between the `div0s` or `div0u` instruction and `ld.w` instruction that reads the DS flag.

IE (bit 4): <u>I</u>nterrupt <u>E</u>nable

This bit controls maskable external interrupts by accepting or disabling them. When IE bit = 1, the CPU enables maskable external interrupts. When IE bit = 0, the CPU disables maskable external interrupts.

This flag cannot be altered in user mode. 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 MMU and debug exceptions, nor is this bit cleared to 0.

C (bit 3): <u>C</u>arry

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): O<u>V</u>erflow

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.

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): <u>Z</u>ero

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): <u>N</u>egative

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	Supervisor mode	User mode	Initial value
SP	Stack Pointer	32 bits	R/W (SSP)	R/W (USP)	Indeterminate
SSP	Supervisor Stack Pointer	32 bits	R/W	R	Indeterminate
USP	User Stack Pointer	32 bits	R/W	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.
There are two types of SP—the Supervisor Stack Pointer (hereinafter referred to as the "SSP") and the User Stack Pointer (hereinafter referred to as the "USP"). When "SP" is specified in an instruction during operation in supervisor mode or user mode, either the SSP or the USP is referenced automatically according to the active mode. When the CPU is operating in user mode, the SSP cannot be referenced.
Although the SSP and the USP can both be referenced in supervisor mode, if the stack pointer is indirectly referenced from the register symbol "SP" in the `pushn` or `popn` instruction, it is always the SSP that is referenced.
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.

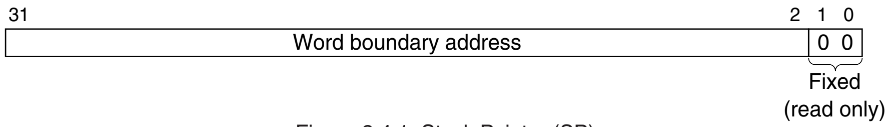


Figure 2.4.1 Stack Pointer (SP)

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`

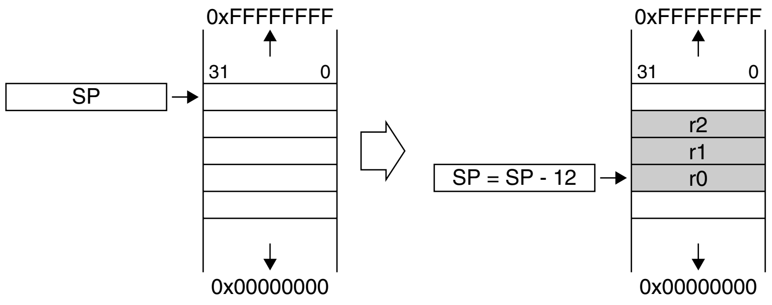


Figure 2.4.2.1 SP and Stack (1)

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`

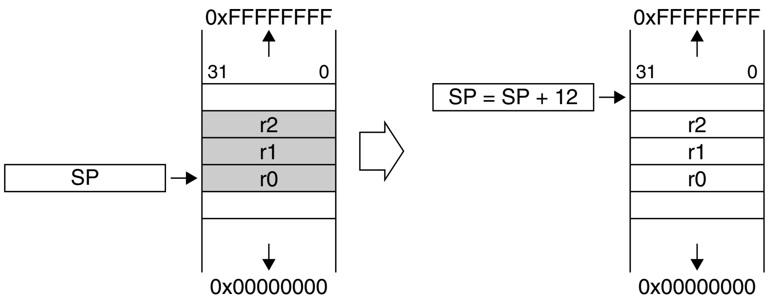


Figure 2.4.3.1 SP and Stack (2)

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]

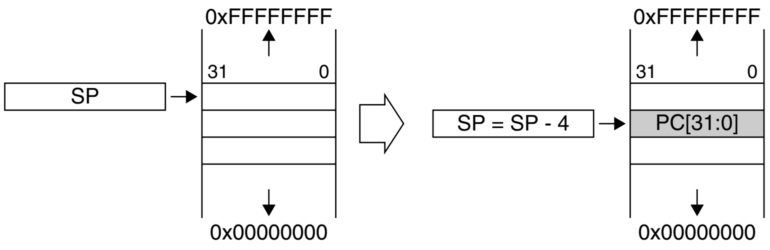


Figure 2.4.4.1 SP and Stack (3)

SP operation by the `ret` instruction

(1) [SP] → PC
(2) SP = SP + 4

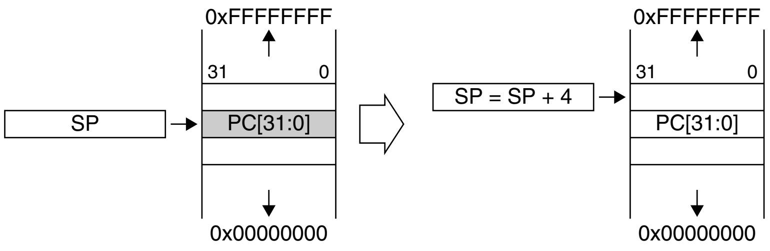


Figure 2.4.4.2 SP and Stack (4)

2.4.5 SP Operation when an Interrupt or Exception Occurs

If an interrupt or software exception resulting from the `int` instruction occurs, the CPU has its operation mode switched to supervisor mode and enters an exception handling process. Thereafter, only the SSP is operated on as the stack pointer.
The CPU pushes the contents of the PC and PSR onto the stack indicated by the SSP 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 SSP address is altered as shown in the diagram below.

SP operation when an interrupt occurred

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

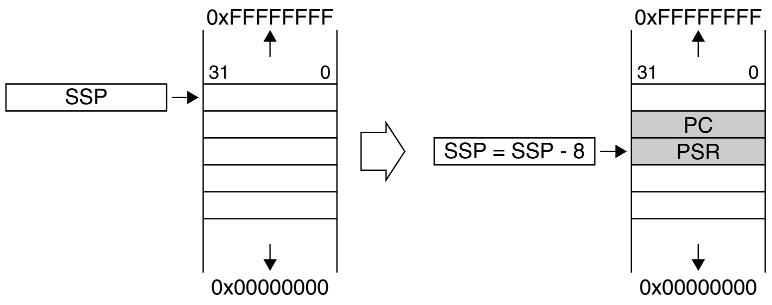


Figure 2.4.5.1 SP and Stack (5)

SP operation when the `reti` instruction is executed

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

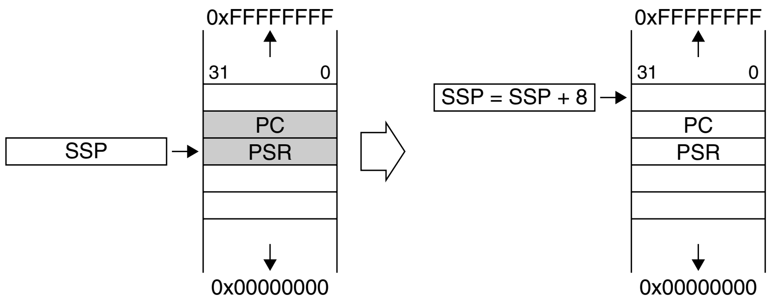


Figure 2.4.5.2 SP and Stack (6)

2.5 Data Pointer (DP)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
DP	Data Pointer	32 bits	R/W	R/W	Indeterminate

The Data Pointer (hereinafter referred to as the "DP") is a 32-bit register that can be used to store a memory address during register indirect addressing with displacement included in the load-related instructions. When combined with extended immediate instructions, the DP supports memory space of up to "DP + 32-bit immediate." However, arithmetic operations on the DP are not supported. Nor can the 2 low-order bits of the DP be accessed for writing, as they are fixed at 0. Therefore, the addresses specifiable by the DP are those that lie on word boundaries.

```
ld.b    %rd,[%dp+imm6]
ld.ub   %rd,[%dp+imm6]
ld.h    %rd,[%dp+imm6]
ld.uh   %rd,[%dp+imm6]
ld.w    %rd,[%dp+imm6]

ld.b    [%dp+imm6],%rs
ld.h    [%dp+imm6],%rs
ld.w    [%dp+imm6],%rs
```

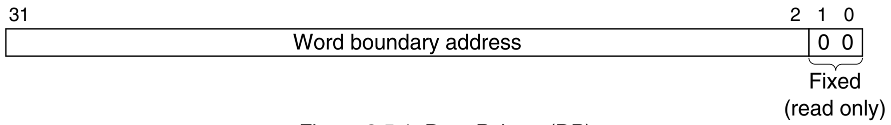


Figure 2.5.1 Data Pointer (DP)

2.6 Trap Table Base Register (TTBR)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
TTBR	Trap Table Base Register	32 bits	R/W	R	0x20000000

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 "0x20000000," 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.
The TTBR cannot be altered in user mode.

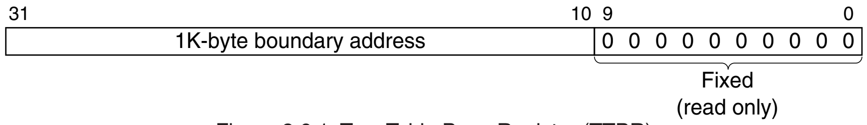


Figure 2.6.1 Trap Table Base Register (TTBR)

2.7 Shift Out Register (SOR)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
SOR	Shift Out Register	32 bits	R/W	R/W	Indeterminate

The Shift Out Register (hereinafter referred to as the "SOR") is a 32-bit register that is used to store the bit shifted out of a general-purpose register when a shift-related instruction is executed. It holds the result of the shift-related instruction that was last executed.

2.8 Loop End Address Register (LEA)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
LEA	Loop End Address Register	32 bits	R/W	R/W	Indeterminate

The Loop End Address Register (hereinafter referred to as the "LEA") is a 32-bit register that is used to store the last address in a range of instructions to be looped when the `loop` instruction is executed.
The fetch address and the LEA are compared whenever the `loop` instruction is executed and the LM flag in the PSR = 1, and the program branches to the address indicated by the LSA when the compared addresses match. At this time, if the LCO is decremented by 1 to become 0, the `loop` instruction is terminated and the LM flag in the PSR is cleared to 0, with control transferred to the instruction next to the LEA. If the LCO = 0 when the `loop` instruction is executed, the program does not branch to the address indicated by the LSA.
Bit 0 of LEA is always handled as 0.

2.9 Loop Start Address Register (LSA)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
LSA	Loop Start Address Register	32 bits	R/W	R/W	Indeterminate

The Loop Start Address Register (hereinafter referred to as the "LSA") is a 32-bit register that is used to store the first address in a range of instructions to be looped when the `loop` instruction is executed. Refer to the LEA.
In a repeat instruction, furthermore, it is used to store the address of the instruction to be repeated.
Bit 0 of LSA is always handled as 0.

2.10 Loop Count Register (LCO)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
LCO	Loop Count Register	32 bits	R/W	R/W	Indeterminate

The Loop Count Register (hereinafter referred to as the "LCO") is a 32-bit register that is used to store the loop count or the number of times operation is to be looped between the LSA and the LEA when the `loop` instruction is executed. Refer to the LEA. In a repeat instruction, furthermore, it is used to store the repeat count or the number of times operation is to be repeated.

2.11 Arithmetic Operation Registers (ALR and AHR)

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

One of the special registers included in the C33 ADV Core CPU is the arithmetic operation register used in multiply/divide and multiply-accumulate 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. Arithmetic and multiply-accumulate 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. In divide operations, the quotient and remainder are stored in the ALR and AHR, respectively. When initialized upon reset, the ALR and AHR become indeterminate.

The results of multiply, divide, or multiply-accumulate operations that are written to ALR and AHR may be loaded into the R4 and R5 registers by setting the LC (bit 17) and HC (bit 16) flags in the PSR to 1.
When LC = 1, the operation result in ALR is loaded into R4.
When HC = 1, the operation result in AHR is loaded into R5.

Therefore, the result of a multiply/divide or multiply-accumulate operation can be referenced directly in a data transfer or arithmetic operation instruction.

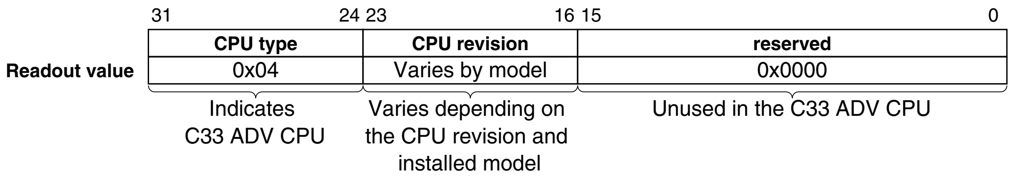
Note: This function just loads the multiply/divide or multiply-accumulate operation results to R4 and R5 along with ALR and AHR. ALR and AHR cannot be accessed through R4 and R5 even if the LC and HC flags are set to 1.

2.12 CPU Identification Register (IDIR)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
IDIR	CPU Identification Register	32 bits	R	R	0x04*XXXXXX*

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

The bit configuration in the IDIR is detailed below.


31
24 23
16 15
0
Readout value
Indicates C33 ADV CPU
Varies depending on the CPU revision and installed model
Unused in the C33 ADV CPU

	CPU type (31–24)	CPU revision (23–16)	reserved (15–0)
Readout value	0x04	Varies by model	0x0000

Figure 2.12.1 CPU Identification Register (IDIR)

2.13 Debug Base Register (DBBR)

Symbol	Register name	Size	Supervisor mode	User mode	Initial value
DBBR	Debug Base Register	32 bits	R	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 ADV CPU, is fixed to 0x00060000.

2.14 Register Notation and Register Numbers

The following describes the register notation and register numbers in the C33 ADV Core CPU 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 "`%`."

2.14.1 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 2.14.1.1 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`

2.14.2 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 2.14.2.1 Special Registers

Special register	Register number	Register notation
PSR	0	`%psr`
SP	1	`%sp`
ALR	2	`%alr`
AHR	3	`%ahr`
LCO *	4	`%lco`
LSA *	5	`%lsa`
LEA *	6	`%lea`
SOR *	7	`%sor`
TTBR *	8	`%ttbr`
DP *	9	`%dp`
IDIR *	10	`%idir`
DBBR *	11	`%dbbr`
–	(12)	-
USP *	13	`%usp`
SSP *	14	`%ssp`
PC	15	`%pc`

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

3 Data Formats

The C33 ADV Core CPU 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 CPU 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 high-order halfword or the 1 low-order byte of the source register.

Memory is accessed in little endian or big 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.

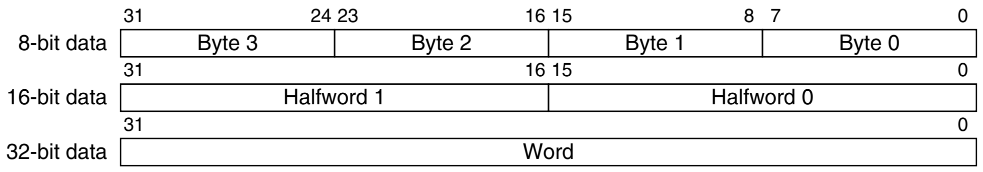


Figure 3.1 Little Endian Format

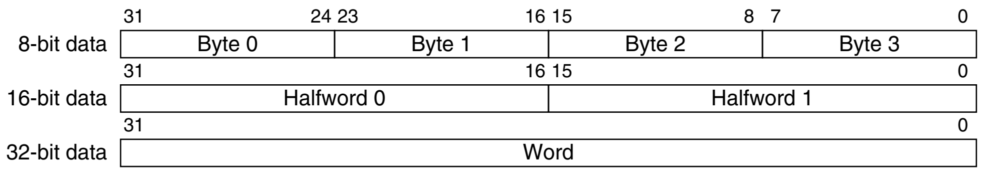


Figure 3.2 Big Endian Format

The data transfer sizes and types are described below.

3.1 Unsigned 8-Bit Transfer (Register → Register)

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

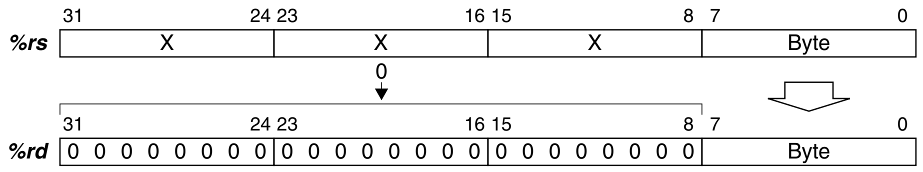


Figure 3.1.1 Unsigned 8-Bit Transfer (Register → Register)

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

3.2 Signed 8-Bit Transfer (Register → Register)

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

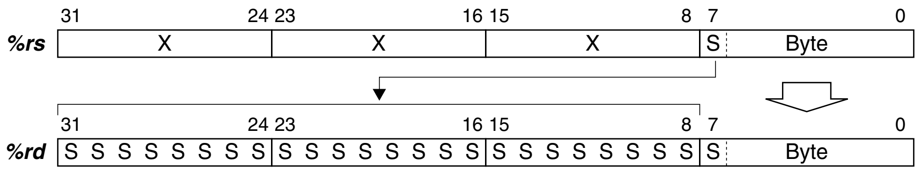


Figure 3.2.1 Signed 8-Bit Transfer (Register → Register)

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

3.3 Unsigned 8-Bit Transfer (Memory → Register)

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

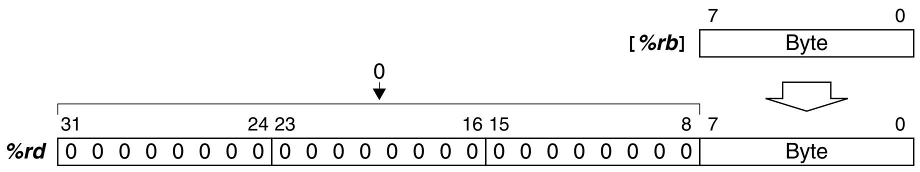


Figure 3.3.1 Unsigned 8-Bit Transfer (Memory → Register)

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

3.4 Signed 8-Bit Transfer (Memory → Register)

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

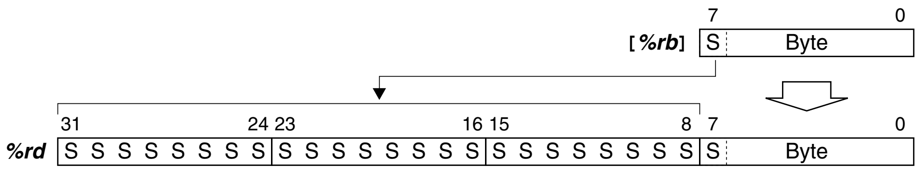


Figure 3.4.1 Signed 8-Bit Transfer (Memory → Register)

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

3.5 8-Bit Transfer (Register → Memory)

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

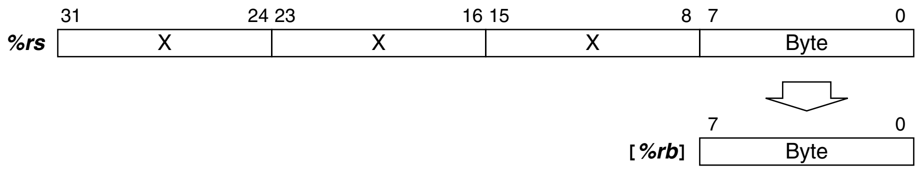


Figure 3.5.1 8-Bit Transfer (Register → Memory)

3.6 Unsigned 16-Bit Transfer (Register → Register)

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

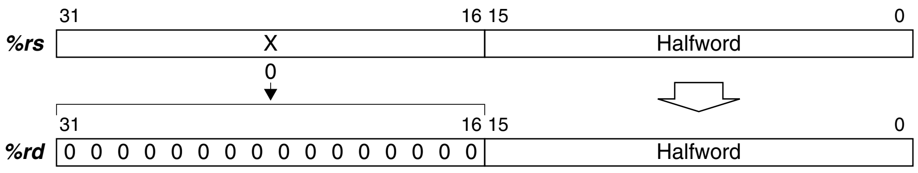


Figure 3.6.1 Unsigned 16-Bit Transfer (Register → Register)

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

3.7 Signed 16-Bit Transfer (Register → Register)

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

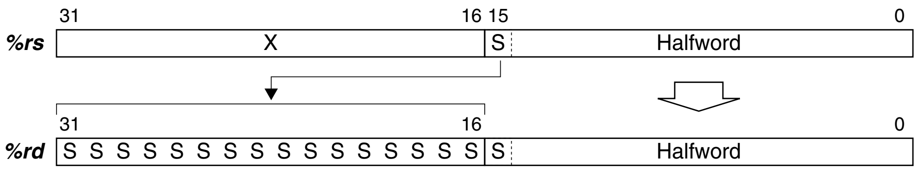


Figure 3.7.1 Signed 16-Bit Transfer (Register → Register)

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

3.8 Unsigned 16-Bit Transfer (Memory → Register)

Example: `ld.uh   %rd,[%rb]`   (For little endian)

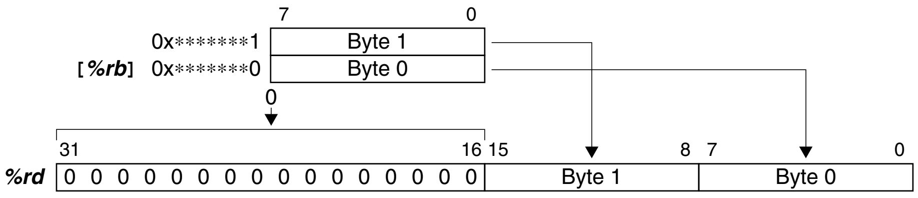


Figure 3.8.1 Unsigned 16-Bit Transfer (Memory → Register)

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

3.9 Signed 16-Bit Transfer (Memory → Register)

Example: `ld.h   %rd,[%rb]`   (For little endian)

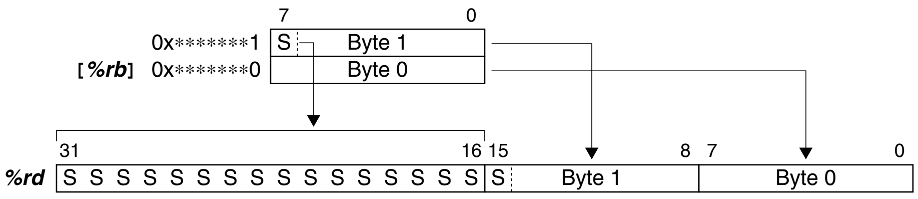


Figure 3.9.1 Signed 16-Bit Transfer (Memory → Register)

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

3.10 16-Bit Transfer (Register → Memory)

Example: `ld.h [%rb],%rs` (For little endian)

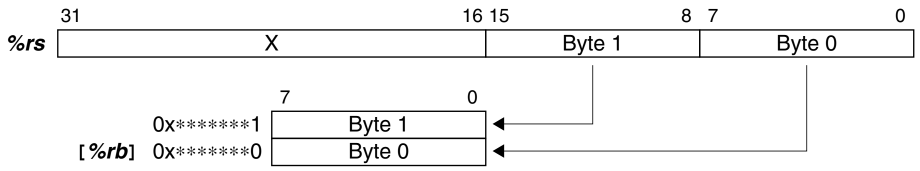


Figure 3.10.1 16-Bit Transfer (Register → Memory)

3.11 32-Bit Transfer (Register → Register)

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

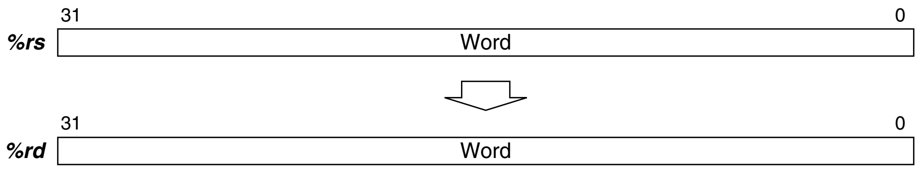


Figure 3.11.1 32-Bit Transfer (Register → Register)

3.12 32-Bit Transfer (Memory → Register)

Example: `ld.w %rd,[%rb]` (For little endian)

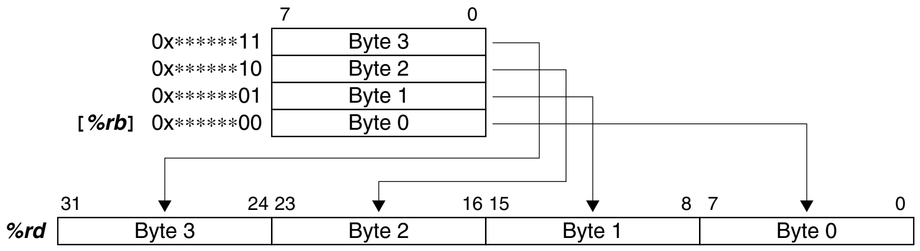


Figure 3.12.1 32-Bit Transfer (Memory → Register)

3.13 32-Bit Transfer (Register → Memory)

Example: `ld.w [%rb],%rs` (For little endian)

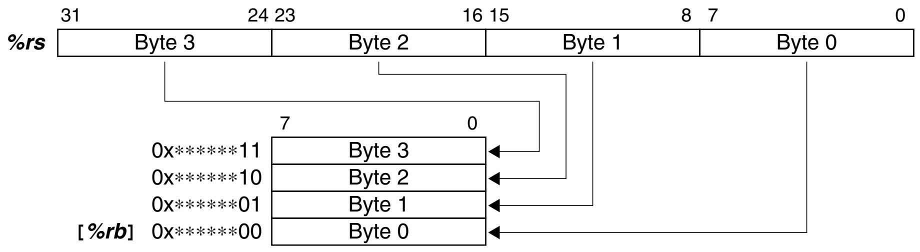


Figure 3.13.1 32-Bit Transfer (Register → Memory)

4 Address Map

The C33 ADV Core CPU supports a 4GB address space. Addresses output from the CPU can be translated into other predefined addresses by the HBCU and MMU, so the CPU does not need to output the addresses that are assigned to the actual ROM, RAM, and I/O devices. In other words, the CPU can access a 4GB linear logical address space (virtual space) that is able to build with free memory configuration. The HBCU translates the logical address output from the CPU into a physical address using the MMU and it sends the translated address to the BBCU, which manages the physical address space, to access the actual device. The 4GB physical address space is divided into 23 areas by the BBCU, and different memory or I/O devices can be mapped into each area.

The logical address output from the CPU is subjected to the five processes listed below by the HBCU, MMU, and BBCU to finally generate the physical address.
1. Block process
2. ASID process
3. Address translation process
4. Mirroring process
5. Area process

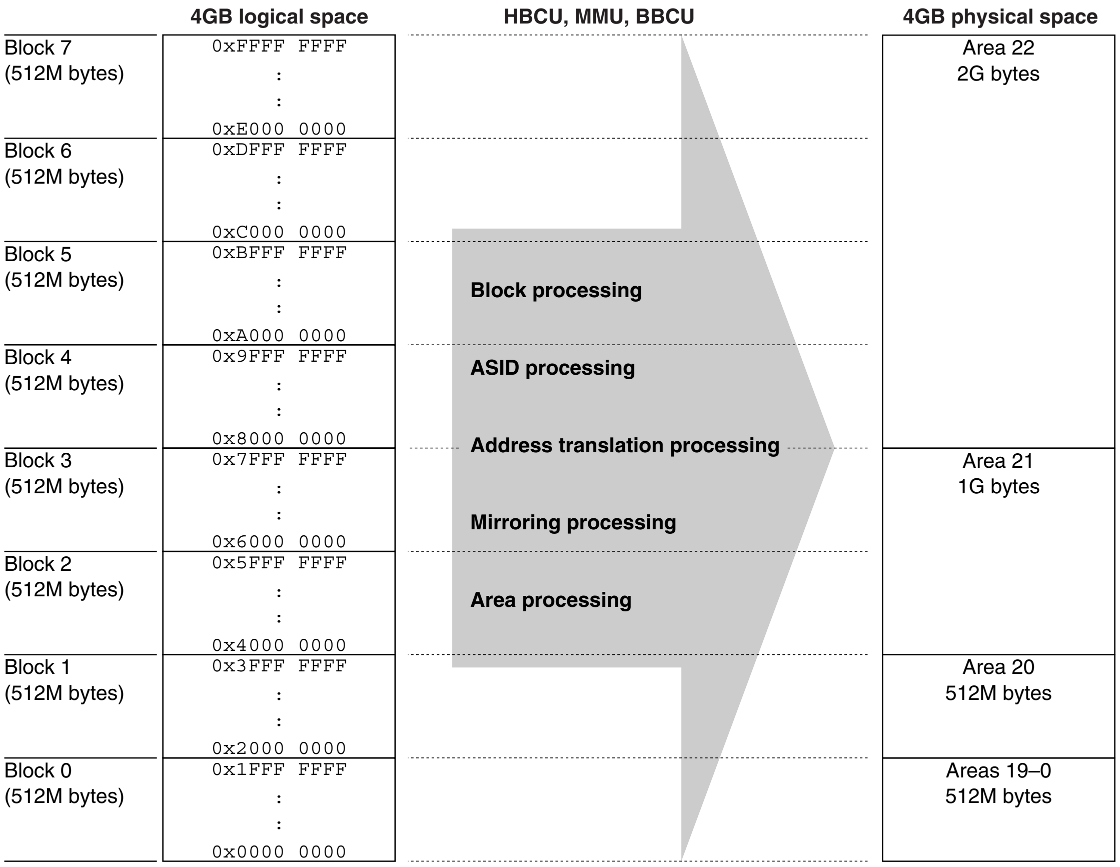


Figure 4.1 Relationship between the Logical and Physical Spaces

For details of the address processing, refer to the Technical Manual of each model.

Area 22	0xFFFF FFFF 0x8000 0000	External Memory	2G bytes
Area 21	0x7FFF FFFF 0x4000 0000	External Memory	1G bytes
Area 20	0x3FFF FFFF 0x2000 0000	External Memory	512M bytes
Area 19	0x1FFF FFFF 0x1000 0000	External Memory	256M bytes
Area 18	0x0FFF FFFF 0x0C00 0000	External Memory	64M bytes
Area 17	0x0BFF FFFF 0x0800 0000	External Memory	64M bytes
Area 16	0x07FF FFFF 0x0600 0000	External Memory	32M bytes
Area 15	0x05FF FFFF 0x0400 0000	External Memory	32M bytes
Area 14	0x03FF FFFF 0x0300 0000	External Memory	16M bytes
Area 13	0x02FF FFFF 0x0200 0000	External Memory	16M bytes
Area 12	0x01FF FFFF 0x0180 0000	External Memory	8M bytes
Area 11	0x017F FFFF 0x0100 0000	External Memory	8M bytes
Area 10	0x00FF FFFF 0x00C0 0000	External Memory	4M bytes
Area 9	0x00BF FFFF 0x0080 0000	External Memory	4M bytes
Area 8	0x007F FFFF 0x0060 0000	External Memory	2M bytes
Area 7	0x005F FFFF 0x0040 0000	External Memory	2M bytes
Area 6	0x003F FFFF 0x0030 0000	External Memory	1M bytes
Area 5	0x002F FFFF 0x0020 0000	External Memory	1M bytes
Area 4	0x001F FFFF 0x0010 0000	External Memory	1M bytes
Area 3	0x000F FFFF 0x0008 0000	Internal RAM	512K bytes
Area 2	0x0007 FFFF 0x0006 0000	For debugging	128K bytes
Area 1	0x0005 FFFF 0x0002 0000	Internal I/O	256K bytes
Area 0	0x0001 FFFF 0x0000 0000	Internal RAM	128K bytes

Figure 4.2 Physical Address Space

5 Instruction Set

The C33 ADV Core CPU instruction set consists of the function-extended instruction set of the C33 STD Core CPU and the new instructions useful for multimedia operation, in addition to the conventional S1C33-series instructions. As the C33 ADV Core CPU is upward object-code compatible with the C33 STD Core CPU, software assets can be transported from the S1C33 series to the C33 ADV 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)	
	`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	
Branch	`jrgt` `jrgt.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: !Z & !(N ^ V)
	`jrge` `jrge.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: !(N ^ V)
	`jrlt` `jrlt.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: N ^ V
	`jrle` `jrle.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: Z | N ^ V
	`jrugt` `jrugt.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: !Z & !C
	`jruge` `jruge.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: !C
	`jrult` `jrult.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: C
	`jrule` `jrule.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: Z | C
	`jreq` `jreq.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: Z
	`jrne` `jrne.d`	*sign8*	PC relative conditional jump Delayed branching possible	Branch condition: !Z
	`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

Classification	Mnemonic		Function
Branch	ret		Subroutine return
	ret.d		Delayed return possible
	reti		Return from interrupt or exception handling
	retd		Return from the debug processing routine
	int	*imm2*	Software exception
	brk		Debug exception
Data transfer	ld.b	%*rd*,%*rs*	General-purpose register (byte) → general-purpose register (sign-extended)
		%*rd*,[%*rb*]	Memory (byte) → general-purpose register (sign-extended)
		%*rd*,[%*rb*]+	Postincrement possible
		%*rd*,[%sp+*imm6*]	Stack (byte) → general-purpose register (sign-extended)
		[%*rb*],%*rs*	General-purpose register (byte) → memory
		[%*rb*]+,%*rs*	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*]	Memory (byte) → general-purpose register (zero-extended)
		%*rd*,[%*rb*]+	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*]	Memory (halfword) → general-purpose register (sign-extended)
		%*rd*,[%*rb*]+	Postincrement possible
		%*rd*,[%sp+*imm6*]	Stack (halfword) → general-purpose register (sign-extended)
		[%*rb*],%*rs*	General-purpose register (halfword) → memory
		[%*rb*]+,%*rs*	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*]	Memory (halfword) → general-purpose register (zero-extended)
		%*rd*,[%*rb*]+	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*]	Memory (word) → general-purpose register
		%*rd*,[%*rb*]+	Postincrement possible
		%*rd*,[%sp+*imm6*]	Stack (word) → general-purpose register
		[%*rb*],%*rs*	General-purpose register (word) → memory
		[%*rb*]+,%*rs*	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
	mirror	%*rd*,%*rs*	Bitwise swap every byte in word
	mac	%*rs*	Multiply-accumulate operation 16 bits × 16 bits + 64 bits → 64 bits
	pushn	%*rs*	Push general-purpose registers %*rs*–%r0 onto the stack
	popn	%*rd*	Pop data for general-purpose registers %*rd*–%r0 off the stack

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	Mode in which the V flag is cleared after instruction execution has been added.
		%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	
Other	`scan0`	*%rd,%rs*	Search for bits whose value = 0	The number of bits that can be scanned has been increased to 32 bits.
	`scan1`	*%rd,%rs*	Search for bits whose value = 1	

5.3 Instructions Added to the C33 ADV Core CPU

Table 5.3.1 Instructions Added to the C33 ADV Core CPU

Classification	Mnemonic		Function
Arithmetic operation	add	%*rd*,%dp	Addition of DP register
	mlt.hw	%*rd*,%*rs*	Signed integer multiplication 32 bits × 16 bits → 64 bits
	mac.hw	%*rs*	Multiply-accumulate operation 32 bits × 16 bits + 64 bits → 64 bits
	mac.w	%*rs*	Multiply-accumulate operation 32 bits × 32 bits + 64 bits → 64 bits
	mac1.h	%*rd*,%*rs*	Single multiply-accumulate operation 16 bits × 16 bits + 64 bits → 64 bits
	mac1.hw	%*rd*,%*rs*	Single multiply-accumulate operation 32 bits × 16 bits + 64 bits → 64 bits
	mac1.w	%*rd*,%*rs*	Single multiply-accumulate operation 32 bits × 32 bits + 64 bits → 64 bits
	div.w	%*rs*	Signed integer division 32 bits / 32 bits → 16 bits ... 16 bits
	divu.w	%*rs*	Unsigned integer division 32 bits / 32 bits → 16 bits ... 16 bits
Branch	jpr jpr.d	%*rb*	PC relative jump Delayed branching possible
	retm		Return from MMU exception handler routine
Data transfer	ld.b	%*rd*,[%dp+*imm6*]	Memory indirect (byte) → general-purpose register (sign-extended)
	ld.ub	%*rd*,[%dp+*imm6*]	Memory indirect (byte) → general-purpose register (zero-extended)
	ld.h	%*rd*,[%dp+*imm6*]	Memory indirect (halfword) → general-purpose register (sign-extended)
	ld.uh	%*rd*,[%dp+*imm6*]	Memory indirect (halfword) → general-purpose register (zero-extended)
	ld.w	%*rd*,[%dp+*imm6*]	Memory indirect (word) → general-purpose register
	ld.b	[%dp+*imm6*],%*rs*	General-purpose register (byte) → memory indirect (sign-extended)
	ld.h	[%dp+*imm6*],%*rs*	General-purpose register (halfword) → memory indirect (sign-extended)
	ld.w	[%dp+*imm6*],%*rs*	General-purpose register (word) → memory indirect
System control	psrset	*imm5*	Set a specified bit in PSR
	psrclr	*imm5*	Clear a specified bit in PSR
Multifunction extension	ext	%*rs*	Execute following instructions for 3 operands
	ext	*cond*	Conditional execution
	ext	*op*,*imm2*	Postshift
	ext	%*rs*,*op*,*imm2*	3 operands execution + postshift
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	macclr		Clear ALR and AHR registers and MO flag to 0
	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
	sat.b	%*rd*,%*rs*	Signed saturation processing of general-purpose register (byte)
	sat.ub	%*rd*,%*rs*	Unsigned saturation processing of general-purpose register (byte)
	sat.h	%*rd*,%*rs*	Signed saturation processing of general-purpose register (halfword)
	sat.uh	%*rd*,%*rs*	Unsigned saturation processing of general-purpose register (halfword)
	sat.w	%*rd*,%*rs*	Signed saturation processing of general-purpose register (word)
	sat.uw	%*rd*,%*rs*	Unsigned saturation processing of general-purpose register (word)
	loop	%*rc*,%*ra*	Execute specified range (general-purpose register) the specified number of times (general-purpose register)
		%*rc*,*imm4*	Execute specified range (immediate) the specified number of times (general-purpose register)
		imm4,*imm4*	Execute specified range (immediate) the specified number of times (immediate)
	repeat	%*rc*	Execute following instructions (as many times as specified by the general-purpose register)
		imm4	Execute following instructions (as many times as specified by the immediate)

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

Table 5.3.2 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
%*rc*	General-purpose register, loop count
%*ra*	General-purpose register, loop address
%sp	Stack pointer
%dp	Data 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)

5.4 Addressing Modes (without `ext` extension)

The instruction set of the C33 ADV Core CPU, as with the S1C33 series, has six discrete addressing modes, as described below. The CPU 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

5.4.1 Immediate Addressing

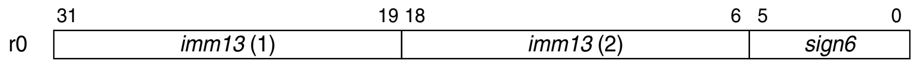
The immediate included in the instruction code that is indicated as *immX* (unsigned immediate) or *signX* (signed immediate) is used as the source data. The immediate size specifiable in each instruction is indicated by a numeral in the symbol (e.g., *imm4* = unsigned 4 bits; *sign6* = signed 6 bits). 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



5.4.2 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`
Loop count register	`%lco`
Loop start address register	`%lsa`
Loop end address register	`%lea`
Trap table base register	`%ttbr`
Data pointer	`%dp`
Shift-out register	`%sor`
User stack pointer	`%usp`
Supervisor stack pointer	`%ssp`

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

5.4.3 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 CPU 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.

5.4.4 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]+`.

5.4.5 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`*`]` or `[%dp+`*`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,[%dp+0x10]`

The halfword data at the address derived by adding 0x20 to the content of the current DP 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 the next section 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."
```

5.4.6 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
```

5.5 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. If three or more `ext` instructions are written successively, only the first and last `ext` instructions (those directly preceding the instruction for which the immediate is to be extended) are effective, and the `ext` instructions written in between have no effect.

Furthermore, there are several multifunction `ext` instructions that have been added to the C33 ADV Core CPU. These `ext` instructions will be detailed later.

5.5.1 Extension of Immediate Addressing

Extension of *imm6*

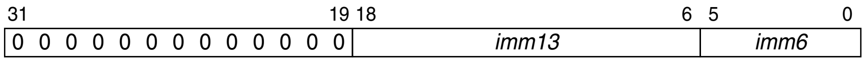
The *imm6* immediate is extended to a 19-bit or 32-bit immediate.

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



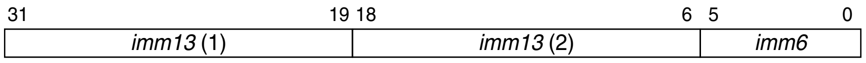
Bits 31–19 are filled with 0 (zero-extension).

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



Extension of *sign6*

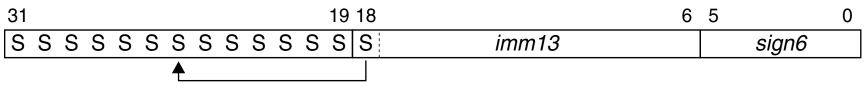
The *sign6* immediate is extended to a sign-extended 19-bit or 32-bit immediate.

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



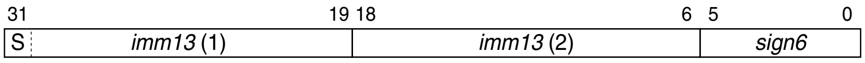
The most significant bit “S” in *imm13* that has been extended by the `ext` instruction is the sign, with which bits 31–19 are extended to become signed 19-bit data. The most significant bit in *sign6* is handled as the MSB data of 6-bit data, and not as the sign.

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



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

5.5.2 Extension of Register Indirect Addressing

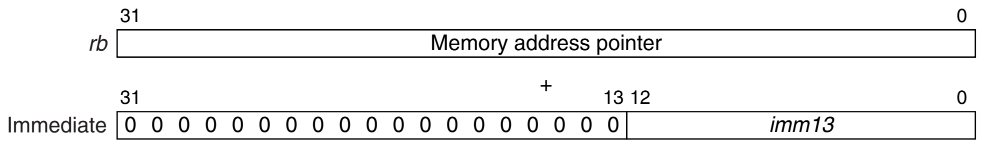
Adding displacement to `[%rb]`

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.

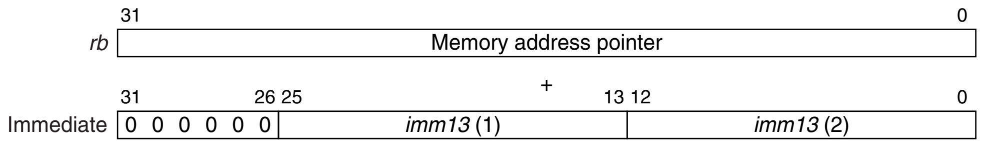
Example:
```
ext    imm13
ld.b   %rd,[%rb]
```



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]
```



Extending `[%sp+imm6]` or `[%dp+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,[%dp+imm6]`

The address referenced in this example is the "dp + *imm6* ∗ 2" address on a halfword boundary.

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

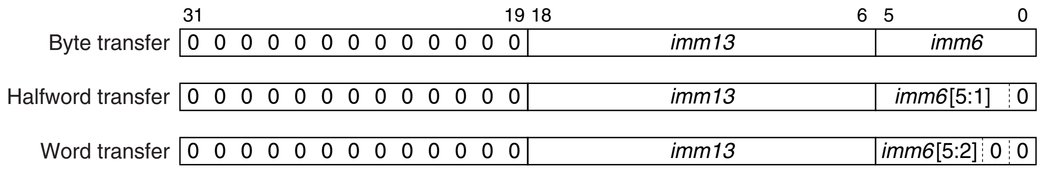
Extending to a 19-bit immediate

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



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

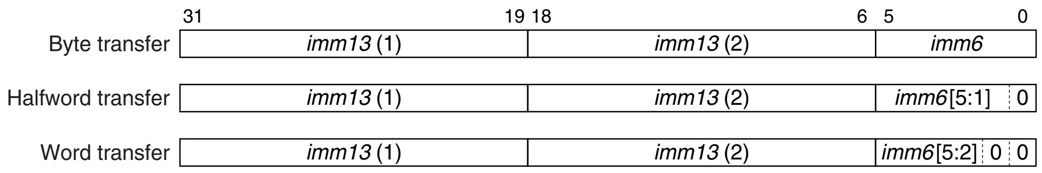
Extending to a 32-bit immediate

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



The extended data and the sp or dp 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.

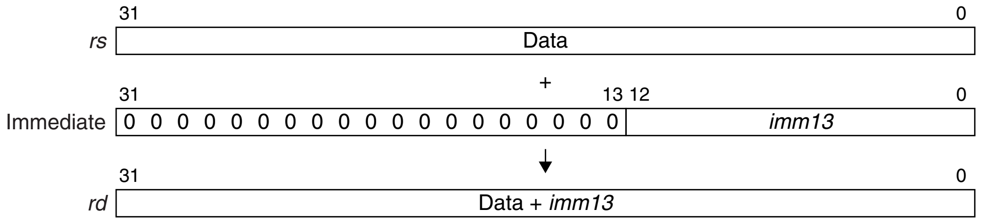
Extending to *rs + imm13*

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*



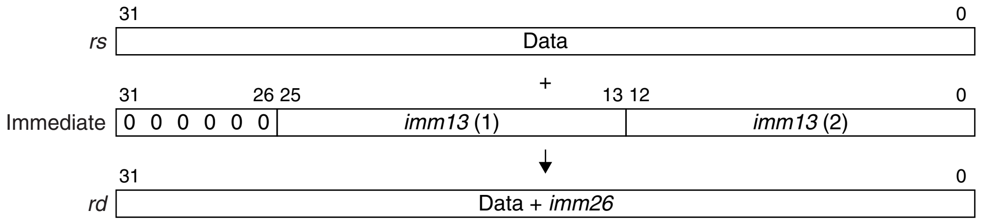
Extending to *rs + imm26*

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*



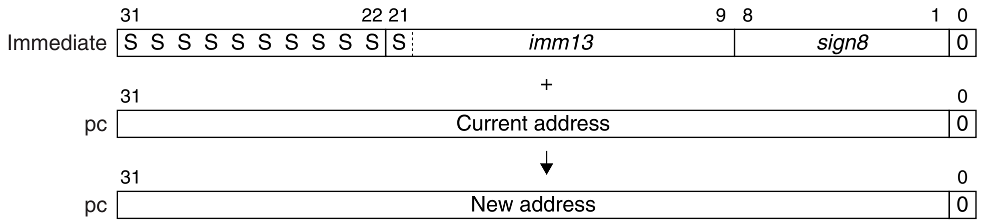
Extending the displacement of PC relative branch instructions

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
```

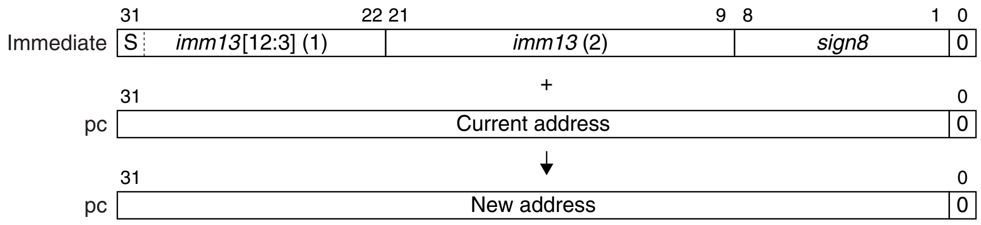


The most significant bit "S" in the immediate that has been extended by the `ext` instruction is the sign, with which bits 31–22 are extended to become signed 22-bit data. The most significant bit in *sign8* is handled as the MSB data of 8-bit data, and not as the sign.

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.

5.5.3 Register Indirect Addressing with Postincrement

If postincrement-added indirect addressing instructions have an `ext` instruction added in data transfer operations, the immediate value specified in the `ext` instruction is added to the register indicating the indirect address after a register-to-memory data transfer has finished. As immediate values are handled as a signed value, subtraction can also be performed.

One `ext` instruction can extend to signed 13-bit quantity, and two `ext` instructions can extend to signed 26-bit quantity. Furthermore, using a multifunction `ext` instruction can extend to signed 32-bit quantity.

```
Example 1: ext    0x100
           ld.w   [%r6]+,%r3          W[r6] ← r3, r6 ← r6 + 0x100

Example 2: ext    %r1
           ld.h   %r5,[%r4]+          r5 ← HW[r4], r4 ← r4 + r1
```

5.5.4 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`, `retm`, 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 multiple `ext` instructions, control returns to the first `ext`; if seven or more `ext` instructions are used simultaneously, control will not return to a location preceding seven or more `ext`s.)

Differences from the C33 STD Core CPU

In the C33 STD Core CPU, the exception handler routine, which is executed when an address misaligned exception occurs caused by an extended `ld.*` instruction with the `ext` instruction(s), returns to the `ld.*` instruction address, not to the `ext` instruction. Therefore, the C33 STD Core CPU does not execute the `ext` instruction(s) after the exception handling has completed.

In the C33 ADV Core CPU, the return address from the misaligned exception handler routine is the location where the `ext` instruction (the first `ext` if two or more `ext` instructions exist) has been stored, so the `ext` instruction(s) can be executed after the exception handling has completed.

5.6 Multifunction `ext` Instructions

The multifunction `ext` instructions include the following:

ext *%rs*	Operand extension
ext *%rs,op,imm2*	Immediate extension and postshift
ext *op,imm2*	Postshift
ext *cond*	Conditional instruction skip

5.6.1 `ext %rs`

This is a 3-operand instruction using three general-purpose registers in ALU or register indirect data transfer instructions. The `ext %rs` instruction is predecoded by the prefetch function, and the third operand, such as an ALU instruction that follows the `ext` instruction, is set. This `ext` instruction is effective for the following instructions.

```
ld.b    %rd,[%rb]          ld.b    [%rb],%rs
ld.b    %rd,[%rb]+         ld.b    [%rb]+,%rs
ld.ub   %rd,[%rb]
ld.ub   %rd,[%rb]+
ld.h    %rd,[%rb]          ld.h    [%rb],%rs
ld.h    %rd,[%rb]+         ld.h    [%rb]+,%rs
ld.uh   %rd,[%rb]
ld.uh   %rd,[%rb]+
ld.w    %rd,[%rb]          ld.w    [%rb],%rs
ld.w    %rd,[%rb]+         ld.w    [%rb]+,%rs
and     %rd,%rs            or      %rd,%rs            xor     %rd,%rs
add     %rd,%rs            adc     %rd,%rs            add     %rd,%dp
sub     %rd,%rs            sbc     %rd,%rs
srl     %rd,imm5           sll     %rd,imm5
srl     %rd,%rs            sll     %rd,%rs
sra     %rd,imm5           sla     %rd,imm5
sra     %rd,%rs            sla     %rd,%rs
rr      %rd,imm5           rl      %rd,imm5
rr      %rd,%rs            rl      %rd,%rs
```

The operation of each instruction when extended by `ext %rs` is listed in Table 5.6.1.1.

Table 5.6.1.1 Functionality of Instructions after Extension

Target instruction	Extension	Operation	
ld.b %*rd*,[%*rb1*]	ext %*rb2*	*rd* ← B[*rb1* + *rb2*]	(signed)
ld.b [%*rb1*],%*rs*	ext %*rb2*	B[*rb1* + *rb2*] ← *rs*	(signed)
ld.b %*rd*,[%*rb1*]+	ext %*rb2*	*rd* ← B[*rb1*]; *rb1* ← *rb1* + *rb2*	(signed)
ld.b [%*rb1*]+,%*rs*	ext %*rb2*	B[*rb1*] ← *rs*; *rb1* ← *rb1* + *rb2*	(signed)
lb.ub %*rd*,[%*rb1*]	ext %*rb2*	*rd* ← B[*rb1* + *rb2*]	(unsigned)
lb.ub %*rd*,[%*rb1*]+	ext %*rb2*	*rd* ← B[*rb1*]; *rb1* ← *rb1* + *rb2*	(unsigned)
ld.h %*rd*,[%*rb1*]	ext %*rb2*	*rd* ← H[*rb1* + *rb2*]	(signed)
ld.h [%*rb1*],%*rs*	ext %*rb2*	H[*rb1* + *rb2*] ← *rs*	(signed)
ld.h %*rd*,[%*rb1*]+	ext %*rb2*	*rd* ← H[*rb1*]; *rb1* ← *rb1* + *rb2*	(signed)
ld.h [%*rb1*]+,%*rs*	ext %*rb2*	H[*rb1*] ← *rs*; *rb1* ← *rb1* + *rb2*	(signed)
ld.uh %*rd*,[%*rb1*]	ext %*rb2*	*rd* ← H[*rb1* + *rb2*]	(unsigned)
ld.uh %*rd*,[%*rb1*]+	ext %*rb2*	*rd* ← H[*rb1*]; *rb1* ← *rb1* + *rb2*	(unsigned)
ld.w %*rd*,[%*rb1*]	ext %*rb2*	*rd* ← W[*rb1* + *rb2*]	
ld.w [%*rb1*],%*rs*	ext %*rb2*	W[*rb1* + *rb2*] ← *rs*	
ld.w %*rd*,[%*rb1*]+	ext %*rb2*	*rd* ← W[*rb1*]; *rb1* ← *rb1* + *rb2*	
ld.w [%*rb1*]+,%*rs*	ext %*rb2*	W[*rb1*] ← *rs*; *rb1* ← *rb1* + *rb2*	
and %*rd*,%*rs1*	ext %*rs2*	*rd* ← *rs1* & *rs2*	
or %*rd*,%*rs1*	ext %*rs2*	*rd* ←*rs1* | *rs2*	
xor %*rd*,%*rs1*	ext %*rs2*	*rd* ← *rs1* ^ *rs2*	
add %*rd*,%*rs1*	ext %*rs2*	*rd* ← *rs1* + *rs2*	
adc %*rd*,%*rs1*	ext %*rs2*	*rd* ← *rs1* + *rs2* + C	
add %*rd*,%dp	ext %*rs*	*rd* ← dp + *rs*	
sub %*rd*,%*rs1*	ext %*rs2*	*rd* ← *rs1* - *rs2*	
sbc %*rd*,%*rs1*	ext %*rs2*	*rd* ← *rs1* - *rs2* - C	
srl %*rd*,*imm5*	ext %*rs*	*rd* ← *rs* >> *imm5*; *rd*[31] ← 0	
srl %*rd*,%*rs2*	ext %*rs1*	*rd* ← *rs1* >> *rs2*; *rd*[31] ← 0	
sll %*rd*,*imm5*	ext %*rs*	*rd* ← *rs* << *imm5*; *rd*[0] ← 0	
sll %*rd*,%*rs2*	ext %*rs1*	*rd* ← *rs1* << *rs2*; *rd*[0] ← 0	
sra %*rd*,*imm5*	ext %*rs*	*rd* ← *rs* >> *imm5*; *rd*[31] ← *rs*[31]	
sra %*rd*,%*rs2*	ext %*rs1*	*rd* ← *rs1* >> *rs2*; *rd*[31] ← *rs1*[31]	
sla %*rd*,*imm5*	ext %*rs*	*rd* ← *rs* << *imm5*; *rd*[0] ← 0	
sla %*rd*,%*rs2*	ext %*rs1*	*rd* ← *rs1* << *rs2*; *rd*[0] ← 0	
rr %*rd*,*imm5*	ext %*rs*	*rd* ← *rs* >> *imm5*; *rd*[31] ← *rs*[0]	
rr %*rd*,%*rs2*	ext %*rs1*	*rd* ← *rs1* >> *rs2*; *rd*[31] ← *rs1*[0]	
rl %*rd*,*imm5*	ext %*rs*	*rd* ← *rs* << *imm5*; *rd*[0] ← *rs*[31]	
rl %*rd*,%*rs*	ext %*rs1*	*rd* ← *rs1* << *rs2*; *rd*[0] ← *rs1*[31]	

5.6.2 `ext %rs,op,imm2`

In addition to an operand extension by `%rs`, this instruction has a postshift function dictated in the `add` or `sub` instruction that follows the `ext` instruction. When postshift is specified in the operand, the result of operation performed by the `add`/`sub` instruction is shifted to the left or right.
This instruction can only be used in the instructions listed below. If used in any other instruction, the shift specification "*op*, *imm2*" is ignored. Therefore, this instruction functions as an `ext %rs` instruction.

Instruction group in which "`ext %rs,op,imm2`" functions effectively

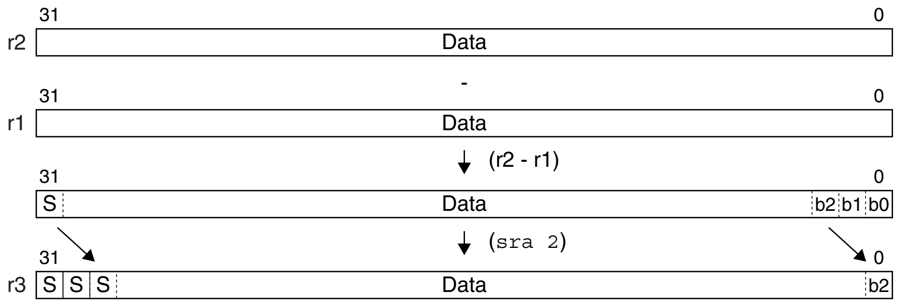
```
add  %rd,%rs      adc  %rd,%rs      add  %rd,%dp
sub  %rd,%rs      sbc  %rd,%rs
```

The postshift operation to be performed following instruction execution is determined by the shift operation instructed by *op* and the number of bits to be shifted specified by *imm2*. The instructions listed in Table 5.6.2.1 can be specified for *op*. The maximum number of bits that can be shifted is 3.

Table 5.6.2.1 Postshift Operation

op	*imm2*	Function
`sra`	0, 1, 2, 3	Arithmetic shift right >> *imm2*, *rd*[31] ← *rd*[31]
`srl`	0, 1, 2, 3	Logical shift right >> *imm2*, *rd*[31] ← 0
`sll`	0, 1, 2, 3	Logical shift left << *imm2*, *rd*[0] ← 0

```
Example: ext  %r1,sra,2
         sub  %r3,%r2        ; r3 ← (r2 - r1) >> 2, r3[31:30] = S
          :       :
```



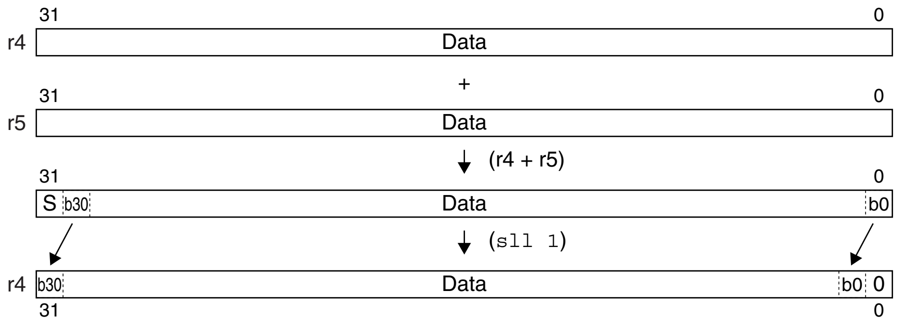
5.6.3 `ext op,imm2`

This instruction dictates a postshift function in the instruction that follows. The operation of this instruction is the same as that of `ext %rs,op,imm2` in Section 5.6.2, with `%rs` removed. This instruction can only be used in the instructions listed below.

Instruction group in which "`ext op,imm2`" functions effectively

```
add  %rd,%rs   add  %rd,imm6   add  %sp,imm10   adc  %rd,%rs   add  %rd,%dp
sub  %rd,%rs   sub  %rd,imm6   sub  %sp,imm10   sbc  %rd,%rs
```

```
Example: ext  sll,1
         add  %r4,%r5        ; r4 ← (r4 + r5) << 1, r4[0] = 0
          :       :
```



5.6.4 `ext` *`cond`*

This instruction dictates that the target instruction that follows it should not be executed, depending on the status of the condition code indicated in *cond* (C, V, Z, N flags in PSR[3:0]). However, be aware that the branch instructions listed below can not be placed directly after this instruction as the instruction has no effect.

Invalid target instructions

```
jrgt   sign8        jrgt.d  sign8
jrge   sign8        jrge.d  sign8
jrlt   sign8        jrlt.d  sign8
jrle   sign8        jrle.d  sign8
jrugt  sign8        jrugt.d sign8
jruge  sign8        jruge.d sign8
jrult  sign8        jrult.d sign8
jrule  sign8        jrule.d sign8
jreq   sign8        jreq.d  sign8
jrne   sign8        jrne.d  sign8
jp     sign8        jp.d    sign8
jp     %rb          jp.d    %rb
jpr    %rb          jpr.d   %rb
call   sign8        call.d  sign8
call   %rb          call.d  %rb
ret                 ret.d
reti                retd
retm                int     imm2
brk                 slp
halt                loop
repeat
```

The `ext` *`cond`* instruction supports the conditions listed below. The target instruction will not be executed if the condition code matches the relevant condition.

Table 5.6.4.1 Conditions

Instruction	Condition
`ext gt`	!Z & !(N ^ V)
`ext ge`	!(N ^ V)
`ext lt`	N ^ V
`ext le`	Z | (N ^ V)
`ext ugt`	!Z & !C
`ext uge`	!C
`ext ult`	C
`ext ule`	Z | C
`ext eq`	Z
`ext ne`	!Z

Example:
```
cmp    %r2,%r3
ext    eq
ld.w   %r4,%r5        Not executed if r2 = r3
```

5.6.5 Combination of `ext` Instructions

The `ext` instructions to extend the immediate and the multifunction `ext` instructions can be used in combination.

The following `ext` instructions can be combined:

```
1)  ext   imm13              Immediate extension 1
    (ext   imm13)            (Immediate extension 2)
    nextop

2)  ext   %rs                Operand extension
    nextop

3)  ext   op,imm2            Postshift
    nextop

4)  ext   %rs,op,imm2        Operand extension + postshift
    nextop

5)  ext   cond               Conditional execution
    nextop

6)  ext   cond               Conditional execution
    ext   imm13              Immediate extension 1
    (ext   imm13)            (Immediate extension 2)
    nextop

7)  ext   cond               Conditional execution
    ext   op,imm2
    nextop

8)  ext   cond               Conditional execution
    ext   op,imm2            Postshift
    ext   imm13              Immediate extension 1
    (ext   imm13)            (Immediate extension 2)
    nextop

9)  ext   cond               Conditional execution
    ext   %rs,op,imm2        Operand extension + postshift
    nextop

10) ext   cond               Conditional execution
    ext   %rs                Operand extension
    nextop

11) ext   op,imm2            Postshift
    ext   imm13              Immediate extension 1
    (ext   imm13)            (Immediate extension 2)
    nextop
```

```
Example: cmp   %r1,%r2         r1 and r2 are compared
         ext   ne              add instruction not executed when r1 != r2
         ext   %r4,sll,3       Operand extension + postshift
         add   %r5,%r6         r5 = (r6 + r4) << 3
```

Note: In the above example, the target to be dictated by the `ext ne` instruction when the condition for the condition code holds true is a non-`ext` instruction existing after a series of `ext` instructions (in this case, the `add` instruction). Be aware that `ext %r4,sll,3` is not the target instruction.

5.7 Data Transfer Instructions

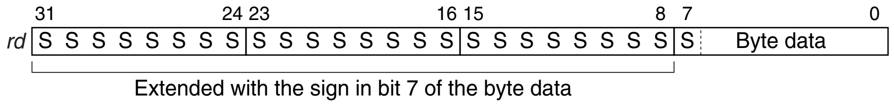
The transfer instructions in the C33 ADV Core CPU support data transfer between one register and another, as well as between a register and memory. A transfer data size and data extension format can be specified in the instruction code. In mnemonics, this specification is classified as follows:

`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

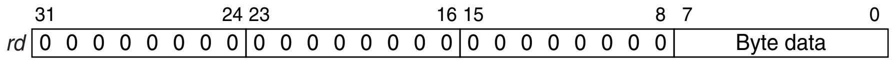
In signed byte or halfword transfers to registers, the source data is sign-extended to 32 bits. In unsigned byte or halfword transfers, the source data is zero-extended to 32 bits.
In transfers in which data is transferred from registers, data of a specified size on the lower side of the register is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

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

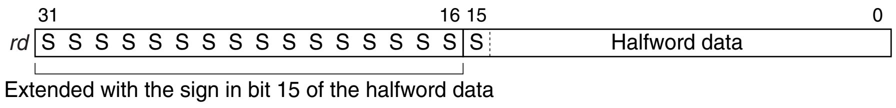
Signed byte data transfer


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

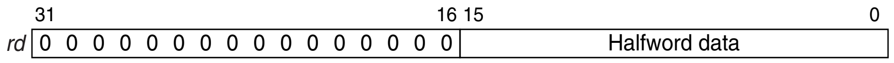
Unsigned byte data transfer



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



5.8 Logical Operation Instructions

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

`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

If a logical operation is performed when the OC flag (bit 21) in the PSR = 1, the V flag (bit 2) in the PSR is cleared.

5.9 Arithmetic Operation Instructions

The instruction set of the C33 ADV Core CPU supports add/subtract, compare, and multiply/divide instructions for arithmetic operations. (The multiply/divide 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 and Divide Instructions

The C33 ADV Core CPU comes standard with multiply/divide functions.

5.10.1 Multiplication Instructions

The instruction set of the C33 ADV Core CPU includes five 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)
`mlt.hw`	32 bits × 16 bits → 64 bits (sign-extended to 64 bits)

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 results of 32-bit × 32-bit and 32-bit × 16-bit operations are 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 ADV Core CPU executes 16-bit × 16-bit multiplication in one cycle and 32-bit × 32-bit multiplication in two cycles.

Differences from the C33 STD Core CPU

- An `mlt.hw` instruction has been added to the C33 ADV Core CPU. Furthermore, a `macclr` instruction has been added that clears the ALR and AHR registers collectively.
- The results of multiply operations are loaded into the R4 and R5 registers along with ALR and AHR, respectively, by setting the LC flag (bit 17) and HC flag (bit 16) in the PSR to 1.

5.10.2 Division Instructions

The C33 ADV Core CPU has signed and unsigned step division functions.

Instructions used for signed step division:	**`div0s`**, **`div1`**, **`div2s`**, **`div3s`**
Instructions used for unsigned step division:	**`div0u`**, **`div1`**

Furthermore, it also has signed batch division and unsigned batch division functions.

Signed batch division instruction:	**`div.w`**
Unsigned batch division instruction:	**`divu.w`**

The following describes the procedure for executing step division and the functionality of each instruction.

1. Preprocessing for step division (`div0s`, `div0u`)

Before starting division, prepare the dividend in the ALR and the divisor in the *rs* register (general-purpose registers R0–R15), and execute `div0s` or `div0u`. The operation of each instruction is described below.

`div0s` (preprocessing for signed step division)

- Extend the AHR with the sign in the ALR (dividend)
  The value set in the AHR is 0x00000000 if the dividend is positive, or 0xFFFFFFFF if the dividend is negative.
- Set the sign bit of the dividend in the DS flag of the PSR
  The DS flag is reset to 0 if the dividend is positive, or set to 1 if the dividend is negative.
- Set the sign bit of the divisor (*rs*) in the N flag of the PSR
  The N flag is reset to 0 if the divisor is positive, or set to 1 if the divisor is negative.

`div0u` (preprocessing for unsigned step division)

- Clear the AHR to 0x00000000
- Reset the DS flag in the PSR to 0
- Reset the N flag in the PSR to 0

* If the dividend is less than 32 bits, it should be filled toward the high-order side before being stored in the ALR, so that the number of times `div1` is executed and the number of bits will match. This helps to reduce the number of cycles in which division is performed. (This method cannot be employed for the batch division described later.)

Example: If the dividend is 0x55, store 0x55000000 in the ALR and execute `div1` eight times, or store 0x00550000 in the ALR and execute `div1` 16 times, or store 0x00000055 in the ALR and execute `div1` 32 times. The calculation result is the same for all.

2. Execution of step division

Execute the `div1` instruction a number of times as needed. For a division of 32 bits / 32 bits, for example, execute the `div1` instruction 32 times. (The number of times the instruction is executed can be adjusted by the number of bits in the dividend, as described in 1 above.) The `div1` instruction operates the same way in both signed and unsigned divisions.

The following processing is performed when the `div1` instruction is executed once.

(1) 64 bits in {AHR, ALR} are shifted one bit to the left (high-order side)   (ALR[0] = 0)

(2) The AHR and *rs* are added or the *rs* is subtracted from the AHR, with the AHR and ALR newly set again depending on the result.
Addition or subtraction is performed on the content of the AHR with the DS flag added as the sign bit (total of 33 bits) and the content of the *rs* register with the N flag added as the sign bit (total of 33 bits). This processing differs depending on the DS and N flags in the PSR, as described below. Note that the value of the 33rd bit in the operation result is referred to as tmp[32] in the explanation below.

When DS = 0 (dividend: positive) and N = 0 (divisor: positive)
- tmp = {0, AHR} - {0, *rs*} is executed
- If tmp[32] = 1, operation finishes with AHR = tmp[31:0] and ALR[0] = 1
  If tmp[32] = 0, operation finishes with AHR and ALR left intact

When DS = 1 (dividend: negative) and N = 0 (divisor: positive)
- tmp = {1, AHR} + {0, *rs*} is executed
- If tmp[32] = 0, operation finishes with AHR = tmp[31:0] and ALR[0] = 1
  If tmp[32] = 1, operation finishes with AHR and ALR left intact

When DS = 0 (dividend: positive) and N = 1 (divisor: negative)
- tmp = {0, AHR} + {1, *rs*} is executed
- If tmp[32] = 1, operation finishes with AHR = tmp[31:0] and ALR[0] = 1
  If tmp[32] = 0, operation finishes with AHR and ALR left intact

When DS = 1 (dividend: negative) and N = 1 (divisor: negative)
- tmp = {1, AHR} - {1, *rs*} is executed
- If tmp[32] = 0, operation finishes with AHR = tmp[31:0] and ALR[0] = 1
  If tmp[32] = 1, operation finishes with AHR and ALR left intact

For unsigned division, the result is obtained from the registers specified below by executing the `div1` instruction a number of times as needed.

AHR = Remainder
ALR = Quotient

For signed division, the correction described below is required.

3. Correction for signed division

For signed division, execute the `div2s` and `div3s` instructions successively after executing the `div1` instruction a number of times as needed, and then correct the result of operation.
For unsigned division, there is no need to execute the `div2s` and `div3s` instructions. If these instructions are executed nevertheless, they function in the same way as the `nop` instruction, without affecting the result of operation.
The functionality of the `div2s` and `div3s` instructions is described below.

`div2s` (correction of the result of signed step division, 1)

If, when the dividend is a negative number, the result of operation in a dividing step (executing the `div1` instruction) becomes 0; it is then possible that the result of operation after all steps have been completed will have a remainder (AHR) that is the same as the divisor, and a quotient (ALR) of a magnitude 1 less than the actual value. The `div2s` instruction corrects the result of operation for such inaccuracies.
The operation of the `div2s` instruction is described below.

When DS = 0 (dividend: positive)
When the dividend is positive, the above problem does not occur. Therefore, the `div2s` instruction terminates without any action being taken (as in the case of the `nop` instruction).

When DS = 1 (dividend: negative)
(1) If N = 0 (divisor: positive), tmp = AHR + *rs* is executed.
If N = 1 (divisor: negative), tmp = AHR - *rs* is executed.
(2) Depending on the result of operation (1)
If tmp is zero, operation finishes with AHR = tmp[31:0] and ALR = ALR + 1.
If tmp is not zero, operation finishes with AHR and ALR left intact.

`div3s` (correction of the result of signed step division, 2)

The quotient obtained from the ALR as a result of step division is always a positive number. If the dividend and divisor have different signs, the result of operation must be negative. The `div3s` instruction corrects the sign in such a case.

When DS = N (dividend and divisor have the same sign)
In this case, the above problem does not occur. Therefore, the `div3s` instruction terminates without any action being taken (as in the case of the `nop` instruction).

When DS ≠ N (dividend and divisor have different signs)
The sign of the ALR (quotient) is inverted.

Following execution of the `div2s` and `div3s` instructions, the final result of a signed division is obtained from the registers specified below.
AHR = Remainder
ALR = Quotient

The C33 ADV Core CPU includes new instructions to perform this series of step divisions collectively.
Batch-division instructions come in two types: one for signed division and one for unsigned division. For either type, a 32-bit ÷ 32-bit operation is executed by one instruction. The input and output registers used in these batch-division instructions are the same as those used for step division, with the dividend and divisor loaded into the ALR and *rs*, respectively, before being executed. The result of a division is obtained from the ALR for the quotient, and from the AHR for the remainder.

Differences from the C33 STD Core CPU

Signed and unsigned batch-division instructions have been added.

`div.w`	Signed batch-division instruction
`divu.w`	Unsigned batch-division instruction

Example execution of step division

(1) Execution of signed 32 bits ÷ 32 bits
(When the dividend and divisor are loaded into R0 and R1, respectively)

```
ld.w    %alr,%r0     ; Set the dividend in the ALR
div0s   %r1          ; Initialization step
div1    %r1          ; Step division
  :       :
div1    %r1          ; Execute the div1 instruction 32 times
div2s   %r1          ; Correction instruction 1
div3s                ; Correction instruction 2
```

The remainder and quotient are loaded into the AHR and ALR, respectively.
The time required for execution of this example is 35 cycles.
For signed divisions, the remainder in the result of division assumes the same sign as the dividend.
Example: (-8) ÷ 5 = -1 Remainder -3
8 ÷ (-5) = -1 Remainder 3

(2) Execution of unsigned 32 bits ÷ 32 bits
(When the dividend and divisor are loaded into R0 and R1, respectively)

```
ld.w    %alr,%r0     ; Set the dividend in the ALR
div0u   %r1          ; Initialization step
div1    %r1          ; Step division
  :       :
div1    %r1          ; Execute the div1 instruction 32 times
```

The remainder and quotient are loaded into the AHR and ALR, respectively.
The time required for execution of this example is 33 cycles.

Example execution of batch division

(1) Execution of signed 32 bits ÷ 32 bits
(When the dividend and divisor are loaded into R0 and R1, respectively)

```
ld.w    %alr,%r0     ; Set the dividend in the ALR
div.w   %r1          ; Execute signed batch division
```

The remainder and quotient are loaded into the AHR and ALR, respectively.
The time required for execution of this example is 35 cycles.

(2) Execution of unsigned 32 bits ÷ 32 bits
(When the dividend and divisor are loaded into R0 and R1, respectively)

```
ld.w    %alr,%r0     ; Set the dividend in the ALR
divu.w %r1           ; Execute unsigned batch division
```

The remainder and quotient are loaded into the AHR and ALR, respectively.
The time required for execution of this example is 35 cycles.

Differences from the C33 STD Core CPU

The results of divide operations are loaded into the R4 and R5 registers along with ALR and AHR, respectively, by setting the LC flag (bit 17) and HC flag (bit 16) in the PSR to 1.

5.11 Multiply-accumulate Operation Instructions

The C33 ADV Core CPU supports a multiply-accumulate operation function that executes an operation such as those listed below a specified number of times.

`mac`	***`%rs`***	16 bits × 16 bits + 64 bits → 64 bits
`mac.hw`	***`%rs`***	32 bits × 16 bits + 64 bits → 64 bits
`mac.w`	***`%rs`***	32 bits × 32 bits + 64 bits → 64 bits

This function helps implement digital signal processing on chip with no need to add a dedicated DSP external to the chip. Multiply-accumulate operations are executed using the `mac`, `mac.hw`, or `mac.w` instruction.

The `mac` *`%rs`* instruction executes the operation
(H[<*rs* + 1>]+) × (H[<*rs* + 2>]+) + {AHR, ALR} → {AHR, ALR} (H: halfword)

The `mac.hw` *`%rs`* instruction executes the operation
(W[<*rs* + 1>]+) × (H[<*rs* + 2>]+) + {AHR, ALR} → {AHR, ALR} (W: word)

The `mac.w` *`%rs`* instruction executes the operation
(W[<*rs* + 1>]+) × (W[<*rs* + 2>]+) + {AHR, ALR} → {AHR, ALR}

a number of times as specified in the *rs* register.

The *rs* register must have set in it the number of times a multiply-accumulate operation must be executed before the operation can be started.
The *rs* register is used as a counter to count the number of times operation is executed. The counter is decremented each time operation is executed. When the *rs* register reaches 0, the multiply-accumulate operation is terminated. Therefore, a multiply-accumulate operation can be performed up to 2^{32} - 1 times (4,294,967,295 times). However, if the value 0 is set in the *rs* register before a multiply-accumulate operation is executed, no operation will be performed. Nor will the AHR and ALR be altered. The *rs* register remains 0, and is not decremented.

In the above expressions, <*rs*+1> and <*rs*+2> denote two general-purpose registers following the *rs* register.
Example: If the R0 register is specified for *rs*, then
<*rs*+1> = R1 register and <*rs*+2> = R2 register.

If the R15 register is specified for *rs*, then
<*rs*+1> = R0 register and <*rs*+2> = R1 register.

In the above expressions, H[<*rs*+1>]+ and H[<*rs*+2>]+ denote the halfword data in memory with base addresses specified based on the contents of the above registers.
In a multiply-accumulate operation, this data is multiplied as signed 16-bit data, and the result is added to the {AHR, ALR} register pair. The sign "+" indicates that the base addresses of the respective data (contents of the <*rs*+1> and <*rs*+2> registers) are incremented (by 2 for halfword, or 4 for word) each time a multiply-accumulate operation is performed.
Example: Set R0 = 16, R1 = 0x100, R2 = 0x120, AHR = ALR = 0 and then execute `mac %r0`
1) {AHR, ALR} = 0 + H[0x100] × H[0x120]
2) {AHR, ALR} = {AHR, ALR} + H[0x102] × H[0x122]
3) {AHR, ALR} = {AHR, ALR} + H[0x104] × H[0x124]
:
16) {AHR, ALR} = {AHR, ALR} + H[0x11E] × H[0x13E]

The result of operation is obtained as signed 64-bit data, with the 32 high-order bits stored in the AHR and the 32 low-order bits stored in the ALR. The register values become R0 = 0, R1 = 0x120, R2 = 0x140.

In the above expressions, W[<*rs*+1>]+ denotes the base address of word data. The data to be loaded is handled as signed 32-bit data, and the address is incremented by 4 each time data is loaded.

Overflow during multiply-accumulate operation

If the result of operation exceeds the range of values representable by signed 64 bits during a multiply-accumulate operation, an overflow is assumed and the MO flag in the PSR is set to 1. Even in this case, the operation is continued until the count set in the *rs* register reaches 0.
The MO flag remains set until it is reset in the software. By reading out the MO flag following execution of the `mac` instruction, it is possible to check whether the result of operation is valid.

Interrupts during multiply-accumulate operation

Interrupts requested during execution of a `mac` instruction are accepted, even during repeat operations. When the program branches to the interrupt handler routine, the address of the `mac` instruction being executed is saved to the stack as the return address. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the `mac` instruction whose execution has been suspended is resumed. However, as the content of the *rs* register at that point in time indicates the remaining count, if the content of the *rs* register is altered during the interrupt handler routine, operation will not be executed the exact number of times initially set. Similarly, if the <*rs*+1> and <*rs*+2> register values change during the interrupt handler routine, the resumed `mac` instruction will not be executed correctly.

Differences from the C33 STD Core CPU

- The `mac.hw` and `mac.w` instructions have been added.

`mac.hw`	`%rs`	32 bits × 16 bits + 64 bits → 64 bits
`mac.w`	`%rs`	32 bits × 32 bits + 64 bits → 64 bits

- The results of multiply-accumulate operations are loaded into the R4 and R5 registers along with ALR and AHR, respectively, by setting the LC flag (bit 17) and HC flag (bit 16) in the PSR to 1.

5.12 Single Multiply-accumulate Operation Instructions

In addition to the instructions to perform a series of multiply-accumulate operations successively (`mac`, `mac.hw`, or `mac.w` instruction), the C33 ADV Core CPU supports instructions to perform a multiply-accumulate operation on the data loaded into a general-purpose register singly only once.

`mac1.h`	***`%rd,%rs`***	16 bits × 16 bits + 64 bits → 64 bits
`mac1.hw`	***`%rd,%rs`***	32 bits × 16 bits + 64 bits → 64 bits
`mac1.w`	***`%rd,%rs`***	32 bits × 32 bits + 64 bits → 64 bits

The result of operation is obtained as signed 64-bit data, with the 32 high-order bits stored in the AHR and the 32 low-order bits stored in the ALR. The AHR and ALR must be initialized before a multiply-accumulate operation is performed. Execute the `macclr` instruction to initialize the AHR and ALR and the MO flag in the PSR to 0. These instructions may be executed repeatedly any number of times to obtain the same result as possible with the `mac` instructions described above. However, when combined with other operation instructions, they can be used to perform more complicated calculations.

Differences from the C33 STD Core CPU

- The single multiply-accumulate operation instructions are supported beginning with the C33 ADV Core CPU.
- The results of single multiply-accumulate operations are loaded into the R4 and R5 registers along with ALR and AHR, respectively, by setting the LC flag (bit 17) and HC flag (bit 16) in the PSR to 1.

5.13 Shift and Rotate Instructions

The instruction set of the C33 ADV Core CPU 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. Furthermore, a special register named SOR is provided for storing the bits that have been shifted out from a general-purpose register by a shift or rotate instruction. 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.

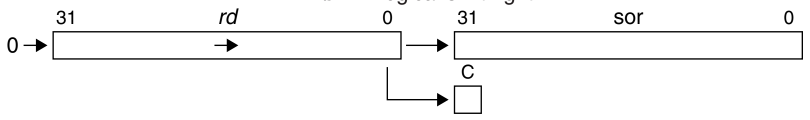
Example: `srl %rd,imm5` Bits 0–31 logically shifted to the right
`srl %rd,%rs` Bits 0–31 logically shifted to the right

In addition, setting the SE (bit 20) in the PSR to 1 causes the C (carry) and V (overflow) flags to change state. The C (carry) flag contains the bit that was last shifted out from the LSB or MSB. The V (overflow) flag changes state depending on the C (carry) and N (negative) flags upon completion of a shift operation, as shown below.

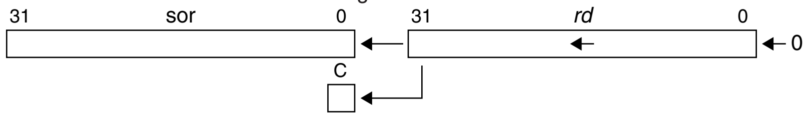
Table 5.13.1 Changes in the V Flag

C flag	N flag	V flag
0	0	0
1	1	0
0	1	1
1	0	1

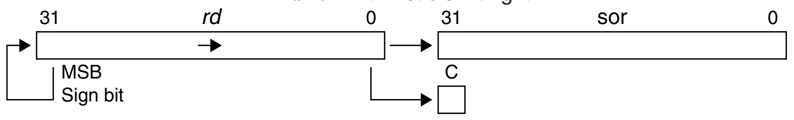
`srl` Logical shift right



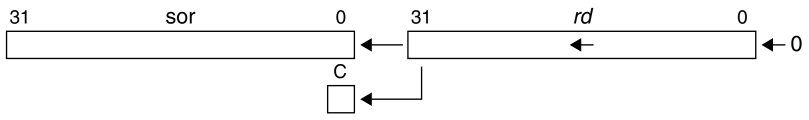
`sll` Logical shift left

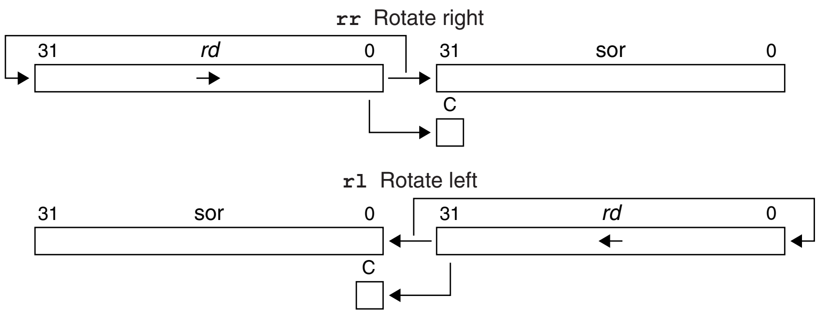


`sra` Arithmetic shift right



`sla` Arithmetic shift left





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

Table 5.13.2 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.

5.14 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.

5.15 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 a range of special registers from the one specified by *ss* to ALR to the stack successively. However, if the register specified by *ss* is the PSR or SP, the register is saved to the stack singly.

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 to *sd* successively. However, if the register specified by *sd* is the PSR or SP, the register is restored from the stack singly.

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. While successive push or pop instructions are being executed, the PM flag (bit 28) in the PSR remains set to 1. If the `pushn`, `pushs`, `popn`, or `pops` instruction is executed when the PM flag (bit 28) = 1, successive push or successive pop operations are performed beginning with the register whose register number is stored in RC[3:0] (bits 27–24) of the PSR.

If the USP, SSP, or PC register is specified in the `pushs` instruction, memory write and SP decrement operations are performed even for special register number #12 for which register is actually nonexistent, in the same way as for other registers. In this case, the data written to memory is indeterminate. If the USP, SSP, or PC register is specified in the `pops` instruction, memory read and SP increment operations are performed even for special register number #12 for which register is actually nonexistent, in the same way as for other registers. In this case, the data read from memory is not reflected in any register.

Exceptions can be made even when a successive push or successive pop operation is performed repeatedly by a `pushn`, `pushs`, `popn`, or `pops` instruction. When an exception is made during execution of one of these instructions, the address of the instruction is saved to the stack as a return address. Furthermore, the register number on which a push or pop operation is being executed is saved to the RC[3:0] bits in the PSR before exception handling is performed. At this time, the PM flag (bit 28) in the PSR is cleared to 0 as the program jumps to the vector address for the exception handler routine. <u>For the MMU and debug exceptions, however, the PSR is not saved, nor is the PM flag (bit 28) cleared to 0. Therefore, be sure to save the PSR and then clear its PM flag (bit 28) to 0 in the respective exception handler routines.</u>

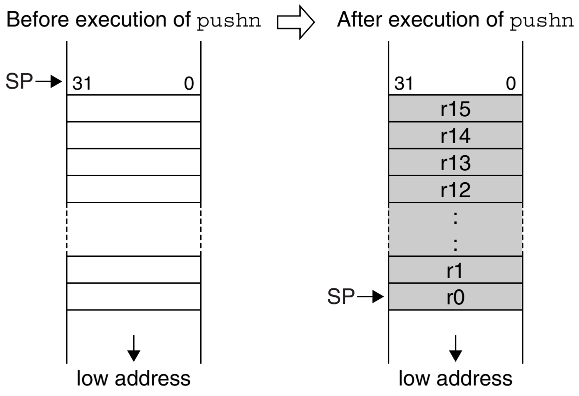
When returned from exception handling by the `reti`, `retm`, or `retd` instruction, the program returns to the `pushn`, `pushs`, `popn`, or `pops` instruction whose execution has been suspended. In this case, however, if the PM flag (bit 28) in the PSR = 1 when instruction execution is resumed, a push/pop operation is restarted from the register whose register number is stored in the RC[3:0] bits of the PSR. The register number present in the register field of the instruction word is not referenced.

If the PM flag (bit 28) in the PSR = 0 when the `pushn`, `pushs`, `popn`, or `pops` instruction is executed, the register number included in the register field of the instruction word is referenced; if the PM flag (bit 28) = 1, the register number saved in the RC[3:0] bits of the PSR is referenced. For the MMU and debug exceptions, the PSR is retained intact when exception handling is performed. Therefore, be aware that if registers are saved using a `pushn` or other instruction in the exception handler routine, the RC[3:0] bits of the PSR will be referenced, making correct instruction execution impossible. (The processing underlined above must be performed.)

Differences from the C33 STD Core CPU

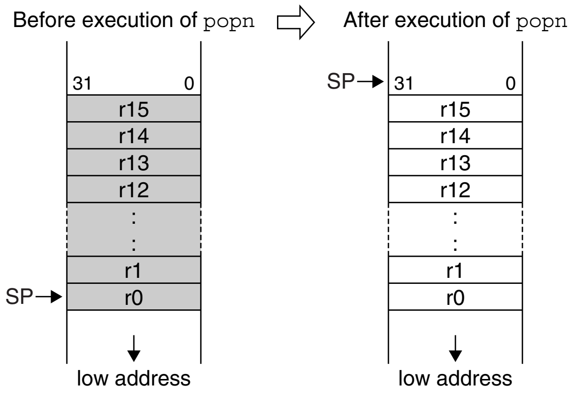
- General-purpose-register single push/pop instructions have been added.
  `push   %`*`rs`*   `pop   %`*`rd`*
- Special-register successive push/pop instructions have been added.
  `pushs  %`*`ss`*   `pops  %`*`sd`*

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 5.15.1 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 5.15.2 Successive Pop of General-Purpose Registers

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

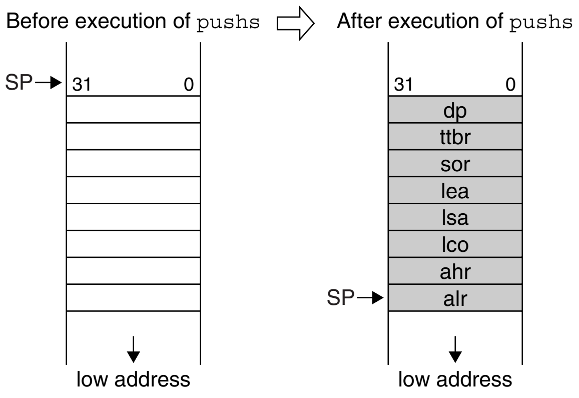


Figure 5.15.3 Successive Push of Special Registers

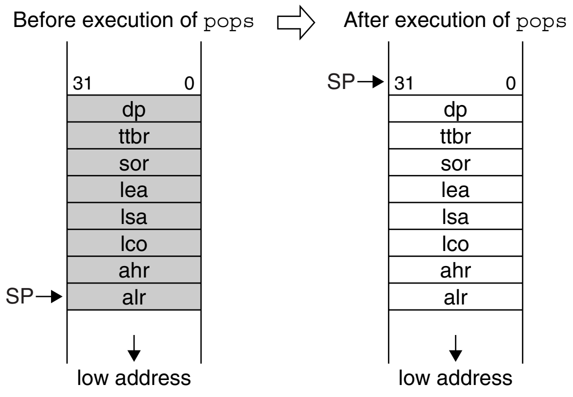


Figure 5.15.4 Successive Pop of Special Registers

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

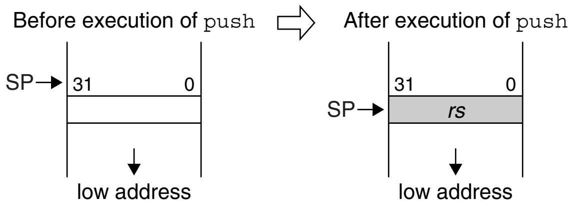


Figure 5.15.5 Single Push of a General-Purpose Register

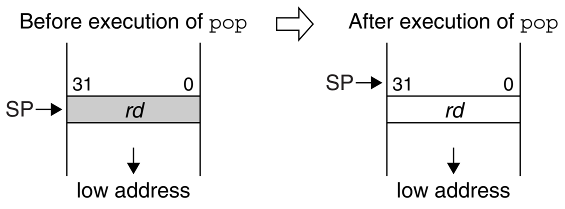


Figure 5.15.6 Single Pop of a General-Purpose Register

5.16 Branch and Delayed Branch Instructions

5.16.1 Types of Branch Instructions

(1) PC relative jump instructions

PC relative jump instructions include the following:

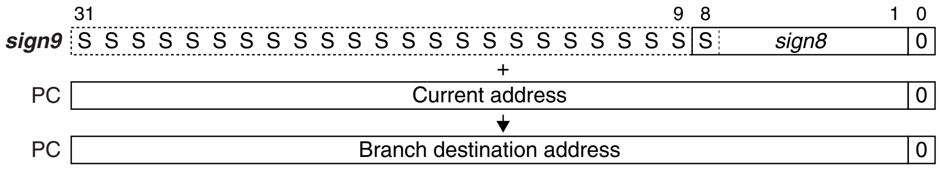
```
jr*   sign8
jp    sign8
jpr   %rb
```

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 ADV Core CPU 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.

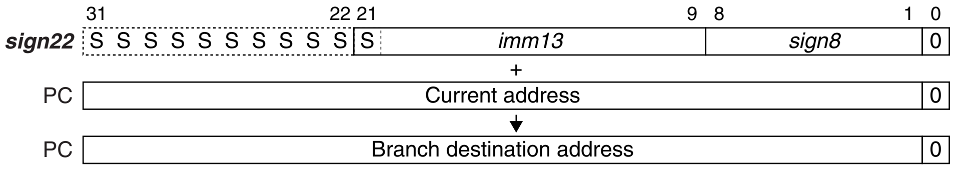

31
9
8
1
0
sign9
S S S S S S S S S S S S S S S S S S S S S S S
S
sign8
0
+
PC
Current address
0
PC
Branch destination address
0

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*.



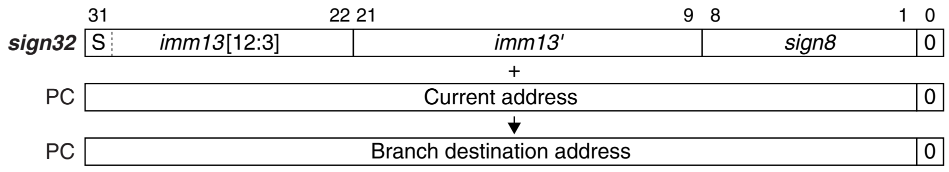
The range of addresses to which jumped is (PC - 2,097,152) to (PC + 2,097,150).

When extended by two `ext` instructions

```
ext  imm13
ext  imm13'
jp   sign8
```

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}

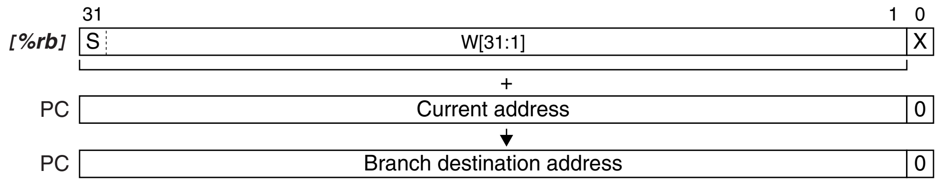


The range of addresses to which jumped is (PC - 2,147,483,648) to (PC + 2,147,483,646).
The above range of addresses to which jumped is a theoretical value, and is actually limited by the range of memory areas used.

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 5.16.1.1.

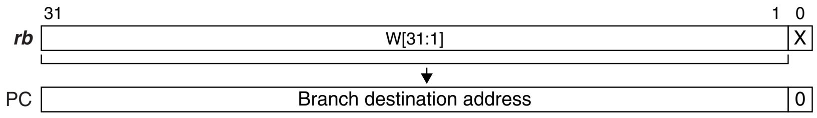
Table 5.16.1.1 Conditional Jump Instructions and Branch Conditions

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 CPU 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 Mode."

Differences from the C33 STD Core CPU

Register indirect relative branch instructions have been added.

```
    jpr  %rb
```

5.16.2 Delayed Branch Instructions

The C33 ADV Core CPU 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
ld.w    %sd,%rs         ld.w    %rd,%ss
add     %rd,%rs         add     %rd,imm6        add     %sp,imm10
add     %rd,%dp
adc     %rd,%rs
sub     %rd,%rs         sub     %rd,imm6        sub     %sp,imm10
sbc     %rd,%rs
mlt.h   %rd,%rs
mltu.h  %rd,%rs
cmp     %rd,%rs         cmp     %rd,sign6
and     %rd,%rs         and     %rd,sign6
or      %rd,%rs         or      %rd,sign6
xor     %rd,%rs         xor     %rd,sign6
not     %rd,%rs         not     %rd,sign6
srl     %rd,%rs         srl     %rd,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
scan0   %rd,%rs
scan1   %rd,%rs
swap    %rd,%rs         swaph   %rd,%rs
mirror  %rd,%rs
macclr
sat.b   %rd,%rs         sat.ub  %rd,%rs
sat.h   %rd,%rs         sat.uh  %rd,%rs
sat.w   %rd,%rs         sat.uw  %rd,%rs
psrclr  imm5
psrset  imm5
ld.c    %rd,imm4
ld.c    imm4,%rs
ld.cf
```

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.

5.17 Scan Instructions

A scan instruction scans the bits in a specified general-purpose register beginning with the MSB, and returns the bit position of the first 0 or 1 found.
The scan instructions in the C33 ADV Core CPU are functionally divided between 8-bit mode and 32-bit mode by the SW flag (bit 22) in the PSR.

8-bit scan mode when PSR[22] (SW flag) = 0
32-bit scan mode when PSR[22] (SW flag) = 1

`scan0 %rd,%rs`

The *rs* register is scanned, and the bit position of the first 0 found (offset from the MSB) is loaded into the *rd* register. When the SW flag in the PSR = 0, only 8 bits from the MSB are scanned, and the result is stored in the *rd* register. If 0s is not found in any bit, the value "0x00000008" in 8-bit scan mode or "0x00000020" in 32-bit scan mode is loaded into the *rd* register, and the C flag is set.

Table 5.17.1 Function of `scan0`

***rs* register** **MSB**	***rd* register**	**Flag**			
		C	**V**	**Z**	**N**
0	0x00000000	0	0	1	0
10	0x00000001	0	0	0	0
110	0x00000002	0	0	0	0
1110	0x00000003	0	0	0	0
1111 0	0x00000004	0	0	0	0
1111 10	0x00000005	0	0	0	0
1111 110	0x00000006	0	0	0	0
1111 1110	0x00000007	0	0	0	0
1111 1111 0	0x00000008	1*	0	0	0
1111 1111 10	0x00000009	0	0	0	0
:	:	:	:	:	:
1111 1111 1111 1111 1111 1111 1111 110	0x0000001e	0	0	0	0
1111 1111 1111 1111 1111 1111 1111 1110	0x0000001f	0	0	0	0
1111 1111 1111 1111 1111 1111 1111 1111	0x00000020	1	0	0	0

* The C flag is set when no 0s is found in the 8 high-order bits in 8-bit scan mode.

`scan1 %rd,%rs`

The *rs* register is scanned, and the bit position of the first 1 found (offset from the MSB) is loaded into the *rd* register. When the SW flag in the PSR = 0, only 8 bits from the MSB are scanned, and the result is stored in the *rd* register. If 1s is not found in any bit, the value "0x00000008" in 8-bit scan mode or "0x00000020" in 32-bit scan mode is loaded into the rd register, and the C flag is set.

Table 5.17.2 Function of `scan1`

***rs* register** **MSB**	***rd* register**	**Flag**			
		C	**V**	**Z**	**N**
1	0x00000000	0	0	1	0
01	0x00000001	0	0	0	0
001	0x00000002	0	0	0	0
0001	0x00000003	0	0	0	0
0000 1	0x00000004	0	0	0	0
0000 01	0x00000005	0	0	0	0
0000 001	0x00000006	0	0	0	0
0000 0001	0x00000007	0	0	0	0
0000 0000 1	0x00000008	1*	0	0	0
0000 0000 01	0x00000009	0	0	0	0
:	:	:	:	:	:
0000 0000 0000 0000 0000 0000 0000 001	0x0000001e	0	0	0	0
0000 0000 0000 0000 0000 0000 0000 0001	0x0000001f	0	0	0	0
0000 0000 0000 0000 0000 0000 0000 0000	0x00000020	1	0	0	0

* The C flag is set when no 1s is found in the 8 high-order bits in 8-bit scan mode.

Differences from the C33 STD Core CPU

The SW flag (bit 22) in the PSR is used for scan-mode discrimination.
SW = 0: 8-bit scan mode, SW = 1: 32-bit scan mode

5.18 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 CPU in HALT mode
`slp`	Places the CPU in SLEEP mode

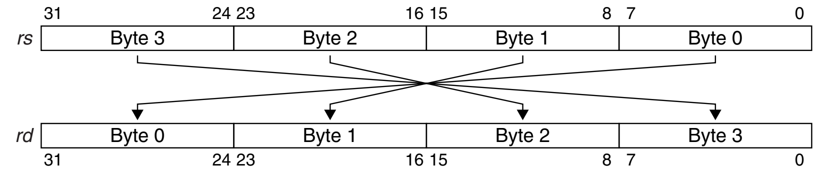
For details on HALT and SLEEP modes, refer to Section 6.4, “Power-Down Mode,” and the Clock Management Unit (CMU) section in the Technical Manual for each S1C33 model.

5.19 Swap and Mirror Instructions

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

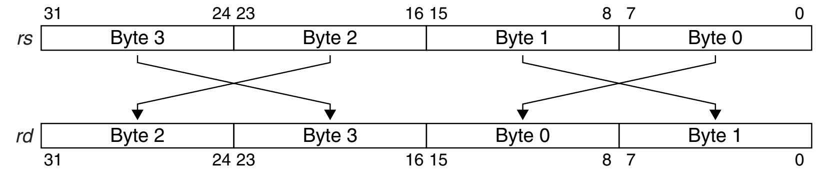
Swap instruction: `swap %rd,%rs`

Big and little endians are converted on a word boundary.



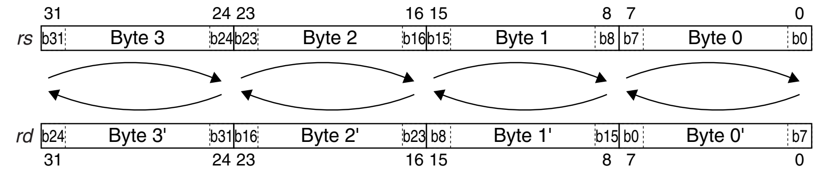
Swap instruction: `swaph %rd,%rs`

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



Mirror instruction: `mirror %rd,%rs`

The 32-bit data in general-purpose registers has its high-order and low-order bits replaced with each other in byte units.

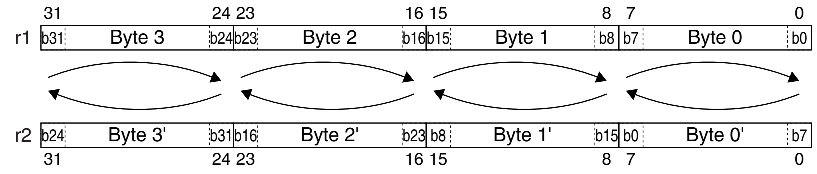


Application for 32-bit mirroring

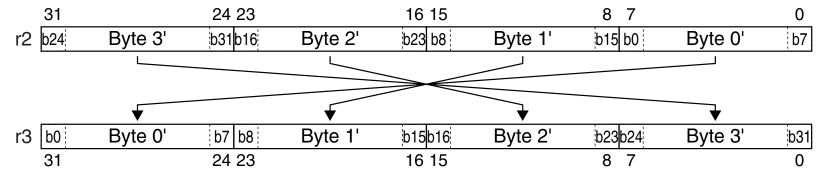
The `mirror` and `swap` instructions can be used in combination to achieve 32-bit mirroring.

```
Example: mirror  %r2,%r1  (1)
         swap    %r3,%r2  (2)
```

(1) Execute the `mirror` instruction



(2) Execute the `swap` instruction



Differences from the C33 STD Core CPU

The `swaph` instruction has been added.

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

5.20 Saturation Instructions

The C33 ADV Core CPU has data transfer instructions with a saturation function available. During data transfer, saturation processing is applied according to the data in the source register as the data is loaded into the destination register. When saturation processing occurs, the S flag (bit 15) in the PSR is set to 1. The available saturation instructions are as follows:

`sat.b`	***`%rd,%rs`***	Signed saturation instruction (byte transfer)
`sat.ub`	***`%rd,%rs`***	Unsigned saturation instruction (byte transfer)
`sat.h`	***`%rd,%rs`***	Signed saturation instruction (halfword transfer)
`sat.uh`	***`%rd,%rs`***	Unsigned saturation instruction (halfword transfer)
`sat.w`	***`%rd,%rs`***	Signed saturation instruction (word transfer)
`sat.uw`	***`%rd,%rs`***	Unsigned saturation instruction (word transfer)

Signed saturation instructions

When a byte or halfword size is specified, saturation processing occurs if the data in the source register exceeds the following range of values:

For byte specification	+127 to -128
For halfword specification	+32,767 to -32,768

When saturation processing occurs, the content of the source register is raised to the maximum value representable by a specified data size and sign-extended to 32 bits as it is loaded into the destination register. When saturation processing does not occur, the content of the source register is transferred to the destination register directly as is.

For a byte-size signed saturation instruction (`sat.b`), the following processing is applied:

rs > +127 → +127 (0x0000007F)
rs < -127 → -128 (0xFFFFFF80)

```
Example:   rs            rd
        0x00000012 → 0x00000012
        0x12345678 → 0x0000007F
        0xFFFFFFFA → 0xFFFFFFFA
        0xFFFFFABC → 0xFFFFFF80
```

For a halfword signed saturation instruction (`sat.h`), the following processing is applied:

rs > +32,767 → +32,767 (0x00007FFF)
rs < -32,768 → -32,768 (0xFFFF8000)

```
Example:   rs            rd
        0x00001234 → 0x00001234
        0x12345678 → 0x00007FFF
        0xFFFFABCD → 0xFFFFABCD
        0xFFABCDEF → 0xFFFF8000
```

If the saturation instruction is specified to be a word size (`sat.w`), the N and V flags in the PSR are checked, depending on which saturation processing is applied.
Saturation processing occurs when the following conditions are satisfied:

When N = 1 and V = 1
Maximum positive value “0x7FFFFFFF” → *rd*

When N = 0 and V = 1
Maximum negative value “0x80000000” → *rd*

Otherwise, saturation processing will not occur, and as a result the content of the source register will be transferred to the destination register directly as is.

Unsigned saturation instructions

When a byte or halfword size is specified, saturation processing occurs if the data in the source register exceeds the following values:

For byte specification	255
For halfword specification	65,535

When saturation processing occurs, the content of the source register is raised to the maximum value representable by a specified data size as it is loaded into the destination register and bits 31–8 are 0-extended. When saturation processing does not occur, the content of the source register is transferred to the destination register directly as is.

For a byte-size unsigned saturation instruction (`sat.ub`), the following processing is applied:
rs > 255 → 255 (0x000000FF)

Example: *rs* *rd*

```
0x00000012 → 0x00000012
0x12345678 → 0x000000FF
0xFFFFFFFA → 0x000000FF
0xFFFFFABC → 0x000000FF
```

For a halfword unsigned saturation instruction (`sat.uh`), the following processing is applied:
rs > 65,535 → 65,535 (0x0000FFFF)

Example: *rs* *rd*

```
0x00001234 → 0x00001234
0x12345678 → 0x0000FFFF
0xFFFFABCD → 0x0000FFFF
0xFFABCDEF → 0x0000FFFF
```

If the saturation instruction is specified to be a word size (`sat.uw`), the C flag in the PSR is checked, depending on which saturation processing is applied.
Saturation processing occurs when the following conditions are satisfied:

When C = 1
0xFFFFFFFF → *rd*

Otherwise, saturation processing will not occur, and as a result the content of the source register will be transferred to the destination register directly as is.

In either case, when saturation processing occurs, the S flag (bit 15) in the PSR is set to 1. This flag remains set until it is cleared in the software.

Table 5.20.1 Conditions for Saturation Processing

Instruction	Condition	Operation	S flag
`sat.b`	*rb* > +127	0x0000007F → *rd*	1
	rb < -128	0xFFFFFF80 → *rd*	1
	+127 ≥ *rb* ≥ -128	*rs* → *rd*	–
`sat.h`	*rb* > +32,767	0x00007FFF → *rd*	1
	rb < -32,768	0xFFFF8000 → *rd*	1
	+32,767 ≥ *rb* ≥ -32,768	*rs* → *rd*	–
`sat.w`	N = 1 & V = 1	0x7FFFFFFF → *rd*	1
	N = 0 & V = 1	0x80000000 → *rd*	1
	Other	*rs* → *rd*	–
`sat.ub`	|*rb*| > 255	0x000000FF → *rd*	1
	|*rb*| ≤ 255	*rs* → *rd*	–
`sat.uh`	|*rb*| > 65,535	0x0000FFFF → *rd*	1
	|*rb*| ≤ 65,535	*rs* → *rd*	–
`sat.uw`	C = 1	0xFFFFFFFF → *rd*	1
	Other	*rs* → *rd*	–

Example 1: Arithmetic/logic instruction and saturation processing

```
add    %r1,%r2
sat.b  %r1,%r1       ; Processed to a value in the range of -128 to +127
  :         :
sub    %r4,0x3
sat.uw %r5,%r4       ; Processed to a value in the range of 0 to 65,535
```

Example 2: Shift instruction and saturation processing

The V flag is effective for saturation shift to the left. (PSR[20] = 1 → V flag changes)

```
// input  : %r12 shift num, %r13 shift data
// output : %r10 result
SL_SAT:
     loop   %r12,END
     sll    %r13,1
     sat.w  %r13,%r13
END:
     ret.d
     ld.w   %r10,%r13
```

The C flag is effective for saturation shift to the right. (PSR[20] = 1 → C flag changes)

```
// input  : %r12 shift num, %r13 shift data
// output : %r10 result
SR_SAT:
     loop   %r12,END
     sra    %r13,1
     sat.w  %r13,%r13
END:
     ret.d
     ld.w   %r10,%r13
```

5.21 Repeat Instructions

5.21.1 Settings

A repeat operation means executing an instruction with its address set in the LSA register a number of times equal to the count set in the LCO register plus 1, when the two conditions specified below are satisfied. (The instruction is executed at least once, even when the value set in the LCO register is zero.)

(1) The RM flag (bit 30) in the PSR is set.

(2) The address set in the LSA register and the current PC match.

Each time the target instruction is executed, the LCO register is decremented by one until it reaches zero, at which time the repeat operation finishes and the RM flag (bit 30) is cleared to 0.
When a repeat instruction is executed, the PC for the next instruction is loaded into the LSA register, a repeat count is loaded into the LCO register, and the RM flag (bit 30) in the PSR is set to 1.
Furthermore, a repeat operation can be performed by setting each of the LSA, LCO, and RM flag (bit 30) individually. (This method is referred to as a "reserved repeat.") When a reserved repeat is performed, the following precautions should be observed:

(1) The RM flag (bit 30) must be set at the end.

(2) There must be one or more instructions before the LSA address after the RM flag (bit 30) is set. (However, if an `ext` instruction is included immediately after the RM flag (bit 30) is set, the extended part must be excluded from the instruction count.)

Given below are the precautions common to both repeat and reserved repeat instructions:

(1) The set values of the LSA and LCO cannot be altered after the RM flag (bit 30) is set.

(2) In no case may an "unrepeatable" instruction (described later) be placed immediately after a repeat instruction or at the LSA address.

Repeat instructions are superior to branch instructions in that there is no need to refetch the instruction at the jump address. For branch instructions, except for delayed branching, all of the prefetched instructions are discarded when the branch conditions are satisfied, and the instruction at the jump address must be fetched again. This is unnecessary for the repeat operation. In addition, reserved repeat allows a set of instructions including `ext` to be executed repeatedly. However, because repeat instructions are 4-clock instructions, even a reserved repeat requires one clock cycle each for setting of the LCO and LSA, and at least four clock cycles to set the PSR, in effect making repeat instructions 6-clock instructions. If the repeat counts are extremely small (e.g., repeated only three or four times), the increase in processing speed resulting from the above advantage may be canceled out.

Repeat instruction format

`repeat` ***`%rc`***	Executes the next instruction that follows a maximum of 4G (0xFFFFFFFF) + 1 times
`repeat` ***`imm4`***	Executes the next instruction that follows a maximum of 15 + 1 times

Example:
```
ld.w     %r1,0x0
repeat   9              ; Repeats instruction execution
ld.w     [%r2]+,%r1     ; Executes 10 times
```

The content of a 10-word (40-byte) area beginning with the R2 address is cleared to 0.

5.21.2 Break from a Repeat Operation

A repeat operation can be exited, if so desired, by generating a `brk` interrupt and clearing the RM flag (bit 30) in that interrupt.

5.21.3 Prohibition of Repeat Operation in Debug and MMU Exceptions

Make sure no repeat operations will be performed in debug exception and MMU exception handler routines.

5.21.4 Exception Handling during Repeat

Repeat instructions accept an exception that occurs during repeat. When an exception is accepted, the CPU saves the address of the repeated target instruction to the stack as the return address, and then saves the PSR to the stack. Furthermore, the CPU clears the RM flag (bit 30) in the PSR to 0 and jumps to the vector address for the exception handler routine. For MMU and debug exceptions, however, the PSR is not saved, nor is the RM flag (bit 30) cleared to 0. Therefore, the PSR and the flag must be protected in the respective exception handler routines. The remaining repeat count is left in the LCO register. In addition, the LCO and LSA registers used in a repeat instruction are used when the CPU returns from the exception handling.
If, when returning from the exception handler routine using the `reti`, `retm`, or `retd` instruction, the RM flag in the PSR = 1 and the PC and LSA register values match, a repeat instruction is resumed. With the remaining repeat count indicated by the LCO, the target instruction to be repeated is executed from where it left off.
Caution must be used for exception handler routines to ensure that the values of the special registers LSA and LCO will not be altered. If the values of the LSA or LCO are altered, the repeat operation cannot be resumed correctly after returning from the exception handler routine. If a `repeat` or `loop` instruction is used in an exception handler routine, the LSA and LCO values must be saved before the exception handler routine is started. Note, however, that loop/repeat operations cannot be performed in the debug exception and MMU exception handler routines.

5.21.5 Use of Multiple Loop/Repeats and Interrupts

If, while servicing an interrupt that occurred during a repeat, it is desired to use another repeat or loop within the interrupt handling, follow the execution procedure described below. However, do not use a loop/repeat in debug or MMU exceptions.

(1) Save the LSA and LCO to registers or memory.

(1') If a reserved loop/repeat is to be used, clear the RM (bit 30) here.

(2) Execute the loop/repeat or the reserved loop/repeat instruction.

(3) When the loop/repeat has finished, load the saved LSA and LCO from the registers or memory.

(4) Set the RM (bit 30).

To use a loop/repeat in an interrupt singly (i.e., not multiple loop/repeats), basically the same procedure as described above may be followed. However, because the RM flag (bit 30) is cleared to 0 except for MMU and debug exceptions, the operation specified in (1') is unnecessary. Furthermore, because the `reti` instruction includes loading of the PSR, the operation for setting the RM flag (bit 30) may also be omitted. Conversely, forcibly clearing the stacked RM flag (bit 30) has no effect, as the `reti` instruction for interrupts in a repeat always returns the RM flag (bit 30) set.

5.21.6 Unrepeatable Instructions

The instructions specified below cannot be used as the target instruction to be repeated. If these instructions are repeated, the program operation cannot be guaranteed.

```
nop
slp
halt
pushn   %rs          pushs   %ss
popn    %rd          pops    %sd
brk
ret                  ret.d
retd                 reti                 retm
int     imm2
ext     imm13        ext     %rs          ext     cond
ext     op,imm2      ext     %rs,op,imm2
mac     %rs          mac.hw  %rs          mac.w   %rs
div.w   %rs          divu.w  %rs
repeat  imm4         repeat  %rb
jrgt    sign8        jrgt.d  sign8
jrge    sign8        jrge.d  sign8
jrlt    sign8        jrlt.d  sign8
jrle    sign8        jrle.d  sign8
jrugt   sign8        jrugt.d sign8
jruge   sign8        jruge.d sign8
jrult   sign8        jrult.d sign8
jrule   sign8        jrule.d sign8
jreq    sign8        jreq.d  sign8
jrne    sign8        jrne.d  sign8
jp      sign8        jp.d    sign8
jp      %rb          jp.d    %rb
jpr     %rb          jpr.d   %rb
call    sign8        call.d  sign8
call    %rb          call.d  %rb
loop    %rc,%ra      loop    %rc,imm4     loop    imm4,imm4
```

5.22 Loop Instructions

5.22.1 Settings

A loop operation performs unconditional branching to the address set in the LSA register when the following three conditions are satisfied:

(1) The LM flag (bit 29) in the PSR is set.

(2) The address set in the LEA register and the current PC match.

(3) The LCO register value is equal to or greater than 1.

When a loop instruction is executed, the next address after the loop instruction is loaded into the LSA, and the loop count specified in the first operand is loaded into the special-register LCO. Specified in the second operand is the end address to be looped, which is loaded into the special-register LEA. At this time, the LM flag (bit 29) in the PSR representing loop mode is set to 1.
The available loop instructions are as follows:

```
loop  %rc,%ra
loop  %rc,imm4
loop  imm4,imm4
```

Instructions are executed sequentially beginning with the next address and, when the execution address PC matches the loop end address (LEA), the LCO is decremented by one and the execution address cycles back to the address indicated by the LSA. This operation is repeated until the LCO value reaches 0.

```
Example:        loop  %r1,loop_end
                ld.w  %r2,[%r3]+
                ld.w  [%r4]+,%r2
      loop_end:
                  :         :
```

In the above example, word data is copied from the address indicated by R3 to the address indicated by R4 a number of times equal to the count indicated by R1 plus 1. If the loop count in the first operand is specified in the *rc*, the target instructions to be looped are executed a maximum of 4G (0xFFFFFFFF) + 1 times; if specified in *imm4*, the instructions are executed a maximum of 15 + 1 times. Note that even when the loop count is set to 0, the target instructions in the loop are executed once.
The loop end address in the second operand is assumed to be an absolute address when specified by *ra*, or assumed to be a relative address from PC + 2 when specified by *imm4*, i.e., (PC + 2 + *imm4* × 2).

Furthermore, a loop operation can be entered into by setting each of the LEA, LSA, LCO, and LM flag (bit 29) individually. (This method is referred to as a "reserved loop.") When a reserved loop is performed, the following precautions should be observed:

(1) The LM flag (bit 29) must be set at the end.

(2) If the address at which the LM flag (bit 29) is set and the address that is set in the LEA are excessively close, a loop operation cannot be performed correctly. Therefore, one of the following conditions must be satisfied:
- A branch is inserted between the instruction that sets the LM flag (bit 29) and the LEA address.
- There are five or more instructions between the instruction that sets the LM flag (bit 29) and the LEA address. (However, if an `ext` instruction is included immediately after the instruction that sets the LM flag (bit 29), the extended part must be excluded from the instruction count.)

Given below are the precautions common to both loop and reserved-loop instructions:

(1) After the LM flag (bit 29) is set, the set values of the LEA, LSA, and LCO cannot be altered until the LCO reaches 0.

(2) No instructions can be placed at the LEA address or the LEA - 2 address that cannot be located at those addresses (described later).

Loop instructions are superior to branch instructions in that the prefetch mechanism monitors loop operating conditions and recycles the fetch address from the LEA to the LSA immediately. In other words, the instructions in a loop can be prefetched successively. For branch instructions, except for delayed branching, all of the prefetched instructions are discarded when the branch conditions are satisfied, and the instruction at the jump address must be fetched again. This is unnecessary for the loop operation.
However, because loop instructions are 5-clock instructions, even a reserved loop requires one clock cycle each to set the LCO, LEA, and LSA, and at least four clock cycles to set the PSR, in effect making loop instructions 7-clock instructions. If the loop counts are extremely small (e.g., looped only three or four times), the increase in processing speed resulting from the above advantage may be canceled out.

There must be at least two instructions that are executed repeatedly by a loop instruction. If it is only necessary to loop one instruction, use a repeat instruction.

5.22.2 Break from a Loop Operation

To stop a loop, the LM flag (bit 29) in the PSR must first be cleared using the `psrclr` instruction prior to branching (i.e., no loop operation can be inserted before the branch instruction after the flag is cleared). Because, if this only involves clearing the LM flag (bit 29), the program may operate erratically while the next instruction in the LSA address is being prefetched, make sure the prefetch queue is cleared by a branch instruction. However, if it is desired that the program return to this loop to execute the loop instructions after branching once, it is not necessary to clear the prefetch queue. Loop mode remains effective, so that unless the execution address PC and the LEA match and LCO = 0, the loop will be continued.

5.22.3 Prohibition of Loop Operation in Debug and MMU Exceptions

Make sure no loop operations will be performed in debug exception and MMU exception handler routines.

5.22.4 Exception Handling during Loop

The `loop` instruction accepts an exception that occurs during a loop. When an exception is accepted, the CPU saves the address of the instruction being executed to the stack as the return address, and then saves the PSR to the stack. Furthermore, the CPU clears the LM flag (bit 29) in the PSR to 0 and jumps to the vector address for the exception handler routine. For MMU and debug exceptions, however, the PSR is not saved, nor is the LM flag (bit 29) cleared to 0. Therefore, the PSR and the flag must be protected in the respective exception handler routines. Note, however, that loop/repeat operations cannot be performed in the debug exception and MMU exception handler routines.

5.22.5 Use of Multiple Loop/Repeats and Interrupts

If, during a loop, it is desired to use another loop or repeat, follow the execution procedure described below. However, do not perform loop/repeat operations in debug and MMU exceptions.

(1) Save the LSA, LEA, and LCO to registers or memory.

(1') If a reserved loop/repeat is to be used, clear the LM (bit 29) here.

(2) Execute the loop/repeat or the reserved loop/repeat instruction.

(3) When the loop/repeat has finished, load the saved LSA, LEA, and LCO from the registers or memory.

(4) Set the LM (bit 29).

However, because the relationship between (4) and the LEA in the original loop is subject to the application of precaution about reserved loops (2) specified in the previous page, it is recommended that multiple loop/repeats be used in a called subroutine (e.g., `call`).

To use a loop/repeat in interrupt handling, basically the same procedure as described above may be followed. However, because except for MMU and debug exceptions the LM flag (bit 29) is cleared to 0, the operation specified in (1') can be omitted. Furthermore, because the `reti` instruction includes loading of the PSR, the operation for setting the LM flag (bit 29) may also be omitted. Conversely, forcibly clearing the stacked LM flag (bit 29) has no effect, as the `reti` instruction for interrupts in a loop always returns the LM flag (bit 29) set.

5.22.6 Restrictions on Use of Instructions

There are some instructions that cannot be executed while a loop is being executed by the loop instruction, and those that may or may not be able to be executed depending on the conditions.

Instructions that cannot be set while a loop is being executed (the LM flag (bit 29) remains set)

`ret` (*1)		`ret.d` (*1)			
`reti`		`retd` (*1)		`retm` (*1)	
`repeat`	*`imm4`* (*2)	`repeat`	*`%rb`* (*2)		
`loop`	*`%rc,%ra`* (*2)	`loop`	*`%rc,imm4`* (*2)	`loop`	*`imm4,imm4`* (*2)

(*1) These instructions may be used provided that the precaution on using a loop/repeat in an interrupt is observed.
(*2) These instructions may be used provided that the precaution on using multiple loop/repeats is observed.

Instructions that cannot be used if any registers used by the `loop` instruction are included

`pops`	*`%sd`*

Instructions that can be used while a loop is being executed (the LM flag (bit 29) remains set) but cannot be used when the PC matches the LEA address

`brk`					
`int`	*`imm2`*				
`ext`	*`%rs`*	`ext`	*`cond`*	`ext`	*`op,imm2`*
`ext`	*`%rs,op,imm2`*	`ext`	*`imm13`*		
`jrgt`	*`sign8`*	`jrge`	*`sign8`*	`jrlt`	*`sign8`*
`jrle`	*`sign8`*	`jrugt`	*`sign8`*	`jruge`	*`sign8`*
`jrult`	*`sign8`*	`jrule`	*`sign8`*	`jreq`	*`sign8`*
`jrne`	*`sign8`*	`jp`	*`sign8`*		
`jp`	*`%rb`*				
`jpr`	*`%rb`*				
`call`	*`sign8`*	`call`	*`%rb`*		
`slp`		`halt`			

Instructions that can be used while a loop is being executed (the LM flag (bit 29) remains set) but cannot be used when the PC matches the LEA address or in the location immediately preceding that address

`jrgt.d`	*`sign8`*	`jrge.d`	*`sign8`*	`jrlt.d`	*`sign8`*
`jrle.d`	*`sign8`*	`jrugt.d`	*`sign8`*	`jruge.d`	*`sign8`*
`jrult.d`	*`sign8`*	`jrule.d`	*`sign8`*	`jreq.d`	*`sign8`*
`jrne.d`	*`sign8`*				
`jp.d`	*`sign8`*	`jp.d`	*`%rb`*		
`jpr.d`	*`%rb`*				
`call.d`	*`sign8`*	`call.d`	*`%rb`*		

5.23 Other Instructions

Flag control instructions

The C33 ADV Core CPU 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* to 1
`psrclr` ***`imm5`***	Clears the PSR bit specified by *imm5* to 0

6 Functions

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

6.1 Transition of the CPU Status

The diagram below shows the transition of the operating status in the C33 ADV Core CPU. The C33 ADV Core CPU supports supervisor mode, so the CPU enters supervisor mode immediately after reset or when an exception in the internal or external device occurs.

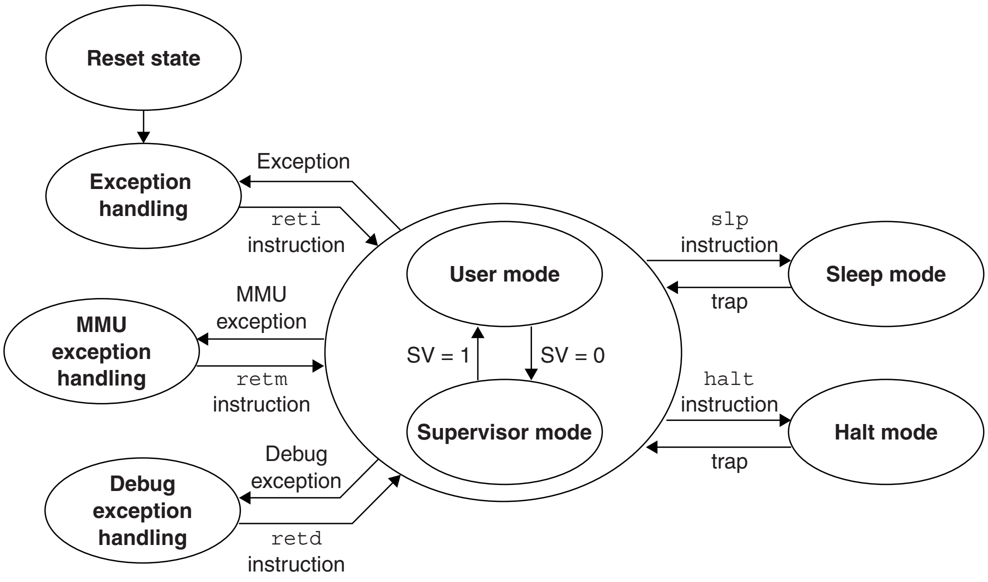


Figure 6.1.1 CPU Status Transition Diagram

6.1.1 Reset State

The CPU 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 Supervisor Mode

After the CPU is reset or an external interrupt or exception occurs, the SV flag (bit 12) in the PSR is set to 0 and the CPU is placed in supervisor mode. In supervisor mode, all CPU resources become available for use. The stack is switched to the area indicated by the SSP, and the stack manipulation instructions (e.g., `push` and `pop`) thereafter reference the SSP as the stack pointer.

6.1.3 User Mode

The CPU is placed in user mode by setting the SV flag (bit 12) in the PSR to 1 while the CPU is in supervisor mode. Although the CPU enters supervisor mode temporarily when an exception occurs, it is returned to user mode by a return instruction after exception handling has finished.
In user mode, register accesses are subject to the following restrictions:

PSR[12]	SV flag	Unchangeable, read only possible
PSR[11:8]	IL bits	Unchangeable, read only possible
PSR[4]	IE flag	Unchangeable, read only possible
TTBR[31:0]		Unchangeable, read only possible
SSP[31:0]		Unchangeable, read only possible

For the limitations on the memory address space, refer to the HBCU and BBCU sections in the Technical Manual for each model.

6.1.4 Exception Handling

When a software or other exception occurs, the SV flag (bit 12) in the PSR is reset to 0 and the CPU is placed in supervisor mode, thereby entering 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

6.1.5 MMU Exception

An MMU exception occurs if, while the MMU is active, the logical address that the CPU attempted to access is not mapped in the MMU table. When an MMU exception occurs, it is necessary that the MMU table be rewritten to the appropriate value in the software. For details, refer to the MMU section in the Technical Manual for each model. Furthermore, the CPU is placed in supervisor mode when the MMU exception handler routine is executed, regardless of whether the SV flag (bit 12) is set.

6.1.6 Debug Exception

The C33 ADV Core CPU 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 CPU can be switched from user mode to this mode by the `brk` instruction or a debug exception. The CPU does not normally enter this mode.

6.1.7 Halt Mode

The CPU is placed in halt mode by executing the `halt` instruction in the software. In halt mode, the CPU is in low-power-consumption mode, with the clock supplied to it turned off. The CPU can be taken out of halt mode by initial reset, NMI, or external interrupt.

6.1.8 Sleep Mode

The CPU is placed in sleep mode by executing the `slp` instruction in the software. In sleep mode, the CPU is in even lower power-consumption mode than in halt mode, with the clock supplied to it and those supplied to the peripheral circuits turned off. The CPU can be taken out of sleep mode by initial reset, NMI, or external interrupt.

6.2 Program Execution

Following initial reset, the CPU 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 ADV Core CPU 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 CPU 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 CPU 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 "0x20000000." 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 ADV Core CPU, instructions are processed in five pipelined stages, so that data transfer 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 5-stage pipeline, each instruction is processed in five stages, with processing of instructions occurring in parallel at several stages, 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:

Instruction fetch	→ F for Fetch
Instruction decode	→ D for Decode
Instruction execution	→ E for Execute
Memory access	→ A for Access
Register write	→ W for Write

Pipelined operation

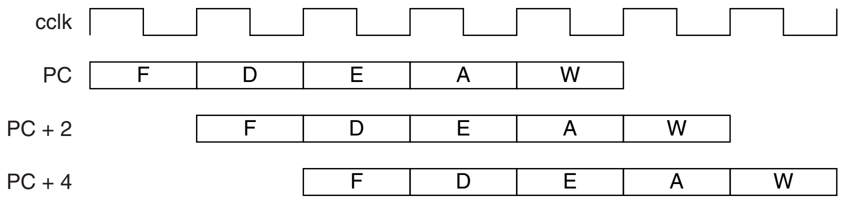


Figure 6.2.1.1 Pipelined Operation

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 F or A stage kept waiting.

6.2.2 Execution Cycles and Flags

The instructions in the C33 ADV Core CPU are processed in parallel at five pipelined stages as described above, so most instructions are executed in one clock cycle. This comprises the basic execution cycle in the C33 ADV Core CPU.
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 of bus contention between the HBCU and BBCU, 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.

S1C33 STD Core CPU compatible instructions

Table 6.2.2.1 Number of Instruction Execution Cycles and Flag Status (S1C33 STD Compatible Instructions)

Classification	Mnemonic		Cycle	Interlock cycle	Flag C	Flag V	Flag Z	Flag N	Remark
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*	1	2 (∗8)	–	–	–	–	
	mltu.h	*%rd,%rs*	1	2 (∗8)	–	–	–	–	
	mlt.w	*%rd,%rs*	2	2 (∗8)	–	–	–	–	
	mltu.w	*%rd,%rs*	2	2 (∗8)	–	–	–	–	
	div0s	*%rs*	1	2 (∗8)	–	–	–	↔	DS change
	div0u	*%rs*	1	2 (∗8)	–	–	–	0	DS = 0
	div1	*%rs*	1	2 (∗8)	–	–	–	–	
	div2s	*%rs*	1	2 (∗8)	–	–	–	–	
	div3s		1	2 (∗8)	–	–	–	–	
Branch	jrgt jrgt.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrge jrge.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrlt jrlt.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrle jrle.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrugt jrugt.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jruge jruge.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrult jrult.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrule jrule.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jreq jreq.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrne jrne.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jp	*sign8*	1–2 (∗7)		–	–	–	–	
	jp.d	*%rb*	2–3 (∗7)		–	–	–	–	
	call	*sign8*	1–2 (∗7)		–	–	–	–	
	call.d	*%rb*	2–3 (∗7)		–	–	–	–	

Classification	Mnemonic		Cycle	Interlock cycle	Flag				Remark
					C	V	Z	N	
Branch	ret ret.d		4–5 (*7)		–	–	–	–	
	reti		6		↔	↔	↔	↔	PSR change
	retd		6		–	–	–	–	DE = 0
	int	*imm2*	7		–	–	–	–	SV = 0, IE = 0
	brk		7		–	–	–	–	DE = 1, IE no change
Data transfer	ld.b	%*rd*,%*rs*	1		–	–	–	–	
		%*rd*,[%*rb*]	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%*rb*]+	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%sp+*imm6*]	1	1–2 (*9)	–	–	–	–	
		[%*rb*],%*rs*	1		–	–	–	–	
		[%*rb*]+,%*rs*	1		–	–	–	–	
		[%sp+*imm6*],%*rs*	1		–	–	–	–	
	ld.ub	%*rd*,%*rs*	1		–	–	–	–	
		%*rd*,[%*rb*]	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%*rb*]+	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%sp+*imm6*]	1	1–2 (*9)	–	–	–	–	
	ld.h	%*rd*,%*rs*	1		–	–	–	–	
		%*rd*,[%*rb*]	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%*rb*]+	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%sp+*imm6*]	1	1–2 (*9)	–	–	–	–	
		[%*rb*],%*rs*	1		–	–	–	–	
		[%*rb*]+,%*rs*	1		–	–	–	–	
		[%sp+*imm6*],%*rs*	1		–	–	–	–	
	ld.uh	%*rd*,%*rs*	1		–	–	–	–	
		%*rd*,[%*rb*]	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%*rb*]+	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%sp+*imm6*]	1	1–2 (*9)	–	–	–	–	
	ld.w	%*rd*,%*rs*	1		–	–	–	–	
		%*rd*,*sign6*	1		–	–	–	–	
		%*rd*,[%*rb*]	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%*rb*]+	1	1–2 (*9)	–	–	–	–	
		%*rd*,[%sp+*imm6*]	1	1–2 (*9)	–	–	–	–	
		[%*rb*],%*rs*	1		–	–	–	–	
		[%*rb*]+,%*rs*	1		–	–	–	–	
		[%sp+*imm6*],%*rs*	1		–	–	–	–	
System control	nop		1		–	–	–	–	
	halt		1		–	–	–	–	
	slp		1		–	–	–	–	
Immediate extension	ext	*imm13*	0–1 (*2)		–	–	–	–	
Bit manipulation	btst	[%*rb*],*imm3*	3		–	–	↔	–	
	bclr	[%*rb*],*imm3*	3		–	–	–	–	
	bset	[%*rb*],*imm3*	3		–	–	–	–	
	bnot	[%*rb*],*imm3*	3		–	–	–	–	
Other	swap	%*rd*,%*rs*	1		–	–	–	–	
	mirror	%*rd*,%*rs*	1		–	–	–	–	
	mac	%*rs*	2 + N × 2	2 (*8)	–	–	–	–	MO change
	pushn	%*rs*	N		–	–	–	–	PM, RC change
	popn	%*rd*	N	1–2 (*9)	–	–	–	–	PM, RC change

Function-extended instructions

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

Classification	Mnemonic		Cycle	Interlock cycle	Flag C	Flag V	Flag Z	Flag N	Remark
Logical operation	`and`	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
	`or`	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
	`xor`	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
	`not`	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
Shift and rotate	`srl`	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	`sll`	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	`sra`	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	`sla`	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	`rr`	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	`rl`	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
Data transfer	`ld.w`	*%rd,%ss*	1		–	–	–	–	
		%sd,%rs	1 (∗10)		–	–	–	–	
Other	`scan0`	*%rd,%rs*	1		↔	0	↔	0	
	`scan1`	*%rd,%rs*	1		↔	0	↔	0	

Added instructions

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

Classification	Mnemonic		Cycle	Interlock cycle	Flag C	Flag V	Flag Z	Flag N	Remark
Arithmetic operation	`add`	*%rd*,%dp	1		–	–	–	–	
	`mlt.hw`	*%rd,%rs*	2	2 (∗8)	–	–	–	–	
	`mac.hw`	*%rs*	2 + N × 2	2 (∗8)	–	–	–	–	MO change
	`mac.w`	*%rs*	3 + N × 2	2 (∗8)	–	–	–	–	MO change
	`mac1.h`	*%rd,%rs*	1	2 (∗8)	–	–	–	–	MO change
	`mac1.hw`	*%rd,%rs*	2	2 (∗8)	–	–	–	–	MO change
	`mac1.w`	*%rd,%rs*	2	2 (∗8)	–	–	–	–	MO change
	`div.w`	*%rs*	35	2 (∗8)	–	–	–	↔	DS change
	`divu.w`	*%rs*	35	2 (∗8)	–	–	–	0	DS = 0
Branch	`jpr` `jpr.d`	*%rb*	3–4 (∗7)		–	–	–	–	
	`retm`		6		–	–	–	–	ME = 0
Data transfer	`ld.b`	*%rd*,[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	`ld.ub`	*%rd*,[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	`ld.h`	*%rd*,[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	`ld.uh`	*%rd*,[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	`ld.w`	*%rd*,[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	`ld.b`	[%dp+*imm6*],*%rs*	1		–	–	–	–	
	`ld.h`	[%dp+*imm6*],*%rs*	1		–	–	–	–	
	`ld.w`	[%dp+*imm6*],*%rs*	1		–	–	–	–	
System control	`psrset`	*imm5*	4		↔	↔	↔	↔	
	`psrclr`	*imm5*	4		↔	↔	↔	↔	
Multifunction extension	`ext`	*%rs*	0–1 (∗2)		–	–	–	–	
	`ext`	*cond*	0–1 (∗2)		–	–	–	–	
	`ext`	*op,imm2*	0–1 (∗2)		–	–	–	–	
	`ext`	*%rs,op,imm2*	0–1 (∗2)		–	–	–	–	
Coprocessor control	`ld.c`	*%rd,imm4*	1		–	–	–	–	
	`ld.c`	*imm4,%rs*	1		–	–	–	–	
	`do.c`	*imm6*	1		–	–	–	–	
	`ld.cf`		1		↔	↔	↔	↔	

Classification	Mnemonic		Cycle	Interlock cycle	Flag				Remark
					C	V	Z	N	
Other	macclr		1		–	–	–	–	MO = 0
	swaph	*%rd,%rs*	1		–	–	–	–	
	push	*%rs*	1		–	–	–	–	
	pop	*%rd*	1	1–2 (∗9)	–	–	–	–	
	pushs	*%ss*	N		–	–	–	–	PM, RC change
	pops	*%sd*	N	1–2 (∗9)	–	–	–	–	PM, RC change
	sat.b	*%rd,%rs*	1		–	–	–	–	S change
	sat.ub	*%rd,%rs*	1		–	–	–	–	S change
	sat.h	*%rd,%rs*	1		–	–	–	–	S change
	sat.uh	*%rd,%rs*	1		–	–	–	–	S change
	sat.w	*%rd,%rs*	1		–	–	–	–	S change
	sat.uw	*%rd,%rs*	1		–	–	–	–	S change
	loop	*%rc,%ra*	5 (∗5)		–	–	–	–	LM change
		%rc,imm4	5 (∗5)		–	–	–	–	LM change
		imm4,imm4	5 (∗5)		–	–	–	–	LM change
	repeat	*%rc*	4 (∗6)		–	–	–	–	RM change
		imm4	4 (∗6)		–	–	–	–	RM change

∗1 Two 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 The flag changes when the OC flag (bit 21) in the PSR = 1.
∗4 The flag changes when the SE flag (bit 20) in the PSR = 1.
∗5 Five cycles only when a `loop` instruction is executed; cycles of each instruction for loop operation
∗6 Four cycles only when a `repeat` instruction is executed; cycles of each instruction for repeat operation
∗7 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.
∗8 These interlock cycles are incurred when AHR and ALR are referenced by the next instruction. However, 0 cycle if the next instruction is combined with `mac`, `mac1`, `mlt`, or `div`. In addition, two interlock cycles when R4 where LC flag = 1 or R5 where HC flag = 1 are referenced.
∗9 Two cycles only when the next instruction uses the *rd* register as an indirect address register
Example for two interlock cycles incurred

```
ld.w  %r1,[%r2] ; r1 ← [r2]
ld.w  %r3,[%r1] ; r1 used as an indirect address register
```

∗10 Four cycles only when the `ld.w %psr,`*`%rs`* instruction is executed

Shown in the Remark column are the PSR flags that affect the program operation other than the C, V, Z, and N flags.

The Interlock cycle column of the table indicates interlock cycles that are necessary before valid data is set in the *rd* register when the instruction shown to the left is executed. In other words, these are penalty clock cycles that are incurred due to the fact that, when the immediately following instruction accesses the *rd* register, it has to wait until valid data is set in the *rd* register. Therefore, the number of cycles in the Interlock cycle column must be added to the execution cycles required for the next instruction.

6.3 Interrupts and Exceptions

When an external interrupt or exception occurs during program execution, the CPU enters an exception handling state. The exception handling state is a process by which the CPU branches to the corresponding user's service routine for the interrupt or exception that occurred. The CPU 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 ADV Core CPU:

(1) Reset, internal exceptions of the CPU, and external interrupts for which the CPU branches to the relevant exception handler routine by referencing the vector table

(2) MMU exception, which is generated to control the logical address conversion table of the MMU when an MMU is used

(3) 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
Zero division	TTBR + 0x10	↑
Address misaligned exception	TTBR + 0x18	
Debug exception	0x00000000 or 0x00060000	
MMU exception	0x00000010	
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 CPU 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 and MMU exceptions have their vectors located at the specific addresses, and the vector table is not referenced for these exceptions. 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
0x00000000 / 0x00060000	Debug exception handler vector
0x00000008 / 0x00060008	PC save area
0x0000000C / 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.

The table below shows the addresses that are referenced when an MMU exception occurs.

Table 6.3.1.3 MMU Exception Vector Address and PC/R0 Save Area

Address	Content
0x00000010	MMU exception handler vector
0x00000018	PC save area
0x0000001C	R0 save area

During MMU exception handling, other exceptions are disabled and not accepted. Nor are multiple MMU exceptions generated, as MMU exceptions are handled in physical addresses. For details, refer to the MMU section in the Technical Manual for each model.

6.3.2 Vector Table

Vector table in the C33 ADV Core CPU

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	Priority
Reset	0	Asynchronous	Interrupt	TTBR + 0x00	High
reserved	1–3	–	–	–	↑
Zero division	4	Synchronous	Exception	TTBR + 0x10	
reserved	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	Low

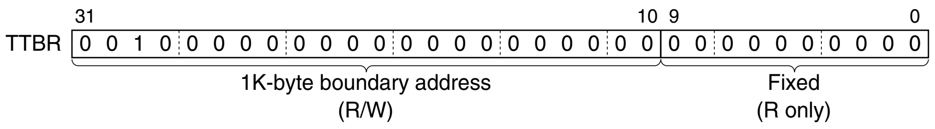
The sources of exceptions in the C33 ADV Core CPU are shown in Table 6.3.2.1.

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 CPU 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 ADV Core CPU, 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.

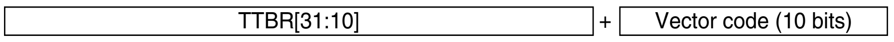
TTBR (Trap Table Base Register)



The initial value of the TTBR, or the value to which the TTBR is initialized when cold reset, is "0x20000000."

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.



Vector code is generated by the CPU.

6.3.3 Exception Handling

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

The exception handling performed by the CPU 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 between the A (memory access) and W (register write) stages of the currently executed instruction.

(2) Saves the contents of the PC and PSR to the stack (SSP), 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. In addition, the SV flag (bit 12) in the PSR is cleared to 0 with the CPU mode thereby switched to supervisor mode, and the RM (bit 30), LM (bit 29), and PM (bit 28) flags are cleared to 0.

(4) Reads the vector for the generated exception from the vector table, and sets it in the PC. The CPU 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 CPU mode is switched back to user mode with processing returned to the suspended instruction.

6.3.4 Reset

The CPU is reset by applying a low-level pulse to its #RESET pin.
The CPU 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 CPU thereby branches to the user's initialization routine, in which it starts executing the program. The reset sequence has priority over all other processing. The C33 ADV Core CPU supports two methods of reset, cold start and hot start.

Cold start (#RESET = low; #NMI = high)

When the #NMI pin is driven high and the #RESET pin then is pulled low to reset the chip, the CPU in the C33 ADV Core CPU cold-starts. In this case, all of the peripheral circuits in the chip, not just the CPU, are initialized, so that this reset method is used primarily for power-on reset.

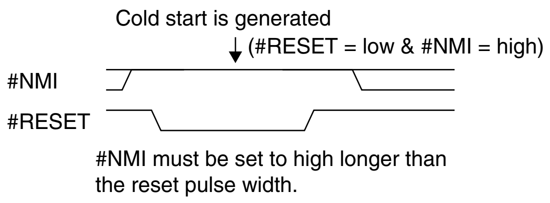

Cold start is generated
(#RESET = low & #NMI = high)
#NMI
#RESET
#NMI must be set to high longer than the reset pulse width.

Figure 6.3.4.1 Cold Reset Timing

Hot start (#RESET = low; #NMI = low)

When the #NMI pin is pulled low and the #RESET pin also is then pulled low to reset the chip, the CPU in the C33 ADV Core CPU hot-starts. In this case, although the CPU is initialized, some peripheral circuits such as the external bus control unit and input/output ports are not initialized. This reset method is therefore used to reset the chip while retaining the external memory or external input/output status.

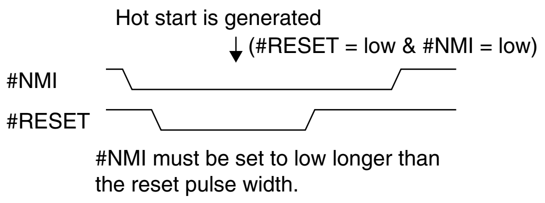

Hot start is generated
(#RESET = low & #NMI = low)
#NMI
#RESET
#NMI must be set to low longer than the reset pulse width.

Figure 6.3.4.2 Hot Reset Timing

The reset clears all bits of the PSR to 0. TTBR is initialized to 0x20000000 by a cold start or is not altered with the previous value before resetting maintained by a hot start. The contents of other registers become indeterminate.

6.3.5 Zero Divide Exception

A zero divide exception occurs if, during execution of a division instruction, the divisor is zero. This exception occurs in the `div0s` and `div0u` instructions that perform preprocessing of a division, and the `div.w` and `divu.w` instructions that perform a series of divide operations collectively as a batch. If the divisor in a division instruction is 0, the CPU performs exception handling after it finishes executing the instruction. The PC value saved to the stack in exception handling is the address for the next instruction.

6.3.6 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 CPU 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 next to the load instruction that caused the exception.

The multiply-accumulate (`mac`) instructions are likely to cause an address misaligned exception, as they handle halfword or word data in memory. In this case as well, the return address saved to the stack is the address of the `mac` instruction, so that after returning from the exception handler routine, the CPU resumes execution of the instruction performing the remaining number of multiply-accumulate operations.

In the load instructions that use the SP and DP 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.

6.3.7 NMI

An NMI is generated when the #NMI input on the CPU is asserted low. When an NMI occurs, the CPU 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 CPU 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 CPU is reset (either cold start or hot start), the program may run out of control, the #NMI input on the CPU is therefore masked in the hardware until the SP is set by the `ld.w %sp,%`*`rs`* instruction.

6.3.8 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 CPU 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.

6.3.9 Maskable External Interrupts

The C33 ADV Core CPU 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 CPU 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 CPU, 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 CPU 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 CPU.

(1) Suspends the instruction currently being executed.
The interrupt is accepted synchronously with the rising edge of the system clock between the A (memory access) and W (register write) stages of the currently executed instruction.

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

(3) Clears the IE flag in the PSR and copy the priority level of the accepted interrupt to the IL field. Furthermore, the SV flag in the PSR is cleared to 0, to switch the CPU mode to supervisor mode.

(4) Reads the vector for the interrupt from the vector address in the vector table, and sets it in the PC. The CPU 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 CPU mode is switched back to user mode, with processing returned to the suspended instruction.

6.3.10 MMU Exception

The C33 ADV Core CPU supports the use of an MMU to allow for memory management by converting the logical address space into the physical address space.
Conversion into physical addresses is achieved by using the information set in the MMU registers, and an MMU exception is generated when the program attempts to access an address that deviates from the logical address space divided by the register information (i.e., a miss). When an MMU exception occurs, the software must update the MMU register that caused the miss and remap the logical address into the physical address.
During an MMU exception, the CPU operates in supervisor mode regardless of whether the SV flag (bit 12) is set. No hardware or NMI interrupts are accepted during an MMU exception. Do not perform loop/repeat operations during an MMU exception.

The following describes MMU exception handling as performed by the CPU.

(1) Suspends the instruction currently being executed.
An MMU exception is generated at the end of the A (memory access) 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 → 0x00000014, R0 → 0x00000018

(3) Sets the ME flag (bit 13) in the PSR to 1.

(4) Loads the PC from the address 0x00000010 that is the MMU exception vector and branches to the MMU exception handler routine.

In the exception handler routine, the MMU registers should be updated to the appropriate value and, at the end of processing, execute the `retm` instruction to return to the suspended instruction. When returning from the exception by the `retm` instruction, the CPU restores the saved data in order of the R0 and the PC.
For details, refer to the MMU section in the Technical Manual for each model.

6.4 Power-Down Mode

The C33 ADV Core CPU supports two power-down modes: HALT and SLEEP modes. In power-down mode, the chip is placed in low-power-consumption mode with the functions of the CPU only or both the CPU and peripheral modules turned off.

6.4.1 HALT Mode

Program execution is halted at the same time that the CPU executes the `halt` instruction in supervisor mode or when the HE flag (bit 31) = 1, and the CPU enters HALT mode.
In HALT mode, the CPU stops operating, but because the peripheral circuits are supplied with clocks, they continue operating.
The CPU is taken out of HALT mode by an initial reset or an interrupt, including NMI. After exiting HALT mode, the CPU performs the relevant exception handling and restarts program execution. When freed from HALT mode by an interrupt, the address of the instruction next to `halt` is saved to the stack by exception handling. Therefore, if the exception handling for the generated interrupt is terminated by the `reti` instruction, the CPU returns to the instruction next to `halt`.

6.4.2 SLEEP Mode

Program execution is halted at the same time the CPU executes the `slp` instruction in supervisor mode or when the HE flag (bit 31) = 1, and the CPU enters SLEEP mode.
In SLEEP mode, both the CPU and the peripheral circuits stop operating. Therefore, the power consumption in the chip can be reduced more significantly than in HALT mode.
The CPU is taken out of SLEEP mode by an initial reset or an external interrupt, including NMI. After exiting SLEEP mode, the CPU performs the relevant exception handling and restarts program execution. When freed from SLEEP mode by an interrupt, the address of the instruction next to `slp` is saved to the stack by exception handling. Therefore, if the exception handling for the generated interrupt is terminated by the `reti` instruction, the CPU returns to the instruction next to `slp`.
Note, however, that since basically in SLEEP mode the oscillator circuit in the chip and the peripheral circuits using its output clock both stop, the CPU is freed from SLEEP mode, typically by an asynchronous external interrupt. Furthermore, because the oscillator circuit restarts oscillating upon exiting SLEEP mode, the CPU must wait until the clock oscillation stabilizes before it can start operating.

6.5 Debug Mode

The C33 ADV Core CPU has debug mode to assist in software development by the user.
The debug mode 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.

• Area break
A debug exception is generated when a specified area in the divided address spaces of the C33 ADV Core CPU (to which an independent #CE signal is output) is accessed.

• Bus break
A debug exception is generated when the data of the selected bus matches the set value.

• Bus trace
The value of the selected bus is traced.

• PC trace
The status of instruction execution by the CPU is traced.

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

(1) Suspends the instruction currently being executed.
A debug exception is generated at the end of the A (memory access) 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 (or 0x00000008)
R0 → 0x0006000C (or 0x0000000C)

(3) Loads the debug exception vector located at the address 0x00060000 (or 0x00000000) 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 CPU restores the saved data in order of the R0 and the PC.
Neither hardware interrupts nor NMI interrupts are accepted during a debug exception. Do not perform loop/repeat operations during a debug exception handling.

6.6 Coprocessor Interface

The C33 ADV Core CPU 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 Instruction Code

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
`%dp`, dp	Data pointer (DP) or its content
`%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, LCO = 0b0100, LSA = 0b0101, LEA = 0b0110, SOR = 0b0111, TTBR = 0b1000, DP = 0b1001, IDIR = 0b1010, DBBR = 0b1011, USP = 0b1101, SSP = 0b1110, 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.
RM	Repeat mode enable flag
LM	Loop mode enable flag
PM	Push/pop mode flag
RC[3:0]	Register counter field
S	Saturation flag
DE	Debug exception flag
ME	MMU exception flag
IL[3:0]	Interrupt level field
MO	MAC overflow flag
DS	Dividend sign flag
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
Extension 3)	*rd* ← *rs*1 + *rs*2 + C (“*op*, *imm2*” is usable)

Code

15			12	11			8	7–4	3–0	
1	0	1	1	1	0	0	0	*r s*	*r d*	0xB8__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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) Extension 3

```
ext   %rs2,op,imm2     ; op = sra, srl, sla, imm2 = 0-3
adc   %rd,%rs1         ; rd ← (rs1 + rs2 + C) op imm2
```

The register *rs2* specified by the `ext` instruction and C (carry) flag are added to the content of the *rs1* register, and the content of the *rs1* register is then shifted as indicated by *op* a number of bits equal to *imm2*, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(3) Delayed instruction

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

(4) Postshift

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `adc` instruction.

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
```

(3)
```
ext   %r2,srl,1
adc   %r3,%r1          ; r3 = (r1 + r2 + C) >> 1
```

add %rd, %dp

Function

Addition
Standard) *rd* ← *rd* + dp
Extension 1) *rd* ← dp + *imm13*
Extension 2) *rd* ← dp + *imm26*
Extension 3) *rd* ← dp + *rs* (“*op*, *imm2*” is usable)

Code

15	14	13	12	11	10	9	8	7	6	5	4	3–0
0	0	0	0	0	0	1	1	0	1	0	1	*rd*

0x035_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Src: Register direct (DP)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

One cycle

Description

(1) Standard

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

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

(2) Extension 1

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

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

(3) Extension 2

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

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

(4) Extension 3

```
ext   %rs,op,imm2      ; op = sra, srl, sla, imm2 = 0-3
add   %rd,%dp          ; rd ← (dp + rs) op imm2
```

The register *rs* specified by the `ext` instruction is added to the content of the DP register, and the content of the DP register is then shifted as indicated by *op* a number of bits equal to *imm2*, and the result is loaded into the *rd* register. The contents of the DP and *rs* registers are not altered.

(5) 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.

(6) Postshift

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation and do not change.

Example

```
(1) add   %r0,%dp          ; r0 = r0 + dp
    ext   0x1
    ext   0x1fff
    add   %r1,%dp          ; r1 = dp + 0x3fff

(2) ext   %r2,srl,1
    add   %r3,%dp          ; r3 = (dp + r2) >> 1
```

add %rd, %rs

Function

Addition

Standard) *rd* ← *rd* + *rs*

Extension 1) *rd* ← *rs* + *imm13*

Extension 2) *rd* ← *rs* + *imm26*

Extension 3) *rd* ← *rs1* + *rs2* ("*op*, *imm2*" is usable)

Code

15	14	13	12	11	10	9	8	7–4	3–0	
0	0	1	0	0	0	1	0	*rs*	*rd*	0x22__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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 *imm*26 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) Extension 3

```
ext   %rs2,op,imm2     ; op = sra, srl, sla, imm2 = 0-3
add   %rd,%rs1         ; rd ← (rs1 + rs2) op imm2
```

The register *rs2* specified by the `ext` instruction is added to the content of the *rs1* register, and the content of the *rs1* register is then shifted as indicated by *op* a number of bits equal to *imm2*, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(5) 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.

(6) Postshift

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `add` instruction.

Example

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

(2) ext   0x1
    ext   0x1fff
    add   %r1,%r2          ; r1 = r2 + 0x3fff

(3) ext   %r2,srl,1
    add   %r3,%r1          ; r3 = (r1 + r2) >> 1
```

add %rd, imm6

Function

Addition

Standard)	*rd* ← *rd* + *imm6*
Extension 1)	*rd* ← *rd* + *imm19*
Extension 2)	*rd* ← *rd* + *imm32*
Extension 3)	Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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.

(5) Postshift (“`ext op,imm2`” only)

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `add` instruction.

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
Extension 3)	Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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.

(3) Postshift (“`ext` *`op`*`,`*`imm2`*” only)
The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation and do not change.

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*
Extension 3) *rd* ← *rs1* & *rs2*

Code

15	14	13	12	11	10	9	8	7–4	3–0	
0	0	1	1	0	0	1	0	rs	rd	0x32__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

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) Extension 3

```
ext   %rs2
and   %rd,%rs1         ; rd ← rs1 & rs2
```

The content of the *rs1* register and the register *rs2* specified by the `ext` instruction are logically AND'ed, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(5) 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.

*1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `or`, `xor`, and `not` instructions, refer to the description of each instruction.)

Example

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

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

(3) ext   %r5
    and   %r3,%r4          ; r3 = r4 & r5
```

and %rd, sign6

Function

Logical AND

Standard)	*rd* ← *rd* & *sign6*
Extension 1)	*rd* ← *rd* & *sign19*
Extension 2)	*rd* ← *rd* & *sign32*
Extension 3)	Unusable

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	1	0	0	*sign6*	*rd*	0x70__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

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.

*1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `or`, `xor`, and `not` instructions, refer to the description of each instruction.)

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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3	2–0	
1	0	1	0	1	1	0	0	*r b*	0	*imm3*	0xAC__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Three cycles

Description

(1) Standard

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

Clears a data bit of the byte data in the address specified with the *rb* register. The 3-bit immediate *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*)
Extension 3)	Unusable

Code

15	14	13	12	11	10	9	8	7–4	3	2–0	
1	0	1	1	0	1	0	0	*rb*	0	*imm3*	0xB4__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Three cycles

Description

(1) Standard

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

Reverses a data bit of the byte data in the address specified with the *rb* register. The 3-bit immediate *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[0x8(or 0x60008)] ← pc + 2, W[0xC(or 0x6000C)] ← r0, pc ← W[0x0(or 0x60000)]
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3			0	
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0x0400

Flag

IE	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	1	–	–	–	–	–	–	–

Mode

–

CLK

Seven 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 (0x0000000 or 0x0060000) and sets it to the PC. Thus the program branches to the debug-handler routine. Furthermore the CPU 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
Extension 3)	Unusable

Code

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

0xB0__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Three cycles

Description

(1) Standard

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

Sets a data bit of the byte data in the address specified with the *rb* register. The 3-bit immediate *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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3	2–0	
1	0	1	0	1	0	0	0	*rb*	0	*imm3*	0xA8__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	↔	–

Mode

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

CLK

Three cycles

Description

(1) Standard

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

Tests a data bit of the byte data in the address specified with the *rb* register and sets the Z (zero) flag if the bit is 0. The 3-bit immediate *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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

`call`	Three cycles
`call.d`	Two 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*
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

`call`	Two cycles
`call.d`	One cycle

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*
Extension 3)	Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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*
Extension 3) Unusable

Code

15			12	11	10	9–4	3–0	
0	1	1	0	1	0	*sign6*	*r d*	0x68__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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.
```

div0s *%rs*

Function

Signed division 1st step

Standard)	Initialization for division
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3			0	
1	0	0	0	1	0	1	1	*rs*				0	0	0	0	0x8B_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	↔	–	–	–	↔

Mode

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

CLK

One cycle

Description

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

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

Therefore, it is necessary that the dividend and divisor in the ALR and the *rs* register have been sign-extended into 32 bits.
The `div1` instruction should be executed after executing the `div0s` instruction. Then correct the results using the `div2s` and `div3s` instructions in signed division.

Example

Signed division (32 bits ÷ 32 bits)
When the dividend has been set to the R0 register and the divisor to the R1 register:

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

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

Caution

(1) A zero-division exception occurs if the `div0s` instruction is executed by setting the *rs* register to 0.
Up to 32-bit data can be used for both dividends and divisors.

(2) The proper DS value may not be obtained if PSR is read using the `ld.w` instruction immediately after the `div0s` instruction has been executed. To avoid this erroneous reading, insert two or more instructions between the `div0s` instruction and `ld.w` instruction that read the DS flag.

div0u %rs

Function

Unsigned division 1st step

Standard)	Initialization for division
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

0x8F_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	0	–	–	–	0

Mode

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

CLK

One cycle

Description

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

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

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

Example

Unsigned division (32 bits ÷ 32 bits)
When the dividend has been set to the R0 register and the divisor to the R1 register:

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

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

Caution

(1) A zero-division exception occurs if the `div0u` instruction is executed by setting the *rs* register to 0.
Up to 32-bit data can be used for both dividends and divisors.

(2) The proper DS value may not be obtained if PSR is read using the `ld.w` instruction immediately after the `div0u` instruction has been executed. To avoid this erroneous reading, insert two or more instructions between the `div0u` instruction and `ld.w` instruction that read the DS flag.

div1 *%rs*

Function

Division

Standard)	Step division
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3			0	
1	0	0	1	0	0	1	1	*rs*				0	0	0	0	0x93_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

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

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

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

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

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

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

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

In unsigned division, the results are obtained from the following registers by executing the necessary `div1` instruction steps.
The results of unsigned division: ALR = Quotient, AHR = Remainder

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

Example Unsigned division (32 bits ÷ 32 bits)
When the dividend has been set to the R0 register and the divisor to the R1 register:

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

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

Signed division (32 bits ÷ 32 bits)
When the dividend has been set to the R0 register and the divisor to the R1 register:

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

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

div2s *%rs*

Function

Correction step 1 for signed division results

Standard)	Correction process for the execution results of signed division
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15–12	11–8	7–4	3–0	
1 0 0 1	0 1 1 1	*rs*	0 0 0 0	0x97_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

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

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

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

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

Example

Signed division (32 bits ÷ 32 bits)
When the dividend has been set to the R0 register and the divisor to the R1 register:

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

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

div3s

Function

Correction step 2 for signed division results

Standard)	Correction process for the execution results of signed division
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15	14	13	12	11	10	9	8	7–4	3	2	1	0	
1	0	0	1	1	0	1	1	*rs*	0	0	0	0	0x9B_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

–

CLK

One cycle

Description

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

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

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

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

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

Example

Signed division (32 bits ÷ 32 bits)
When the dividend has been set to the R0 register and the divisor to the R1 register:

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

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

div.w %*rs*

Function

Signed 32-bit division

Standard)	ALR ← ALR / *rs* (quotient), AHR ← ALR / *rs* (remainder)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3–0	
0	0	0	0	0	0	1	0	0	1	0	1	*rs*	0x025_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	↔	–	–	–	↔

Mode Register direct `%rs` = `%r0` to `%r15`

CLK 35 cycles

Description Signed 32-bit division is performed using the ALR register as the dividend and the *rs* register as the divisor. The quotient and remainder resulting from this operation are loaded into the ALR and AHR, respectively.

Example When r0 = -20, r1 = 6

```
ld.w   %alr,%r0        ; ALR ← -20
div.w  %r1             ; ALR ← -3, AHR ← -2
```

Caution When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

divu.w %rs

Function

Unsigned 32-bit division

Standard)	ALR ← ALR / *rs* (quotient), AHR ← ALR / *rs* (remainder)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

0x021_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	0	–	–	–	0

Mode

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

CLK

35 cycles

Description

Unsigned 32-bit division is performed using the ALR register as the dividend and the *rs* register as the divisor. The quotient and remainder resulting from this operation are loaded into the ALR and AHR, respectively.

Example

When r0 = 20, r1 = 6

```
ld.w   %alr,%r0         ; ALR ← 20
divu.w %r1              ; ALR ← 3, AHR ← 2
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

do.c *imm6*

Function

Coprocessor execution

Standard)	W[CA(*imm6*)]
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7	6	5					0	
1	0	1	1	1	1	1	1	0	0	*imm6*						0xBF0_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3)	Unusable

Code

15		13	12 … 0	
1	1	0	*imm13*	0xC0__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Immediate data (unsigned)

CLK

Zero or one cycle (zero cycles when predecoded)

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` *`imm13`* 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, only two instructions, the first and the last (prior to the target instruction) are effective and the middles are invalidated.
See descriptions of each instruction for the extension contents and the usage.

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      ; Valid
ext   0x1         ; Invalid
ext   0x1fff      ; Valid
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.

ext *%rs*

Function

Operand extension

Standard)	Extends to three operands
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

0x3F_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Zero or one cycle (zero cycles when predecoded)

Description

The register indicated by *rs* is added for extension to the operand that immediately follows the instruction, so the instruction functions as a 3-operand instruction.

Extendable instructions

ALU instructions	Addressing mode
`add`, `sub`, `adc`, `sbc`, `and`, `or`, `xor`	`%rd`, `%rs`

Shift instructions	Addressing mode
`srl`, `sll`, `sra`, `sla`, `rr`, `rl`	`%rd`, `imm5`
`srl`, `sll`, `sra`, `sla`, `rr`, `rl`	`%rd`, `%rs`

Load instructions	Addressing mode
`ld.b`, `ld.ub`, `ld.h`, `ld.uh`, `ld.w`	`%rd`, `[%rb]`
`ld.b`, `ld.ub`, `ld.h`, `ld.uh`, `ld.w`	`[%rb]`, `%rs`
`ld.b`, `ld.ub`, `ld.h`, `ld.uh`, `ld.w`	`%rd`, `[%rb]+`
`ld.b`, `ld.ub`, `ld.h`, `ld.uh`, `ld.w`	`[%rb]+`, `%rs`

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

```
(1) ext    %r3
    add    %r1,%r2          ; r1 ← r2 + r3

(2) ext    %r2
    sla    %r3,5            ; r3(31:5) ← r2 << 5, r3(4:0) ← 0000

(3) ext    %r3
    ld.h   %r5,[%r6]        ; r5 ← [r6 + r3]

(4) ext    %r4
    ld.w   [%r7]+,%r8       ; [r7] ← r8, r7 ← r7 + r4
```

ext %rs, op, imm2

Function

Operand extension with postshift

Standard)	Extends the operand with `add`, `sub`, or other arithmetic instructions postshifted
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–2	1–0	
0	0	1	1	1	1	1	1	*rs*	*op*	*imm2*	0x3F__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Src: Register direct `%rs` = `%r0` to `%r15`
op: Shift mode (`sra`, `srl`, `sll`)
imm2: Number of bits shifted

CLK

Zero or one cycle (zero cycles when predecoded)

Description

The register indicated by *rs* is added for extension to the operand that immediately follows the instruction. Furthermore, the result of various arithmetic instructions executed is shifted a maximum of 3 bits by specifying *op* and *imm2*. The type and direction of shift is determined by *op*, and `sra`, `srl`, or `sll` can be specified for it. The result is shifted in the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are determined only by the result of the arithmetic instruction executed, and are not changed by the shift operation.
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.

Instruction group for which *op* and *imm2* function effectively

```
sub  %rd,%rs    sbc  %rd,%rs
add  %rd,%rs    adc  %rd,%rs    add  %rd,%dp
```

Example

```
(1) ext  %r3,sll,2
    add  %r1,%r2          ; r1(31:2) = (r2 + r3) << 2, r1(1:0) = 00

(2) ext  %r1,srl,3
    ext  0x05
    sub  %r2,%r3          ; %r2(28:0) = (r3 - 5) >> 3, r2(31:29) = 000
```

ext *op*, *imm2*

Function

Immediate extension with postshift

Standard)	Postshifts `add`, `adc`, or `sub` instructions
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3 2	1 0	
0	0	1	1	1	0	1	1	0	0	0	0	*op*	*imm2*	0x3B0_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

op: Shift mode (`sra`, `srl`, `sll`)
imm2: Number of bits shifted

CLK

Zero or one cycle (zero cycles when predecoded)

Description

The result of various arithmetic instructions executed is shifted a maximum of 3 bits by specifying *op* and *imm2*. The type and direction of shift is determined by *op*, and `sra`, `srl`, or `sll` can be specified for it. The result is shifted in the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are determined only by the result of the arithmetic instruction executed, and are not changed by the shift operation.
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.

Instruction group for which *op* and *imm2* function effectively

```
sub  %rd,%rs    sub  %rd,imm6    sub  %sp,imm10    sub  %rd,%rs
add  %rd,%rs    add  %rd,imm6    add  %sp,imm10    adc  %rd,%rs
add  %rd,%dp
```

Example

```
(1) ext   sll,2
    add   %r1,%r2  ; r1(31:2) ← (r1 + r2) << 2, r1(1:0) ← 00

(2) ext   sra,1
    ext   0x0123
    add   %r3,%r4  ; r3(30:0) ← (r4 + 0x0123) >> 1, r3(31) ← r3(31)
```

ext *cond*

Function

Conditional execution

Standard)	Does not execute the next instruction when the condition is satisfied
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7–4	3			0	
0	0	1	1	1	0	1	1	*cond*	0	0	0	0	0x3B_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

cond: psr(3:0)

CLK

Zero or one cycle (zero cycles when predecoded)

Description

If the condition of the status flag in the PSR and *cond* match, the next instruction is not executed. Shown below are the *cond* and the flag conditions.

cond	Flag condition
`gt`	!Z & !(N ^ V)
`ge`	!(N ^ V)
`lt`	N ^ V
`le`	Z | (N ^ V)
`ugt`	!Z & !C
`uge`	!C
`ult`	C
`ule`	Z | C
`eq`	Z
`ne`	!Z

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

```
cmp      %r2,%r3
ext      ult            ; When PSR(C) = 1, the next instruction
                        ; is not executed.
add      %r1,%r2        ; r1 ← r1 + r2
```

Caution

Do not place the branch instructions listed below directly after this instruction. If accompanied by those instructions, this instruction becomes invalid.

```
jrgt   sign8    jrgt.d  sign8    jrge    sign8    jrge.d  sign8
jrlt   sign8    jrlt.d  sign8    jrle    sign8    jrle.d  sign8
jrugt  sign8    jrugt.d sign8    jruge   sign8    jruge.d sign8
jrult  sign8    jrult.d sign8    jrule   sign8    jrule.d sign8
jreq   sign8    jreq.d  sign8    jrne    sign8    jrne.d  sign8
jp     sign8    jp.d    sign8    jp      %rb      jp.d    %rb
jpr    %rb      jpr.d   %rb
call   sign8    call    %rb      call.d  sign8    call.d  %rb
ret             ret.d            reti             retd
retm
int    imm2     brk
slp             halt
loop            repeat
```

halt

Function

HALT

Standard)	Sets the CPU to HALT mode
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode –

CLK One cycle

Description

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

Example

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

int *imm2*

Function

Software exception

Standard)	ssp ← ssp - 4, W[ssp] ← pc + 2, ssp ← ssp - 4, W[ssp] ← psr, pc ← Software exception vector
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

SV	IE	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
0	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 ADV core CPU 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: 0x20000000).
The `reti` instruction should be used for return from the handler routine.

Example

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

Caution

When the `int` instruction is executed, the CPU uses SSP as the stack pointer regardless of its operation mode.

jp *%rb* / jp.d *%rb*

Function

Unconditional jump

Standard)	pc ← *rb*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	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*
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

`jp`	Two cycles
`jp.d`	One cycle

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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

`jpr`	Four cycles
`jpr.d`	Three 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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3) Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

Description

(1) Standard

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

If the condition below has been met, this instruction doubles the signed 8-bit immediate *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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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 signed 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
Extension 3) Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3)	Unusable

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Signed PC relative

CLK

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

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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	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)
Extension 3)	*rd*(7:0) ← B[*rb1* + *rb2*], *rd*(31:8) ← B[*rb1* + *rb2*](7)

Code

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

0x20__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext   %rb2
ld.b  %rd,[%rb1]     ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the content of the *rb1* register with that of the *rb2* register added comprises the memory address, the byte data in which is transferred to the *rd* register. The contents of the *rb1* and *rb2* registers are 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) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← B[*rb*](7), *rb* ← *rb* + *sign13*

Extension 2) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← B[*rb*](7), *rb* ← *rb* + *sign26*

Extension 3) *rd*(7:0) ← B[*rb1*], *rd*(31:8) ← B[*rb1*](7), *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7 – 4	3 – 0	
0	0	1	0	0	0	0	1	*rb*	*rd*	0x21__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

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

CLK

One cycle

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. Following data transfer, the address in the *rb* register is incremented by 1.

(2) Extension 1

```
ext    imm13          ; = sign13
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. Following data transfer, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,095.

(3) Extension 2

```
ext    imm13          ; = sign26(25:13)
ext    imm13          ; = sign26(12:0)
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. Following data transfer, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,431.

(4) Extension 3

```
ext    %rb2
ld.b   %rd,[%rb1]+    ; memory address = rb1, rb1 ← rb1 + rb2
```

The byte data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The *rb1* register contains the memory address to be accessed. Following data transfer, the *rb2* register is added to the address in the *rb1* register, with the result stored in *rb1*. The range of *rb2* is -2,147,483,648 to +2,147,483,647.

ld.b *%rd*, [%dp + *imm6*]

Function

Signed byte data transfer

Standard)	*rd*(7:0) ← B[dp + *imm6*], *rd*(31:8) ← B[dp + *imm6*](7)
Extension 1)	*rd*(7:0) ← B[dp + *imm19*], *rd*(31:8) ← B[dp + *imm19*](7)
Extension 2)	*rd*(7:0) ← B[dp + *imm32*], *rd*(31:8) ← B[dp + *imm32*](7)
Extension 3)	Unusable

Code

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

0xE0__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.b   %rd,[%dp + imm6]       ; memory address = dp + 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 DP 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,[%dp + imm6]       ; memory address = dp + imm19,
                              ; imm6 ← imm19(5:0)
```

The `ext` instruction extends the displacement to a 19-bit quantity. As a result, the content of the DP 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,[%dp + imm6]       ; memory address = dp + imm32,
                              ; imm6 ← imm32(5:0)
```

The two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the DP 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,[%dp + 0x1]   ; r0 ← [dp + 0x41] sign-extended
```

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)
Extension 3)	Unusable

Code

15			12	11	10	9–4	3–0	
0	1	0	0	0	0	*imm6*	*r d*	0x40__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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)
Extension 3) B[*rb1* + *rb2*] ← *rs*(7:0)

Code

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

0x34__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext     %rb2
ld.b    [%rb1],%rs      ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the 8 low-order bits of the *rs* register are transferred to the address indicated by the content of the *rb1* register with the *rb2* register added. The contents of the *rb1* and *rb2* registers are not altered.

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

Function

Signed byte data transfer

Standard) B[*rb*] ← *rs*(7:0), *rb* ← *rb* + 1
Extension 1) B[*rb*] ← *rs*(7:0), *rb* ← *rb* + *sign13*
Extension 2) B[*rb*] ← *rs*(7:0), *rb* ← *rb* + *sign26*
Extension 3) B[*rb1*] ← *rs*(7:0), *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7–4	3–0	
0	0	1	1	0	1	0	1	*r b*	*r s*	0x35__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.b   [%rb]+,%rs     ; memory address = rb, rb ← rb + 1
```

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.

(2) Extension 1

```
ext    imm13          ; = sign13
ld.b   [%rb]+,%rs     ; memory address = rb, rb ← rb + sign13
```

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, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,095.

(3) Extension 2

```
ext    imm13          ; = sign26(25:13)
ext    imm13          ; = sign26(12:0)
ld.b   [%rb]+,%rs     ; memory address = rb, rb ← rb + sign26
```

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, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,431.

(4) Extension 3

```
ext    %rb2
ld.b   [%rb1]+,%rs    ; memory address = rb1, rb1 = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing. The 8 low-order bits of the *rs* register are transferred to the address indicated by the *rb1* register. Following data transfer, the *rb2* register is added to the content of the *rb1* register, with the result stored in the *rb1* register. The range of *rb2* is -2,147,483,648 to +2,147,483,647.

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

Function

Signed byte data transfer

Standard)	B[dp + *imm6*] ← *rs*(7:0)
Extension 1)	B[dp + *imm19*] ← *rs*(7:0)
Extension 2)	B[dp + *imm32*] ← *rs*(7:0)
Extension 3)	Unusable

Code

15			12	11	10	9–4	3–0	
1	1	1	1	0	1	*imm6*	*r s*	0xF4__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

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

The 8 low-order bits of the *rs* register are transferred to the specified memory location. The content of the current DP 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   [%dp + imm6],%rs      ; memory address = dp + 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 DP with the 19-bit immediate *imm19* added.

(3) Extension 2

```
ext    imm13                 ; = imm32(31:19)
ext    imm13                 ; = imm32(18:6)
ld.b   [%dp + imm6],%rs      ; memory address = dp + 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 DP with the 32-bit immediate *imm32* added.

Example

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

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)
Extension 3)	Unusable

Code

15	14	13	12	11	10	9–4	3–0	
0	1	0	1	0	1	*imm6*	*r s*	0x54__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One 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
Extension 3)	Unusable

Code

15			12	11			8	7	4	3	0	
1	0	1	1	0	0	0	1	*imm4*		*r d*		0xB1__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0
1	0	1	1	0	1	0	1	*imm4*	*r s*

0xB5__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

Mode

–

CLK

One cycle

Description

(1) Standard

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

(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.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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	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)
Extension 3)	*rd*(15:0) ← H[*rb1* + *rb2*], *rd*(31:16) ← H[*rb1* + *rb2*](15)

Code

15			12	11			8	7–4	3–0	
0	0	1	0	1	0	0	0	*rb*	*rd*	0x28__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext   %rb2
ld.h  %rd,[%rb1]     ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the content of the *rb1* register with that of the *rb2* register added comprises the memory address, the halfword data in which is transferred to the *rd* register. The contents of the *rb1* and *rb2* registers are 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) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15), *rb* ← *rb* + *sign13*
Extension 2) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15), *rb* ← *rb* + *sign26*
Extension 3) *rd*(15:0) ← H[*rb1*], *rd*(31:16) ← H[*rb1*](15), *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7			4	3			0	
0	0	1	0	1	0	0	1	*rb*				*rd*				0x29__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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. Following data transfer, the address in the *rb* register is incremented by 2.

(2) Extension 1

```
ext     imm13           ; = sign13
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. Following data transfer, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,094.

(3) Extension 2

```
ext     imm13           ; = sign26(25:13)
ext     imm13           ; = sign26(12:0)
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. Following data transfer, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,430.

(4) Extension 3

```
ext     %rb2
ld.h    %rd,[%rb1]+     ; memory address = rb1, rb1 ← rb1 + rb2
```

The halfword data in the specified memory location is transferred to the *rd* register after being sign-extended to 32 bits. The *rb1* register contains the memory address to be accessed. Following data transfer, the *rb2* register is added to the address in the *rb1* register, with the result stored in *rb1*. The range of *rb2* is -2,147,483,648 to +2,147,483,646.

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*, [%dp + *imm6*]

Function

Signed halfword data transfer

Standard) *rd*(15:0) ← H[dp + *imm6* × 2], *rd*(31:16) ← H[dp + *imm6* × 2](15)
Extension 1) *rd*(15:0) ← H[dp + *imm19*], *rd*(31:16) ← H[dp + *imm19*](15)
Extension 2) *rd*(15:0) ← H[dp + *imm32*], *rd*(31:16) ← H[dp + *imm32*](15)
Extension 3) Unusable

Code

15			12	11	10	9 … 4	3 … 0	
1	1	1	0	1	0	*imm6*	*rd*	0xE8__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.h   %rd,[%dp + imm6]      ; memory address = dp + 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 DP 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,[%dp + imm6]      ; memory address = dp + imm19,
                             ; imm6 = imm19(5:0)
```

The `ext` instruction extends the displacement to a 19-bit quantity. As a result, the content of the DP 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,[%dp + imm6]      ; memory address = dp + imm32,
                             ; imm6 = imm32(5:0)
```

The two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the DP 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,[%dp + 0x2]  ; r0 ← [dp + 0x42] sign-extended
```

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

Function

Signed halfword data transfer

Standard) *rd*(15:0) ← H[sp + *imm6* × 2], *rd*(31:16) ← H[sp + *imm6* × 2](15)
Extension 1) *rd*(15:0) ← H[sp + *imm19*], *rd*(31:16) ← H[sp + *imm19*](15)
Extension 2) *rd*(15:0) ← H[sp + *imm32*], *rd*(31:16) ← H[sp + *imm32*](15)
Extension 3) Unusable

Code

15			12	11	10	9–4	3–0	
0	1	0	0	1	0	*imm6*	*r d*	0x48__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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)
Extension 3) H[*rb1* + *rb2*] ← *rs*(15:0)

Code

15			12	11			8	7–4	3–0	
0	0	1	1	1	0	0	0	*rb*	*rs*	0x38__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext     %rb2
ld.h    [%rb1],%rs     ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the 16 low-order bits of the *rs* register are transferred to the address indicated by the content of the *rb1* register with the *rb2* register added. The contents of the *rb1* and *rb2* registers are 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) H[*rb*] ← *rs*(15:0), *rb* ← *rb* + *sign13*
Extension 2) H[*rb*] ← *rs*(15:0), *rb* ← *rb* + *sign26*
Extension 3) H[*rb1*] ← *rs*(15:0), *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7–4	3–0
0	0	1	1	1	0	0	1	*r b*	*r s*

0x39__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.h   [%rb]+,%rs     ; memory address = rb, rb ← rb + 2
```

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.

(2) Extension 1

```
ext    imm13          ; = sign13
ld.h   [%rb]+,%rs     ; memory address = rb, rb ← rb + sign13
```

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, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,094.

(3) Extension 2

```
ext    imm13          ; = sign26(25:13)
ext    imm13          ; = sign26(12:0)
ld.h   [%rb]+,%rs     ; memory address = rb, rb ← rb + sign26
```

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, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,430.

(4) Extension 3

```
ext    %rb2
ld.h   [%rb1]+,%rs    ; memory address = rb1, rb1 = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing. The 16 low-order bits of the *rs* register are transferred to the address indicated by the *rb1* register. Following data transfer, the *rb2* register is added to the content of the *rb1* register, with the result stored in the *rb1* register. The range of *rb2* is -2,147,483,648 to +2,147,483,646.

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 [%dp + *imm6*], *%rs*

Function

Signed halfword data transfer

Standard) H[dp + *imm6* × 2] ← *rs*(15:0)
Extension 1) H[dp + *imm19*] ← *rs*(15:0)
Extension 2) H[dp + *imm32*] ← *rs*(15:0)
Extension 3) Unusable

Code

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

0xF8__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

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

The 16 low-order bits of the *rs* register are transferred to the specified memory location. The content of the current DP 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   [%dp + imm6],%rs      ; memory address = dp + 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 DP 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   [%dp + imm6],%rs      ; memory address = dp + 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 DP 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  [%dp + 0x2],%r0  ; H[dp + 0x42] ← 16 low-order bits of r0
```

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)
Extension 3)	Unusable

Code

15			12	11	10	9–4	3–0	
0	1	0	1	1	0	*imm6*	*r s*	0x58__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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
Extension 3)	Unusable

Code

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

0xA5__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Extension 3) *rd*(7:0) ← B[*rb1* + *rb2*], *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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

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

CLK

One cycle

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.

(4) Extension 3

```
ext     %rb2
ld.ub   %rd,[%rb1]    ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the content of the *rb1* register with that of the *rb2* register added comprises the memory address, the byte data in which is transferred to the *rd* register. The contents of the *rb1* and *rb2* registers are 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) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0, *rb* ← *rb* + *sign13*
Extension 2) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0, *rb* ← *rb* + *sign26*
Extension 3) *rd*(7:0) ← B[*rb1*], *rd*(31:8) ← 0, *rb1* ← *rb1* + *rb2*

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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. Following data transfer, the address in the *rb* register is incremented by 1.

(2) Extension 1

```
ext     imm13         ; = sign13
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. Following data transfer, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,095.

(3) Extension 2

```
ext     imm13         ; = sign26(25:13)
ext     imm13         ; = sign26(12:0)
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. Following data transfer, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,431.

(4) Extension 3

```
ext     %rb2
ld.ub   %rd,[%rb1]+   ; memory address = rb1, rb1 ← rb1 + rb2
```

The byte data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The *rb1* register contains the memory address to be accessed. Following data transfer, the *rb2* register is added to the address in the *rb1* register, with the result stored in *rb1*. The range of *rb2* is -2,147,483,648 to +2,147,483,647.

ld.ub *%rd*, [%dp + *imm6*]

Function

Unsigned byte data transfer

Standard)	*rd*(7:0) ← B[dp + *imm6*], *rd*(31:8) ← 0
Extension 1)	*rd*(7:0) ← B[dp + *imm19*], *rd*(31:8) ← 0
Extension 2)	*rd*(7:0) ← B[dp + *imm32*], *rd*(31:8) ← 0
Extension 3)	Unusable

Code

15–12	11–10	9–4	3–0	
1 1 1 0	0 1	*imm6*	*r d*	0xE4__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.ub   %rd,[%dp + imm6]     ; memory address = dp + 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 DP 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,[%dp + imm6]     ; memory address = dp + imm19,
                             ; imm6 ← imm19(5:0)
```

The `ext` instruction extends the displacement to a 19-bit quantity. As a result, the content of the DP 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,[%dp + imm6]     ; memory address = dp + imm32,
                             ; imm6 ← imm32(5:0)
```

The two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the DP 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,[%dp + 0x1]   ; r0 ← [dp + 0x41] zero-extended
```

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
Extension 3)	Unusable

Code

15			12	11	10	9	4	3	0
0	1	0	0	0	1	*imm6*		*r d*	

0x44__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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
Extension 3)	Unusable

Code

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

0xAD__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3) *rd*(15:0) ← H[*rb1* + *rb2*], *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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext     %rb2
ld.uh   %rd,[%rb1]    ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the content of the *rb1* register with that of the *rb2* register added comprises the memory address, the halfword data in which is transferred to the *rd* register. The contents of the *rb1* and *rb2* registers are 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)	*rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0, *rb* ← *rb* + *sign13*
Extension 2)	*rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0, *rb* ← *rb* + *sign26*
Extension 3)	*rd*(15:0) ← H[*rb1*], *rd*(31:16) ← 0, *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7–4	3–0	
0	0	1	0	1	1	0	1	r b	r d	0x2D__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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. Following data transfer, the address in the *rb* register is incremented by 2.

(2) Extension 1

```
ext     imm13         ; = sign13
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. Following data transfer, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,094.

(3) Extension 2

```
ext     imm13         ; = sign26(25:13)
ext     imm13         ; = sign26(12:0)
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. Following data transfer, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,430.

(4) Extension 3

```
ext     %rb2
ld.uh   %rd,[%rb1]+   ; memory address = rb1, rb1 ← rb1 + rb2
```

The halfword data in the specified memory location is transferred to the *rd* register after being zero-extended to 32 bits. The *rb1* register contains the memory address to be accessed. Following data transfer, the *rb2* register is added to the address in the *rb1* register, with the result stored in *rb1*. The range of *rb2* is -2,147,483,648 to +2,147,483,646.

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*, [%dp + *imm6*]

Function

Unsigned halfword data transfer

Standard) *rd*(15:0) ← H[dp + *imm6* × 2], *rd*(31:16) ← 0
Extension 1) *rd*(15:0) ← H[dp + *imm19*], *rd*(31:16) ← 0
Extension 2) *rd*(15:0) ← H[dp + *imm32*], *rd*(31:16) ← 0
Extension 3) Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.uh   %rd,[%dp + imm6]      ; memory address = dp + 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 DP 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,[%dp + imm6]      ; memory address = dp + imm19,
                              ; imm6 = imm19(5:0)
```

The `ext` instruction extends the displacement to a 19-bit quantity. As a result, the content of the DP 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,[%dp + imm6]      ; memory address = dp + imm32,
                              ; imm6 = imm32(5:0)
```

The two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the DP 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,[%dp + 0x2] ; r0 ← [dp + 0x42] zero-extended
```

ld.uh *%rd*, [%sp + *imm6*]

Function

Unsigned halfword data transfer

Standard)	*rd*(15:0) ← H[sp + *imm6* × 2], *rd*(31:16) ← 0
Extension 1)	*rd*(15:0) ← H[sp + *imm19*], *rd*(31:16) ← 0
Extension 2)	*rd*(15:0) ← H[sp + *imm32*], *rd*(31:16) ← 0
Extension 3)	Unusable

Code

15			12	11	10	9 4	3 0
0	1	0	0	1	1	*imm6*	*r d*

0x4C__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
0	0	1	0	1	1	1	0	*rs*	*rd*	0x2E__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0
1	0	1	0	0	1	0	0	*s s*	*r d*

0xA4__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Src: Register direct `%ss` = `%psr`, `%sp`, `%alr`, `%ahr`, `%lco`, `%lsa`, `%lea`, `%sor`, `%ttbr`, `%dp`, `%idir`, `%dbbr`, `%usp`, `%ssp`, `%pc`

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

CLK

One cycle

Description

(1) Standard

The content of a special 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,%psr   ; r0 ← psr
```

Caution

(1) The `%sp` referenced in this instruction is either the SSP or USP, depending on the operation mode.

User mode → USP
Supervisor mode → SSP

(2) 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.

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

Function

Word data transfer

Standard)	$rd \leftarrow W[rb]$
Extension 1)	$rd \leftarrow W[rb + imm13]$
Extension 2)	$rd \leftarrow W[rb + imm26]$
Extension 3)	$rd \leftarrow W[rb1 + rb2]$

Code

15			12	11			8	7–4	3–0	
0	0	1	1	0	0	0	0	*rb*	*rd*	0x30__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext   %rb2
ld.w  %rd,[%rb1]     ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the content of the *rb1* register with that of the *rb2* register added comprises the memory address, the word data in which is transferred to the *rd* register. The contents of the *rb1* and *rb2* registers are 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) *rd* ← W[*rb*], *rb* ← *rb* + *sign13*
Extension 2) *rd* ← W[*rb*], *rb* ← *rb* + *sign26*
Extension 3) *rd* ← W[*rb1*], *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7–4	3–0	
0	0	1	1	0	0	0	1	*rb*	*rd*	0x31__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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. Following data transfer, the address in the *rb* register is incremented by 4.

(2) Extension 1

```
ext    imm13           ; = sign13
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. Following data transfer, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,092.

(3) Extension 2

```
ext    imm13           ; = sign26(25:13)
ext    imm13           ; = sign26(12:0)
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. Following data transfer, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,428.

(4) Extension 3

```
ext    %rb2
ld.w   %rd,[%rb1]+     ; memory address = rb1, rb1 ← rb1 + rb2
```

The word data in the specified memory location is transferred to the *rd* register. The *rb1* register contains the memory address to be accessed. Following data transfer, the *rb2* register is added to the address in the *rb1* register, with the result stored in *rb1*. The range of *rb2* is -2,147,483,648 to +2,147,483,644.

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*, [%dp + *imm6*]

Function

Word data transfer

Standard)	*rd* ← W[dp + *imm6* × 4]
Extension 1)	*rd* ← W[dp + *imm19*]
Extension 2)	*rd* ← W[dp + *imm32*]
Extension 3)	Unusable

Code

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

0xF0__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.w   %rd,[%dp + imm6]      ; memory address = dp + imm6 × 4
```

The word data in the specified memory location is transferred to the *rd* register. The content of the current DP 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,[%dp + imm6]     ; memory address = dp + imm19,
                             ; imm6 = imm19(5:0)
```

The `ext` instruction extends the displacement to a 19-bit quantity. As a result, the content of the DP 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,[%dp + imm6]     ; memory address = dp + imm32,
                             ; imm6 = imm32(5:0)
```

The two `ext` instructions extend the displacement to a 32-bit quantity. As a result, the content of the DP 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*, [%sp + *imm6*]

Function

Word data transfer

Standard) *rd* ← W[sp + *imm6* × 4]
Extension 1) *rd* ← W[sp + *imm19*]
Extension 2) *rd* ← W[sp + *imm32*]
Extension 3) Unusable

Code

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

0x50__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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*
Extension 3)	Unusable

Code

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

0x6C__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	0	0	0	0	0	*r s*	*s d*	0xA0__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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`, `%lco`, `%lsa`, `%lea`, `%sor`, `%ttbr`, `%dp`, `%idir`, `%dbbr`, `%usp`, `%ssp`, `%pc`

CLK

One cycle (4 cycles only when a `ld.w %psr,%rs` instruction is executed)

Description

(1) Standard
The content of the *rs* register (word data) is transferred to a special 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  %sp,%r0    ; sp ← r0
```

Caution

The `%sp` referenced in this instruction is either the SSP or USP, depending on the operation mode.
User mode → USP
Supervisor mode → SSP

In user mode, the SSP cannot be altered.

ld.w [%*rb*], %*rs*

Function

Word data transfer

Standard)	W[*rb*] ← *rs*
Extension 1)	W[*rb* + *imm13*] ← *rs*
Extension 2)	W[*rb* + *imm26*] ← *rs*
Extension 3)	W[*rb1* + *rb2*] ← *rs*

Code

15	14	13	12	11	10	9	8	7–4	3–0	
0	0	1	1	1	1	0	0	*rb*	*rs*	0x3C__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

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.

(4) Extension 3

```
ext    %rb2
ld.w   [%rb1],%rs     ; memory address = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing, so the content of the *rs* register is transferred to the address indicated by the content of the *rb1* register with the *rb2* register added. The contents of the *rb1* and *rb2* registers are 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) W[*rb*] ← *rs*, *rb* ← *rb* + *sign13*
Extension 2) W[*rb*] ← *rs*, *rb* ← *rb* + *sign26*
Extension 3) W[*rb1*] ← *rs*, *rb1* ← *rb1* + *rb2*

Code

15			12	11			8	7	4	3	0
0	0	1	1	1	1	0	1	*r b*		*r s*	

0x3D__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.w   [%rb]+,%rs     ; memory address = rb, rb ← rb + 4
```

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.

(2) Extension 1

```
ext    imm13          ; = sign13
ld.w   [%rb]+,%rs     ; memory address = rb, rb ← rb + sign13
```

The content of the *rs* register is transferred to the specified memory location. The *rb* register contains the memory address to be accessed. Following data transfer, *sign13* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign13* is -4,096 to +4,092.

(3) Extension 2

```
ext    imm13          ; = sign26(25:13)
ext    imm13          ; = sign26(12:0)
ld.w   [%rb]+,%rs     ; memory address = rb, rb ← rb + sign26
```

The content of the *rs* register is transferred to the specified memory location. The *rb* register contains the memory address to be accessed. Following data transfer, *sign26* is added to the address in the *rb* register, with the result stored in *rb*. The range of *sign26* is -33,554,432 to +33,554,428.

(4) Extension 3

```
ext    %rb2
ld.w   [%rb1]+,%rs    ; memory address = rb1, rb1 = rb1 + rb2
```

The addressing mode changes to 3-operand register indirect addressing. The content of the *rs* register is transferred to the address indicated by the *rb1* register. Following data transfer, the *rb2* register is added to the content of the *rb1* register, with the result stored in the *rb1* register. The range of *rb2* is -2,147,483,648 to +2,147,483,644.

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 [%dp + *imm6*], %rs

Function

Word data transfer

Standard)	W[dp + *imm6* × 4] ← *rs*
Extension 1)	W[dp + *imm19*] ← *rs*
Extension 2)	W[dp + *imm32*] ← *rs*
Extension 3)	Unusable

Code

15	14	13	12	11	10	9–4	3–0	
1	1	1	1	1	1	*imm6*	*r s*	0xFC__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One cycle

Description

(1) Standard

```
ld.w    [%dp + imm6],%rs       ; memory address = dp + imm6 × 4
```

The content of the *rs* register is transferred to the specified memory location. The content of the current DP 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    [%dp + imm6],%rs       ; memory address = dp + 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 DP 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    [%dp + imm6],%rs       ; memory address = dp + 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 DP with the 32-bit immediate *imm32* added. Make sure the *imm6* specified here resides on a word boundary (two least significant bits = 0).

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*
Extension 3)	Unusable

Code

15			12	11	10	9–4	3–0	
0	1	0	1	1	1	*imm6*	*r s*	0x5C__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

One 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).

loop %rc, %ra

Function

Loop execution

Standard)	Execute pc + 2 to *ra*, *rc* + 1 times
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	1	1	0	0	1	*ra*	*rc*	0xB9__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	↔	–	–	–	–	–	–	–	–	–	–	–

Mode

Addr: Register direct `%ra` = `%r0` to `%r15`
Count: Register direct `%rc` = `%r0` to `%r15`

CLK

Five cycles

Description

This instruction directs that a range of instructions, from the instruction next to the `loop` to the instruction at the absolute address specified by the *ra* register, be executed a number of times equal to the count in the *rc* register plus 1. When the `loop` instruction is executed, the LM flag (bit 29) in the PSR is set to 1, indicating that loop execution is in progress.
The start address for the instructions to be repeated is stored in the special register LSA, and the end address is stored in the LEA register. The number of times execution is to be repeated is stored in the LCO register. When the execution address and the LEA match, the LCO is decremented by 1 and the LSA is loaded into the PC, causing the program flow to jump to the start address. When the LCO value reaches 0, execution of the loop finishes and the LM flag (bit 29) in the PSR is cleared to 0, with the instructions following the LEA address executed normally.
To stop execution of a loop and return to normal instruction execution, the LM flag (bit 29) in the PSR must be cleared to 0 in the program.
When the value specified for the loop execution count, *rc*, is 1, the program flow returns from the LEA to the LSA once. In other words, the range of instructions from the LEA to the LSA is executed twice.

Example

When r0 = 2, r1 = end

```
     loop  %r0,%r1            ; loop start
     ld.w  %r2,[%r3]+         ; copy data
end: ld.w  [%r4]+,%r2         ;   [%r3] to [%r4]
```

Three words of data are copied.

Caution

As interrupts are accepted even during loop execution, if a `loop` instruction is to be executed in the interrupt handler routine, save the LSA, LEA, and LCO registers to the stack in order to protect the register data. Do not use this instruction in the debug exception or MMU exception handler routines.

loop %rc, imm4

Function

Loop execution

Standard)	Execute pc + 2 to pc + 2 + *imm4* × 2, *rc* + 1 times
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	1	1	1	0	1	0	*imm4*	*rc*	0xBA__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	↔	–	–	–	–	–	–	–	–	–	–	–

Mode

Addr: Unsigned immediate *imm4*
Count: Register direct `%rc` = `%r0` to `%r15`

CLK

Five cycles

Description

This instruction directs that a range of instructions, from the instruction next to the `loop` to the instruction at the address indicated by twice the value of unsigned immediate *imm4* with PC + 2 added, be executed a number of times equal to the count in the *rc* register plus 1. When the `loop` instruction is executed, the LM flag (bit 29) in the PSR is set to 1, indicating that loop execution is in progress.
The start address for the instructions to be repeated is stored in the special register LSA, and the end address is stored in the LEA register. The number of times execution is to be repeated is stored in the LCO register.
When the execution address and the LEA match, the LCO is decremented by 1 and the LSA is loaded into the PC, causing the program flow to jump to the start address. When the LCO value reaches 0, execution of the loop finishes and the LM flag (bit 29) in the PSR is cleared to 0, with the instructions following the LEA address executed normally.
To stop execution of a loop and return to normal instruction execution, the LM flag (bit 29) in the PSR must be cleared to 0 in the program.
If the value specified for the loop execution count, *rc*, is 1, the program flow returns from the LEA to the LSA once. In other words, the range of instructions from the LEA to the LSA is executed twice.

Example

When r1 = 3

```
loop  %r1,1            ; loop start
ld.w  %r2,[%r3]+       ; copy data
ld.w  [%r4]+,%r2       ;   [%r3] to [%r4]
```

Four words of data are copied.

Caution

As interrupts are accepted even during loop execution, if a `loop` instruction is to be executed in the interrupt handler routine, save the LSA, LEA, and LCO registers to the stack in order to protect the register data. Do not use this instruction in the debug exception or MMU exception handler routines.

loop *imm4(count)*, *imm4(addr)*

Function

Loop execution

Standard)	Execute pc + 2 to pc + 2 + *imm4(addr)* × 2, *imm4(count)* + 1 times
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	1	1	0	1	1	*imm4(addr)*	*imm4(count)*	0xBB__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	↔	–	–	–	–	–	–	–	–	–	–	–

Mode

Addr: Unsigned immediate *imm4*
Count: Unsigned immediate *imm4*

CLK

Five cycles

Description

This instruction directs that a range of instructions, from the instruction next to the `loop` to the instruction at the address indicated by twice the value of unsigned immediate *imm4(addr)* with PC + 2 added, be executed a number of times equal to the count indicated by the unsigned *imm4(count)* plus 1. When the `loop` instruction is executed, the LM flag (bit 29) in the PSR is set to 1, indicating that loop execution is in progress.
The start address for the instructions to be repeated is stored in the special register LSA, and the end address is stored in the LEA register. The number of times execution is to be repeated is stored in the LCO register.
When the execution address and the LEA match, the LCO is decremented by 1 and the LSA is loaded into the PC, causing the program flow to jump to the start address. When the LCO value reaches 0, execution of the loop finishes and the LM flag (bit 29) in the PSR is cleared to 0, with the instructions following the LEA address executed normally.
To stop execution of a loop and return to normal instruction execution, the LM flag (bit 29) in the PSR must be cleared to 0 in the program.
When the loop execution count, *imm4(count)*, is set to 1, the program flow returns from the LEA to the LSA once. In other words, the range of instructions from the LEA to the LSA is executed twice.

Example

```
loop   7,1               ; loop start
ld.w   %r2,[%r3]+        ; copy data
ld.w   [%r4]+,%r2        ;   [%r3] to [%r4]
```

Eight words of data are copied.

Caution

As interrupts are accepted even during loop execution, if a `loop` instruction is to be executed in the interrupt handler routine, save the LSA, LEA, and LCO registers to the stack in order to protect the register data. Do not use this instruction in the debug exception or MMU exception handler routines.

mac %rs

Function

Multiply-accumulate operation

Standard) "{ahr, alr} ← {ahr, alr} +H[<*rs*+1>] × H[<*rs*+2>], <*rs*+1> ← <*rs*+1> + 2, <*rs*+2> ← <*rs*+2> + 2" × *rs* times

Extension 1) Unusable

Extension 2) Unusable

Extension 3) Unusable

Code

15			12	11			8	7			4	3			0	
1	0	1	1	0	0	1	0		*rs*			0	0	0	0	0xB2_0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

Mode

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

CLK

2 + N × 2 cycles

Description

The `mac %rs` instruction executes the operation "{AHR, ALR} ← {AHR, ALR} + H[<*rs*+1>]+ × H[<*rs*+2>]+" (64 bits + 16 bits × 16 bits) a number of times equal to the count specified by the *rs* register.

The *rs* register is used as a counter to count the number of times the operation is performed, with the count decremented for each operation performed. The `mac` instruction finishes when the *rs* register reaches 0. Therefore, operation can be repeated as many times as 2^{32} - 1. If the `mac` instruction is executed after 0 is set in the *rs* register, no multiply-accumulate operation will be performed, nor will the AHR and ALR registers be altered. The *rs* register is not decremented; it remains at 0.

Note that <*rs*+1> and <*rs*+2> are two general-purpose registers that follow the *rs* register.

Example:

If the R0 resister is specified for *rs*, then <*rs*+1> = R1 register and <*rs*+2> = R2 register.

If the R15 resister is specified for *rs*, then <*rs*+1> = R0 register and <*rs*+2> = R1 register.

In the multiply-accumulate operation, the data in memory with its base address specified by these registers is treated as signed data. The base addresses in these registers are incremented each time operation is performed (by 2). The result of operation is obtained as signed 64-bit data, with the 32 high-order bits stored in the AHR and the 32 low-order bits stored in the ALR.

If the result of operation exceeds the range of values representable by signed 64 bits during multiply-accumulate operation, an overflow is assumed and the MO flag in the PSR is set to 1. Even in this case, the operation is continued until the count set in the *rs* register decrements to 0. The MO flag remains set until it is cleared in the software. Whether the result of operation is valid can be checked by reading the MO flag following execution of the `mac` instruction.

While the `mac` instruction is being executed, interrupts are accepted even during repetition. When the program branches to the interrupt handler routine, the address of the currently executed `mac` instruction is saved to the stack as the return address. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the CPU resumes execution of the suspended `mac` instruction. However, as the content of the *rs* register at that point is the remaining count, if the content of the *rs* register was altered in the interrupt handler routine, the *rs* register may not reflect the correct count. Similarly, if the <*rs*+1> or <*rs*+2> register value changed in the interrupt handler routine, the resumed `mac` instruction may not be executed correctly.

Example

```
mac  %r1  ; "{ahr,alr} ← {ahr,alr} + H[r2]+ × H[r3]+"
          ; is executed r1 times.
```

Caution

(1) The memory addresses specified by the <*rs*+1> and <*rs*+2> registers should respectively indicate those located at halfword boundaries. If an odd address is specified, an address misaligned exception is generated.

(2) When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

mac.hw *%rs*

Function

Multiply-accumulate operation

Standard) "{ahr, alr} ← {ahr, alr} + W[<*rs*+1>] × H[<*rs*+2>], <*rs*+1> ← <*rs*+1> + 4, <*rs*+2> ← <*rs*+2> + 2" × *rs* times

Extension 1) Unusable

Extension 2) Unusable

Extension 3) Unusable

Code

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

0x015_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

Mode

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

CLK

2 + N × 2 cycles

Description

The `mac.hw %rs` instruction executes the operation "{AHR, ALR} ← {AHR, ALR} + W[<*rs*+1>]+ × H[<*rs*+2>]+" (64 bits + 32 bits × 16 bits) a number of times equal to the count specified by the *rs* register.
The *rs* register is used as a counter to count the number of times the operation is performed, with the count decremented for each operation performed. The `mac.hw` instruction finishes when the *rs* register reaches 0. Therefore, operation can be repeated as many times as 2^{32} - 1. If the `mac.hw` instruction is executed after 0 is set in the *rs* register, no multiply-accumulate operation will be performed, nor will the AHR and ALR registers be altered. The *rs* register is not decremented; it remains at 0.
Note that <*rs*+1> and <*rs*+2> are two general-purpose registers that follow the *rs* register.
Example:
If the R0 resister is specified for *rs*, then <*rs*+1> = R1 register and <*rs*+2> = R2 register.
If the R15 resister is specified for *rs*, then <*rs*+1> = R0 register and <*rs*+2> = R1 register.

In the multiply-accumulate operation, the data in memory with its base address specified by these registers is treated as signed data. The base addresses in these registers are incremented each time operation is performed (by 4 and 2, respectively). The result of operation is obtained as signed 64-bit data, with the 32 high-order bits stored in the AHR and the 32 low-order bits stored in the ALR.
If the result of operation exceeds the range of values representable by signed 64 bits during multiply-accumulate operation, an overflow is assumed and the MO flag in the PSR is set to 1. Even in this case, the operation is continued until the count set in the *rs* register decrements to 0. The MO flag remains set until it is cleared in the software. Whether the result of operation is valid can be checked by reading the MO flag following execution of the `mac.hw` instruction.
While the `mac.hw` instruction is being executed, interrupts are accepted even during repetition. When the program branches to the interrupt handler routine, the address of the currently executed `mac.hw` instruction is saved to the stack as the return address. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the CPU resumes execution of the suspended `mac.hw` instruction. However, as the content of the *rs* register at that point is the remaining count, if the content of the *rs* register was altered in the interrupt handler routine, the *rs* register may not reflect the correct count. Similarly, if the <*rs*+1> or <*rs*+2> register value changed in the interrupt handler routine, the resumed `mac.hw` instruction may not be executed correctly.

Example

```
mac.hw  %r1     ; "{ahr,alr} ← {ahr,alr} + W[r2]+ × H[r3]+"
                ; is executed r1 times.
```

Caution

(1) The memory addresses specified by the <*rs*+1> and <*rs*+2> registers should respectively indicate those located at word and halfword boundaries. If an odd address is specified, an address misaligned exception is generated.

(2) When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

mac.w %rs

Function

Multiply-accumulate operation

Standard) “{ahr, alr} ← {ahr, alr} + W[<*rs*+1>] × W[<*rs*+2>], <*rs*+1> ← <*rs*+1> + 4, <*rs*+2> ← <*rs*+2> + 4” × *rs* times

Extension 1) Unusable

Extension 2) Unusable

Extension 3) Unusable

Code

15	14	13	12	11	10	9	8	7	6	5	4	3–0	
0	0	0	0	0	0	0	1	0	0	0	1	*rs*	0x011_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

Mode

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

CLK

3 + N × 2 cycles

Description

The `mac.w %rs` instruction executes the operation “{AHR, ALR} ← {AHR, ALR} + W[<*rs*+1>]+ × W[<*rs*+2>]+” (64 bits + 32 bits × 32 bits) a number of times equal to the count specified by the *rs* register. The *rs* register is used as a counter to count the number of times the operation is performed, with the count decremented for each operation performed. The `mac.w` instruction finishes when the *rs* register reaches 0. Therefore, operation can be repeated as many times as 2^{32} - 1. If the `mac.w` instruction is executed after 0 is set in the *rs* register, no multiply-accumulate operation will be performed, nor will the AHR and ALR registers be altered. The *rs* register is not decremented, it remains at 0.

Note that <*rs*+1> and <*rs*+2> are two general-purpose registers that follow the *rs* register.

Example:

If the R0 resister is specified for *rs*, then <*rs*+1> = R1 register and <*rs*+2> = R2 register.

If the R15 resister is specified for *rs*, then <*rs*+1> = R0 register and <*rs*+2> = R1 register.

In the multiply-accumulate operation, the data in memory with its base address specified by these registers is treated as signed data. The base addresses in these registers are incremented each time operation is performed (by 4). The result of operation is obtained as signed 64-bit data, with the 32 high-order bits stored in the AHR and the 32 low-order bits stored in the ALR.

If the result of operation exceeds the range of values representable by signed 64 bits during multiply-accumulate operation, an overflow is assumed and the MO flag in the PSR is set to 1. Even in this case, the operation is continued until the count set in the *rs* register decrements to 0. The MO flag remains set until it is cleared in the software. Whether the result of operation is valid can be checked by reading the MO flag following execution of the `mac.w` instruction.

While the `mac.w` instruction is being executed, interrupts are accepted even during repetition. When the program branches to the interrupt handler routine, the address of the currently executed `mac.w` instruction is saved to the stack as the return address. Therefore, when the interrupt handler routine is terminated by the `reti` instruction, the CPU resumes execution of the suspended `mac.w` instruction. However, as the content of the *rs* register at that point is the remaining count, if the content of the *rs* register was altered in the interrupt handler routine, the *rs* register may not reflect the correct count. Similarly, if the <*rs*+1> or <*rs*+2> register value changed in the interrupt handler routine, the resumed `mac.w` instruction may not be executed correctly.

Example

```
mac.w  %r1       ; "{ahr,alr} ← {ahr,alr} + W[r2]+ × W[r3]+"
                 ; is executed r1 times.
```

Caution

(1) The memory addresses specified by the <*rs*+1> and <*rs*+2> registers should each indicate those located at word boundaries. If a non-word-aligned address is specified, an address misaligned exception is generated.

(2) When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

mac1.h *%rd, %rs*

Function

Multiply-accumulate operation

Standard)	{ahr, alr} ← {ahr, alr} + *rd*(15:0) × *rs*(15:0)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

0xA7__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

Mode

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

CLK

One cycle

Description

The 16 low-order bits of the *rd* register and the 16 low-order bits of the *rs* register are multiplied together, and the product is added to the 64-bit register pair, AHR and ALR.
If the operation resulted in the 64-bit register pair AHR and ALR overflowing, the MO flag (bit 7) in the PSR is set to 1. The MO flag remains set until it is cleared in the software.

Example

```
mac1.h  %r1,%r2  ; {ahr,alr} ← r1[15:0] × r2[15:0] + {ahr,alr}
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

mac1.hw *%rd, %rs*

Function

Multiply-accumulate operation

Standard)	{ahr, alr} ← {ahr, alr} + *rd*(31:0) × *rs*(15:0)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

Mode

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

CLK

Two cycles

Description

The entire content of the *rd* register and the 16 low-order bits of the *rs* register are multiplied together, and the product is added to the 64-bit register pair, AHR and ALR.
If the operation resulted in the 64-bit register pair AHR and ALR overflowing, the MO flag (bit 7) in the PSR is set to 1. The MO flag remains set until it is cleared in the software.

Example

```
mac1.hw  %r1,%r2  ; {ahr,alr} ← r1[31:0] × r2[15:0] + {ahr,alr}
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

mac1.w *%rd, %rs*

Function

Multiply-accumulate operation

Standard)	{ahr, alr} ← {ahr, alr} + *rd* × *rs*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

0xB3__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

Mode

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

CLK

Two cycles

Description

The entire contents of the *rd* and the *rs* registers are multiplied together, and the product is added to the 64-bit register pair, AHR and ALR. If the operation resulted in the 64-bit register pair AHR and ALR overflowing, the MO flag (bit 7) in the PSR is set to 1. The MO flag remains set until it is cleared in the software.

Example

```
mac1.w  %r1,%r2         ; {ahr,alr} ← r1 × r2 + {ahr,alr}
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

macclr

Function

Clear AHR and ALR

Standard)	{ahr, alr} ← 0, MO ← 0
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3			0	
0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0x0190

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	0	–	–	–	–	–

Mode

–

CLK

One cycle

Description

(1) Standard

The AHR and the ALR registers and the MO flag (bit 7) in the PSR are cleared.
The `macclr` instruction is used to clear the AHR and ALR registers and the MO flag before a `mac1` or `mac` instruction is executed. It operates differently from the normal forwarding mechanism.
Therefore, if the `macclr` instruction is followed by any instruction that uses the AHR, ALR, and PSR as the source registers (e.g., an `ld` *`%rd`*`,`*`%ss`* or `pushs` *`%ss`* instruction) within two instructions, the `macclr` instruction may not be executed correctly.

When the LC flag (bit 17) = 1, not just the ALR register but the R4 register is cleared. When the HC flag (bit 16) = 1, not just the AHR register but the R5 register is also cleared.

(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.

mirror *%rd, %rs*

Function

Mirror

Standard) *rd*(31:24) ← *rs*(24:31), *rd*(23:16) ← *rs*(16:23), *rd*(15:8) ← *rs*(8:15), *rd*(7:0) ← *rs*(0:7)

Extension 1) Unusable

Extension 2) Unusable

Extension 3) Unusable

Code

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

0x96__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

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

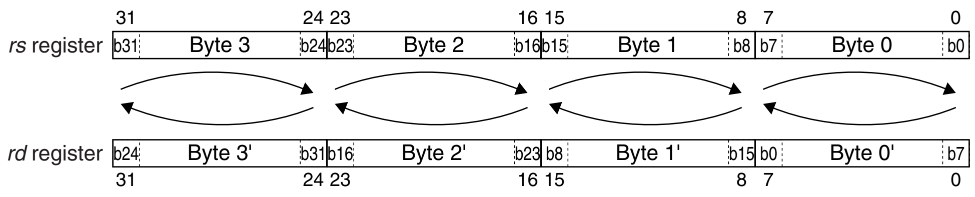
CLK

One cycle

Description

(1) Standard

Swaps the bit order of the *rs* register high and low in byte data units and loads the results to the *rd* register.



(2) Delayed instruction

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

Example

When r1 = 0x88442211

```
mirror   %r0,%r1        ; r0 ← 0x11224488
```

Mirror operation for 32-bit data (when r1 = 0x44332211)

```
swap     %r1,%r1        ; r1 ← 0x11223344
mirror   %r1,%r1        ; r1 ← 0x8844CC22
```

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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	0	0	0	1	0	*rs*	*rd*	0xA2__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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 *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.

(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.

Example

```
mlt.h  %r0,%r1          ; alr ← r0(15:0) × r1(15:0)
                        ; signed multiplication
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register.

mlt.hw *%rd, %rs*

Function

Signed 32-bit × 16-bit multiplication

Standard)	{ahr, alr} ← *rd*(31:0) × *rs*(15:0)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	0	0	0	1	1	*rs*	*rd*	0xA3__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Two cycles

Description

The entire content of the *rd* register and the 16 low-order bits of the *rs* register are multiplied together, and the 48-bit product resulting from the operation is loaded into the AHR and ALR register pair after being sign-extended to 64 bits.

Example

```
mlt.hw  %r1,%r2          ; {ahr,alr} ← r1(31:0) × r2(15:0)
                         ; signed multiplication
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

mlt.w *%rd, %rs*

Function

Signed 32-bit × 32-bit multiplication

Standard)	{ahr, alr} ← *rd* × *rs*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Two 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
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

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
Extension 3)	Unusable

Code

15			12	11			8	7 4	3 0	
1	0	1	0	0	1	1	0	*rs*	*rd*	0xA6__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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 *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.

(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.

Example

```
mltu.h  %r0,%r1        ; alr ← r0(15:0) × r1(15:0)
                       ; unsigned multiplication
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register.

mltu.w *%rd, %rs*

Function

Unsigned 32-bit × 32-bit multiplication

Standard)	{ahr, alr} ← *rd* × *rs*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Two 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
```

Caution

When the LC flag (bit 17) = 1, the data written to the ALR register is simultaneously written to the R4 register. When the HC flag (bit 16) = 1, the data written to the AHR register is simultaneously written to the R5 register.

nop

Function No operation

Standard)	No operation
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	∗1	↔	↔

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.

∗1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `and`, `or`, and `xor` instructions, refer to the description of each instruction.)

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*
Extension 3) Unusable

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	1	1	1	*sign6*	*r d*	0x7C__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	∗1	↔	↔

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.

∗1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `and`, `or`, and `xor` instructions, refer to the description of each instruction.)

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*
Extension 3)	*rd* ← *rs1* | *rs2*

Code

15			12	11			8	7–4	3–0	
0	0	1	1	0	1	1	0	*rs*	*rd*	0x36__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

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) Extension 3

```
ext   %rs2
or    %rd,%rs1         ; rd ← rs1 | rs2
```

The content of the *rs1* register and the register *rs2* specified by the `ext` instruction are logically OR'ed, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(5) 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.

*1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `and`, `not`, and `xor` instructions, refer to the description of each instruction.)

Example

```
(1) or    %r0,%r0          ; r0 = r0 | r0

(2) ext   0x1
    ext   0x1fff
    or    %r1,%r2          ; r1 = r2 | 0x00003fff

(3) ext   %r5
    or    %r3,%r4          ; r3 = r4 | r5
```

or %rd, sign6

Function

Logical OR

Standard)	*rd* ← *rd* | *sign6*
Extension 1)	*rd* ← *rd* | *sign19*
Extension 2)	*rd* ← *rd* | *sign32*
Extension 3)	Unusable

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	1	0	1	*sign6*	*r d*	0x74__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	∗1	↔	↔

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.

∗1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `and`, `not`, and `xor` instructions, refer to the description of each instruction.)

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
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

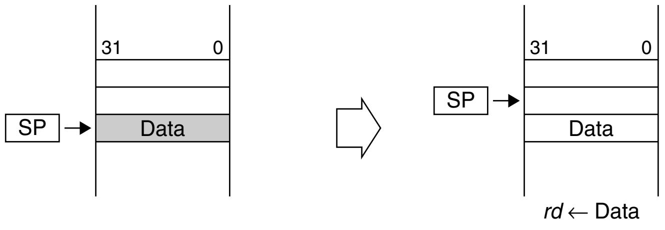
CLK

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
Extension 3)	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

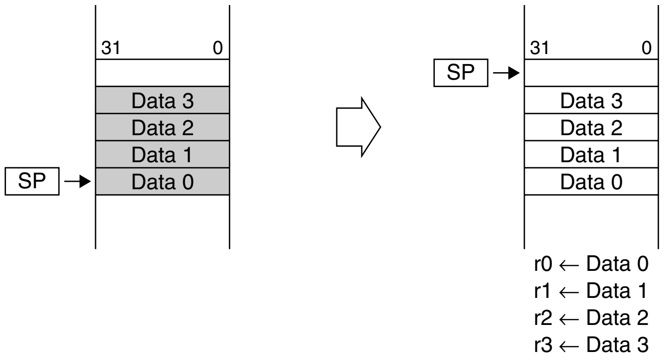
RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

Mode Register direct `%rd` = `%r0` to `%r15`

CLK N 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
```

Caution If the PM flag (bit 28) in the PSR = 0 when the `popn` instruction is executed, the register number in the register field of the instruction word is referenced; if PM = 1, the register number stored in RC[3:0] of the PSR is referenced. Successive pop operations are performed until a register that matches this referenced register is reached.

pops *%sd*

Function

Pop

Standard)	When *sd* = psr or sp: *sd* ← W[sp], sp ← sp + 4
	When *sd* = alr to pc: "*sN* ← W[sp], sp ← sp + 4" repeated for *sN* = alr to *sd*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

Mode

Register direct `%sd` = `%psr`, `%sp`, `%alr`, `%ahr`, `%lco`, `%lsa`, `%lea`, `%sor`, `%ttbr`, `%dp`, `%idir`, `%dbbr`, `%usp`, `%ssp`, `%pc`

CLK

N cycles, where N = number of registers to be restored

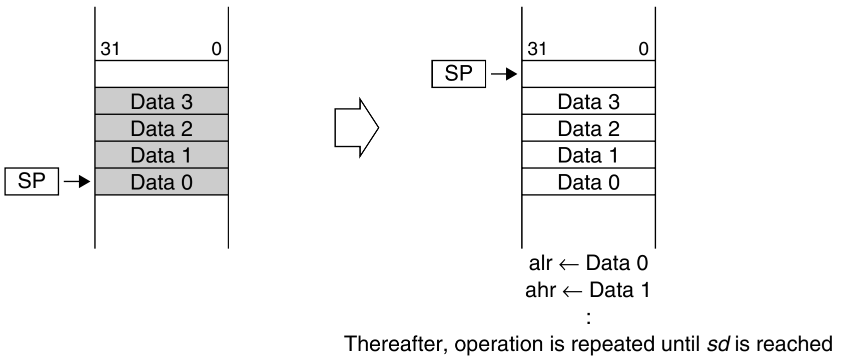
Description

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 PSR or SP register
The PSR or SP register is restored from the stack singly. The word data at the address indicated by the current SP is restored to the *sd* register, and the SP is incremented by an amount equivalent to 1 word (4 bytes).

(2) When the *sd* register is the ALR, AHR, LCO, LSA, LEA, SOR, DP, IDIR, DBBR, USP, SSP, or PC 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 target special register to be restored is altered in order of register number, including nonexistent registers. This operation is repeated until a register that matches *sd* is reached. The *sd* must be the same register as specified in the corresponding `pushs` instruction.

Stack operation when `pops %sd` (where `%sd` = `%alr` to `%pc`) is executed



Example

```
(1) pops   %sp     ; sp is restored singly
(2) pops   %lea    ; registers are restored in order of alr, ahr,
                   ; lco, lsa, and lea
```

Caution

(1) If the IDIR, DBBR, USP, SSP, or PC register is specified as the *sd* register, memory read and SP incrementation are performed uniformly for the IDIR, DBBR, USP, SSP, and the special register number #12, which is nonexistent register. At this time, the read-out data is not reflected in any register.

(2) When a pop operation is performed for the SP, USP, or SSP register, although memory read and SP incrementation are performed normally, the saved data that has been popped off the stack is not written back to the register.

(3) When a pop operation is performed for the PC register, although the SP is incremented normally, the PC register is not altered.

(4) If the PM flag (bit 28) in the PSR = 0 when the `pops` instruction is executed, the register number in the register field of the instruction word is referenced; if PM = 1, the register number stored in RC[3:0] of the PSR is referenced. Successive pop operations are performed until a register that matches this referenced register is reached.

psrclr *imm5*

Function

Clear PSR bit

Standard)	psr ← psr & !*imm5*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7		5	4				0	
1	0	1	1	1	1	1	1	1	0	0	*imm5*					0xBF8_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔

Mode

Immediate

CLK

Four cycles

Description

(1) Standard

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, ... 30, and 31 representing bits 0, 1, 2, ... 30, and 31, respectively.

(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.

Example

```
psrclr  2        ; V ← 0 (V flag cleared)
```

psrset *imm5*

Function

Set PSR bit

Standard)	psr ← psr | *imm5*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7		5	4–0	
1	0	1	1	1	1	1	1	0	1	0	*imm5*	0xBF4_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔

Mode

Immediate

CLK

Four cycles

Description

(1) Standard

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, ... 30, and 31 representing bits 0, 1, 2, ... 30, and 31, respectively.

(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.

Example

```
psrset  12      ; SV ← 1 (SV flag set)
```

push %*rs*

Function

Push

Standard)	sp ← sp - 4, W[sp] ← *rs*
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3			0
0	0	0	0	0	0	0	0	0	0	0	1	*r s*			

0x001_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

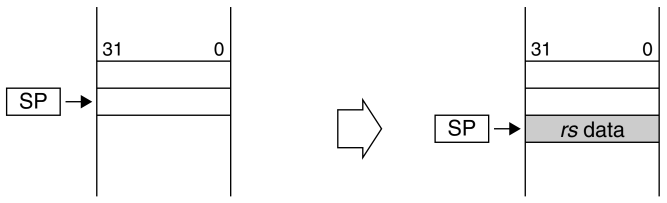
CLK

One cycle

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
Extension 3)	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

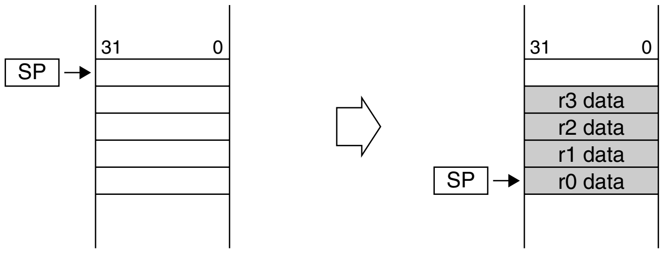
RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

Mode Register direct `%rs` = `%r0` to `%r15`

CLK N 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
```

Caution If the PM flag (bit 28) in the PSR = 0 when the `pushn` instruction is executed, the register number in the register field of the instruction word is referenced; if PM = 1, the register number stored in RC[3:0] of the PSR is referenced. Successive push operations are performed beginning with the register that matches this referenced register.

pushs *%ss*

Function

Push

Standard)	When *ss* = psr or sp: sp ← sp - 4, W[sp] ← *ss*
	When *ss* = alr to pc: "sp ← sp - 4, W[sp] ← *sN*" repeated for *sN* = *ss* to alr
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

Mode

Register direct `%ss` = `%psr`, `%sp`, `%alr`, `%ahr`, `%lco`, `%lsa`, `%lea`, `%sor`, `%ttbr`, `%dp`, `%idir`, `%dbbr`, `%usp`, `%ssp`, `%pc`

CLK

N cycles, where N = number of registers to be saved

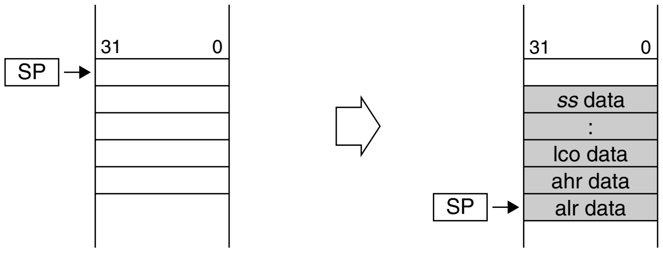
Description

Save the data of special registers to the stack.

(1) When the *ss* register is the PSR or SP register
The PSR or SP register is saved to the stack singly.
The current SP is decremented by an amount equivalent to 1 word (4 bytes), and the content of the *ss* register is saved to that address.

(2) When the *ss* register is the ALR, AHR, LCO, LSA, LEA, SOR, DP, IDIR, DBBR, USP, SSP, or PC register
The current SP is decremented by an amount equivalent to 1 word (4 bytes), and the content of the *ss* register is saved to that address. Next, the target special register to be saved is altered in order of register number, including nonexistent registers. This operation is repeated until the ALR register is reached.

Stack operation when `pushs %ss` (where `%ss` = `%alr` to `%pc`) is executed



Operation repeated from *ss* until the alr register is reached

Example

```
(1) pushs  %sp    ; sp is saved singly
(2) pushs  %lea   ; registers are saved in order of lea, lsa, lco,
                  ; ahr, and alr
```

Caution

(1) If the IDIR, DBBR, USP, SSP, or PC register is specified as the *ss* register, memory write and SP decrementation are performed uniformly for the IDIR, DBBR, USP, SSP, and the special register number #12 which is nonexistent register. In this case, the data written to memory is indeterminate.

(2) The SP, SSP, or USP register contents saved by the `pushs` instruction will not be restored to the register. The `pops %sd` instruction only increments the SP value.

(3) If the PM flag (bit 28) in the PSR = 0 when the `pushs` instruction is executed, the register number in the register field of the instruction word is referenced; if PM = 1, the register number stored in RC[3:0] of the PSR is referenced. Successive push operations are performed beginning with the register that matches this referenced register.

repeat *%rc*

Function

Repeat execution

Standard)	pc + 2 executed *rc* + 1 times
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3–0	
0	0	0	0	0	0	1	0	1	0	0	1	*rc*	0x029_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

CLK

Four cycles

Description

Direct that the instruction next to the `repeat` be executed a number of times equal to the count in the *rc* register plus 1. When the `repeat` instruction is executed, the RM flag (bit 30) in the PSR is set to 1, indicating that repeat execution is in progress.
The number of times operation is to be repeated is stored in the LCO register, and the address of the target instruction to be repeated (i.e., the instruction next to the `repeat`) is stored in the LSA register. When the target instruction to be repeated is executed, the PC and the value of the LSA register match, so the PC is fixed and the same instruction is executed repeatedly. The LCO is decremented by 1 each time the target repeat instruction is executed, and the same instruction is executed until the LCO decrements to 0.
When LCO = 0, the RM flag (bit 30) in the PSR is cleared to 0, at which time execution of repeat finishes.
If the value specified for the repeat count, *rc*, is 1, the instruction next to the `repeat` is executed twice.

Example

When r0 = 99

```
repeat %r0              ; repeat start
ld.w   [%r1]+,%r2       ; fill
```

100 words of data are filled with the data r2, beginning with the address indicated by r1.

Caution

As interrupts are accepted even during repeat execution, if any instruction that uses the LCO and LSA registers (e.g., `loop` or `repeat` instruction) is to be executed in the interrupt handler routine, save the LCO and LSA registers to the stack or equivalent to protect the register data.
Do not use this instruction in the debug exception or MMU exception handler routines.

repeat *imm4*

Function

Repeat execution

Standard)	pc + 2 executed *imm4* + 1 times
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

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

0x02D_

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	–	–	–	–	–	–	–	–	–	–	–	–

Mode

Unsigned immediate

CLK

Four cycles

Description

Direct that the instruction next to the `repeat` be executed a number of times equal to the value of *imm4* plus 1. When the `repeat` instruction is executed, the RM flag (bit 30) in the PSR is set to 1, indicating that repeat execution is in progress.

The number of times operation is to be repeated is stored in the LCO register, and the address of the target instruction to be repeated (i.e., the instruction next to the `repeat`) is stored in the LSA register. When the target instruction to be repeated is executed, the PC and the value of the LSA register match, so the PC is fixed and the same instruction is executed repeatedly. The LCO is decremented by 1 each time the target repeat instruction is executed, and the same instruction is executed until the LCO decrements to 0.

When LCO = 0, the RM flag (bit 30) in the PSR is cleared to 0, at which time execution of repeat finishes.

If the value specified for the repeat count, *imm4*, is 1, the instruction next to the `repeat` is executed twice.

Example

```
repeat 7                ; repeat start
ld.w   [%r1]+,%r2       ; fill
```

Eight words of data are filled with the data r2, beginning with the address indicated by r1.

Caution

As interrupts are accepted even during repeat execution, if any instruction that uses the LCO and LSA registers (e.g., `loop` or `repeat` instruction) is to be executed in the interrupt handler routine, save the LCO and LSA registers to the stack or the equivalent to protect the register data.

Do not use this instruction in the debug exception or MMU exception handler routines.

ret / ret.d

Function

Return from subroutine

Standard)	pc ← W[sp], sp ← sp + 4
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

–

CLK

`ret`	Five cycles
`ret.d`	Four 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[0xC (or 0x6000C)], pc ← W[0x8 (or 0x60008)]
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	0	–	–	–	–	–	–	–

Mode

–

CLK

Six 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[ssp + 4], psr ← W[ssp], ssp ← ssp + 8
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔

Mode

–

CLK

Six 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 SSP is incremented by an amount equivalent to 2 words.

Example

```
reti            ; Return from a trap handler routine
```

Caution

(1) When the `reti` instruction is executed, the CPU uses SSP as the stack pointer regardless of its operation mode.

(2) Depending on the S1C33 model, a few clock cycles are expended until the interrupt request signal to the CPU is negated after the cause-of-interrupt flag in the ITC (interrupt controller) has been reset (using a `ld` instruction). This may cause the process to be unable to return from the interrupt handler routine if the `reti` instruction is executed immediately following the `ld` instruction that resets the cause-of-interrupt flag. To avoid this problem, (a) reset the cause-of-interrupt flag as far in advance of executing the `reti` instruction as possible, or (b) insert an instruction to read the cause-of-interrupt flag register between resetting the cause-of-interrupt flag and executing `reti` instruction.

retm

Function

Return from an MMU exception handler routine

Standard)	r0 ← W[0x1C], pc ← W[0x18]
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7			4	3			0	
0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0x06C0

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	0	–	–	–	–	–	–

Mode

–

CLK

Six cycles

Description

Restore the contents of the R0 and PC that were saved to the MMU exception memory space when an MMU exception occurred to the respective registers, and return from the MMU exception handler routine.

Example

```
retm              ; Return from an MMU exception 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
Extension 3)	*rd* ← *rs1* << *rs2*

Code

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	0	1	1	1	0	1	*rs*	*rd*	0x9D__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

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

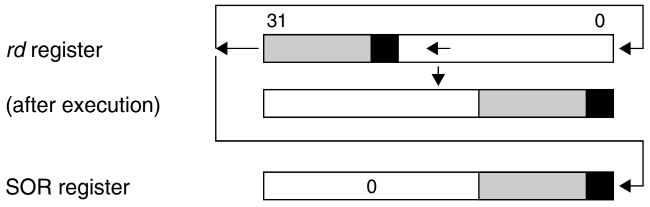
CLK

One cycle

Description

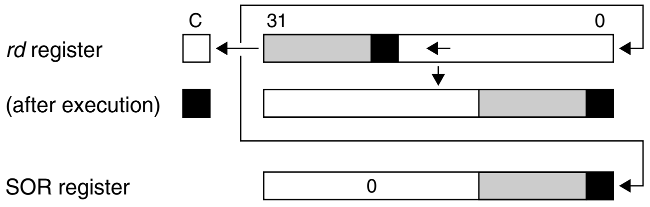
(1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in standard mode, except that the C flag is involved. The bit that has been shifted out from the most significant bit position is placed in the C flag of the PSR and in the least significant bit of the *rd* register.



The V flag changes state depending on the status of the C and Z flag upon completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext  %rs1
rl   %rd,%rs2
```

The *rs1* register is rotated to the left as many bits as specified by *rs2*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be rotated is given by the `ext %rs1` instruction.
The value in the most significant bit of the *rd* register is placed in the least significant bit.
The shifted-out bit can be read from the SOR 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 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
Extension 3)	*rd* ← *rs* << *imm5*

Code

When *imm5*(4) = 0, rotated to the left by 0 to 15 bits

15			12	11			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			12	11			8	7–4	3–0	
0	0	1	1	0	1	1	1	*imm5*(3:0)	*r d*	0x37__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

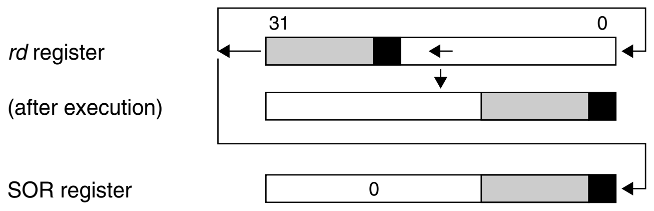
CLK

One cycle

Description

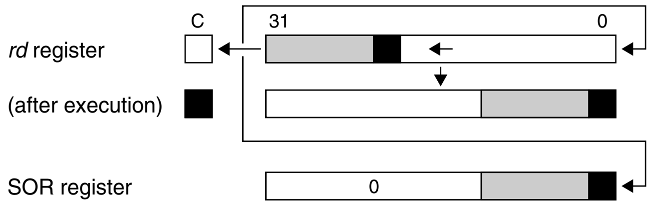
(1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The shifted-out bit can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in standard mode, except that the C flag is involved. The bit that has been shifted out from the most significant bit position is placed in the C flag of the PSR and in the least significant bit of the *rd* register.



The V flag changes state depending on the status of the C and Z flags upon completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs
rl    %rd,imm5
```

The *rs* register is rotated to the left as many bits as specified by *imm5*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be rotated is given by the `ext %rs` instruction.
The value in the most significant bit of the *rd* register is placed in the least significant bit.
The shifted-out bit can be read from the SOR 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 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
Extension 3)	*rd* ← *rs1* >> *rs2*

Code

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	0	1	1	0	0	1	*rs*	*rd*	0x99__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

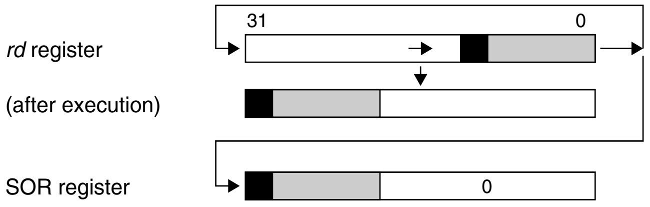
The C and V flags change when the SE flag (bit 20) in the PSR = 1.

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

CLK One cycle

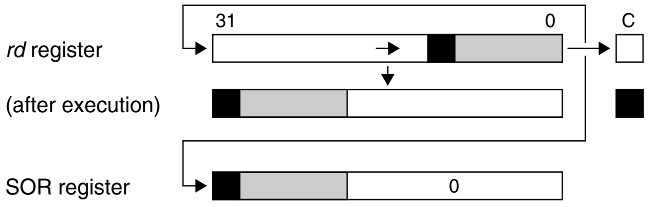
Description (1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode, except that the C flag is involved. The bit that has been shifted out from the least significant bit position is placed in the C flag of the PSR and in the most significant bit of the *rd* register.



The V flag changes state depending on the status of the C and Z flags upon completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext  %rs1
rr   %rd,%rs2
```

The *rs1* register is rotated to the right as many bits as specified by *rs2*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be rotated is given by the `ext %rs1` instruction.
The value in the least significant bit of the *rd* register is placed in the most significant bit.
The shifted-out bit can be read from the SOR 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 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
Extension 3)	*rd* ← *rs* >> *imm5*

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

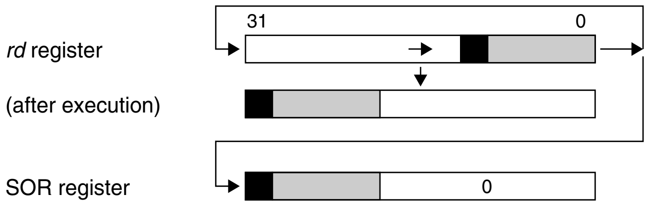
One cycle

Description

(1) Standard: When the SE flag (bit 20) in the PSR = 0

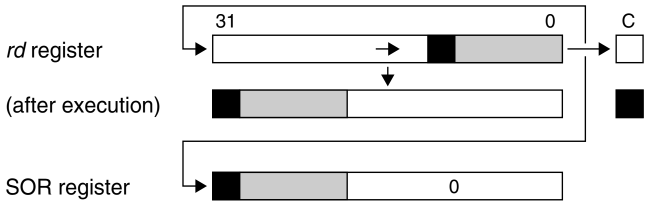
The *rd* register is rotated as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. The value in the least significant bit of the *rd* register is placed in the most significant bit.

The shifted-out bit can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode, except that the C flag is involved. The bit that has been shifted out from the least significant bit position is placed in the C flag of the PSR and in the most significant bit of the *rd* register.



The V flag changes state depending on the status of the C and Z flags upon completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs
rr    %rd,imm5
```

The *rs* register is rotated to the right as many bits as specified by *imm5*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be rotated is given by the `ext %rs` instruction.
The value in the least significant bit of the *rd* register is placed in the most significant bit.
The shifted-out bit can be read from the SOR 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 included.

sat.b *%rd, %rs*

Function

Signed saturation (8 bits)

Standard)	*rd* ← *rs* if -128 ≤ *rs* ≤ +127;
	rd ← 0xFFFFFF80 if *rs* < -128;
	rd ← 0x0000007F if *rs* > +127
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	0	1	1	1	1	0	*rs*	*rd*	0x9E__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Perform signed 8-bit saturation processing.
The content of the *rs* register is tested, and the result is stored in the *rd* register.

rs condition	Processing	S flag
-128 ≤ *rs* ≤ +127	*rs* → *rd*	–
rs < -128	0xFFFFFF80 → *rd*	1
rs > +127	0x0000007F → *rd*	1

(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 = 0x555
```
sat.b   %r2,%r1       ; 0x0000007F → r2
```
When r1 = 0xFEDBA987
```
sat.b   %r2,%r1       ; 0xFFFFFF80 → r2
```

sat.h *%rd, %rs*

Function

Signed saturation (16 bits)

Standard) *rd* ← *rs* if -32,768 ≤ *rs* ≤ +32,767;
rd ← 0xFFFF8000 if *rs* < -32,768;
rd ← 0x00007FFF if *rs* > +32,767

Extension 1) Unusable
Extension 2) Unusable
Extension 3) Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	1	0	1	1	0	*r s*	*r d*	0xB6__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Perform signed 16-bit saturation processing.
The content of the *rs* register is tested, and the result is stored in the *rd* register.

rs condition	Processing	S flag
-32,768 ≤ *rs* ≤ +32,767	*rs* → *rd*	–
rs < -32,768	0xFFFF8000 → *rd*	1
rs > +32,767	0x00007FFF → *rd*	1

(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 = 0x555

```
sat.h   %r2,%r1       ; 0x00000555 → r2
```

When r1 = 0xFEDBA987

```
sat.h   %r2,%r1       ; 0xFFFF8000 → r2
```

sat.ub *%rd, %rs*

Function

Unsigned saturation (8 bits)

Standard) *rd* ← *rs* if *rs* ≤ 255;
rd ← 0x000000FF if *rs* > 255

Extension 1) Unusable
Extension 2) Unusable
Extension 3) Unusable

Code

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	0	1	1	1	1	1	*rs*	*rd*	0x9F__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Perform unsigned 8-bit saturation processing.
The content of the *rs* register is tested, and the result is stored in the *rd* register.

rs condition	Processing	S flag
rs ≤ 255	*rs* → *rd*	–
rs > 255	0x000000FF → *rd*	1

(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 = 0x555
```
sat.ub  %r2,%r1     ; 0x000000FF → r2
```
When r1 = 0xFEDBA987
```
sat.ub  %r2,%r1     ; 0x000000FF → r2
```

sat.uh *%rd, %rs*

Function

Unsigned saturation (16 bits)

Standard) *rd* ← *rs* if *rs* ≤ 65,535;
rd ← 0x0000FFFF if *rs* > 65,535

Extension 1) Unusable
Extension 2) Unusable
Extension 3) Unusable

Code

15			12	11			8	7–4	3–0	
1	0	1	1	0	1	1	1	*rs*	*rd*	0xB7__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Perform unsigned 16-bit saturation processing.
The content of the *rs* register is tested, and the result is stored in the *rd* register.

rs condition	Processing	S flag
rs ≤ 65,535	*rs* → *rd*	–
rs > 65,535	0x0000FFFF → *rd*	1

(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 = 0x555

```
sat.uh  %r2,%r1     ; 0x00000555 → r2
```

When r1 = 0xFEDBA987

```
sat.uh  %r2,%r1     ; 0x0000FFFF → r2
```

sat.uw *%rd, %rs*

Function

Unsigned saturation (32 bits)

Standard) *rd* ← *rs* if C = 0;
rd ← 0xFFFFFFFF if C = 1

Extension 1) Unusable

Extension 2) Unusable

Extension 3) Unusable

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Perform unsigned 32-bit saturation processing.

The content of the C flag is tested, and the result is stored in the *rd* register.

Flag condition	Processing	S flag
C = 0	*rs* → *rd*	–
C = 1	0xFFFFFFFF → *rd*	1

(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 = 0x555 and C = 0

```
sat.uw  %r2,%r1     ; 0x00000555 → r2
```

When r1 = 0xFEDBA987 and C = 1

```
sat.uw  %r2,%r1     ; 0xFFFFFFFF → r2
```

sat.w *%rd, %rs*

Function Signed saturation (32 bits)

Standard) *rd* ← *rs* if V = 0;
rd ← 0x80000000 if V = 1 & N = 0;
rd ← 0xFFFFFFFF if V = 1 & N = 1

Extension 1) Unusable
Extension 2) Unusable
Extension 3) Unusable

Code

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

0xBD__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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

Perform singed 32-bit saturation processing.
The contents of the flags are tested, and the result is stored in the *rd* register.

Flag condition	Processing	S flag
V = 0	*rs* → *rd*	–
V = 1 & N = 0	0x80000000 → *rd*	1
V = 1 & N = 1	0xFFFFFFFF → *rd*	1

(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 V = 1 and N = 1

```
sat.w   %r2,%r1       ; 0x7FFFFFFF → r2
```

When V = 1 and N = 0

```
sat.w   %r2,%r1       ; 0x80000000 → r2
```

sbc %rd, %rs

Function

Subtraction with borrow

Standard) *rd* ← *rd* - *rs* - C

Extension 1) Unusable

Extension 2) Unusable

Extension 3) *rd* ← *rs1* - *rs2* - C (“*op*, *imm2*” is usable)

Code

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

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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) Extension 3

```
ext   %rs2,op,imm2     ; op = sra, srl, sla, imm2 = 0-3
sbc   %rd,%rs1         ; rd ← (rs1 - rs2 - C) op imm2
```

The register *rs2* specified by the `ext` instruction and C (carry) flag are subtracted from the content of the *rs1* register, and the content of the *rs1* register is then shifted as indicated by *op* a number of bits equal to *imm2*, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(3) Delayed instruction

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

(4) Postshift

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `sbc` instruction.

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
```

(3)
```
ext   %r2,srl,1
sbc   %r3,%r1          ; r3 = (r1 - r2 - C) >> 1
```

scan0 *%rd, %rs*

Function

Scan bits for 0

Standard)	*rd* ← bit offset of 0 in *rs*(31:0)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15			12	11			8	7	4	3	0	
1	0	0	0	1	0	1	0	*rs*		*rd*		0x8A__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	0	↔	0

Mode

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

CLK

One cycle

Description

(1) Standard: When the SW flag (bit 22) in the PSR = 0

When the `scan0` instruction is executed after the SW flag (bit 22) in the PSR is reset to 0, the same processing as in the C33 STD Core CPU is performed.

The most significant byte in the *rs* register (bits 31–24) is scanned for a logic 0 and, when a bit that contains a 0 is found, the position of that bit (offset from the MSB) is loaded into the *rd* register. If MSB = 0, data “0” is loaded into the *rd* register and the Z flag is set. If 0s are not found in any bits of the most significant byte of the *rs* register, 0x00000008 is loaded into the *rd* register and the C flag is set.

8 high-order bits in *rs* (bin)	*rd* register (hex)	Flag			
		C	V	Z	N
`0xxx xxxx`	0x00000000	0	0	1	0
`10xx xxxx`	0x00000001	0	0	0	0
`110x xxxx`	0x00000002	0	0	0	0
`1110 xxxx`	0x00000003	0	0	0	0
`1111 0xxx`	0x00000004	0	0	0	0
`1111 10xx`	0x00000005	0	0	0	0
`1111 110x`	0x00000006	0	0	0	0
`1111 1110`	0x00000007	0	0	0	0
`1111 1111`	0x00000008	1	0	0	0

(2) Advanced mode: When the SW flag (bit 22) in the PSR = 1

When the `scan0` instruction is executed after the SW flag (bit 22) in the PSR is set to 1, it functions as a 32-bit `scan0` instruction.

All bits in the *rs* register (bits 31–0) are scanned for a logic 0 and, when a bit that contains a 0 is found, the position of that bit (offset from the MSB) is loaded into the *rd* register. If MSB = 0, data “0” is loaded into the *rd* register and the Z flag is set. If 0s are not found in any bits of the *rs* register, 0x00000020 is loaded into the *rd* register and the C flag is set.

rs register (bin)	*rd* register (hex)	Flag			
		C	V	Z	N
`0xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000000	0	0	1	0
`10xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000001	0	0	0	0
`110x xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000002	0	0	0	0
`1110 xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000003	0	0	0	0
`1111 0xxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000004	0	0	0	0
`1111 10xx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000005	0	0	0	0
`1111 110x xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000006	0	0	0	0
`1111 1110 xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000007	0	0	0	0
`1111 1111 0xxx xxxx xxxx xxxx xxxx xxxx`	0x00000008	0	0	0	0
`1111 1111 10xx xxxx xxxx xxxx xxxx xxxx`	0x00000009	0	0	0	0
:	:	:	:	:	:
`1111 1111 1111 1111 1111 1111 1111 110x`	0x0000001e	0	0	0	0
`1111 1111 1111 1111 1111 1111 1111 1110`	0x0000001f	0	0	0	0
`1111 1111 1111 1111 1111 1111 1111 1111`	0x00000020	1	0	0	0

(3) Delayed instruction

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

Example

```
psrset  22              ; psr(22) ← 1
scan0   %r1,%r0         ; Bits in r0(31:0) scanned for 0
```

scan1 *%rd, %rs*

Function Scan bits for 1

Standard)	*rd* ← bit offset of 1 in *rs*(31:0)
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	Unusable

Code

15–12	11–8	7–4	3–0	
1 0 0 0	1 1 1 0	*rs*	*rd*	0x8E__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	0	↔	0

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

CLK One cycle

Description (1) Standard: When the SW flag (bit 22) in the PSR = 0

When the `scan1` instruction is executed after the SW flag (bit 22) in the PSR is reset to 0, the same processing as in the S1C33 series is performed.

The most significant byte in the *rs* register (bits 31–24) is scanned for a logic 1 and, when a bit that contains a 1 is found, the position of that bit (offset from the MSB) is loaded into the *rd* register. If MSB = 1, data “0” is loaded into the *rd* register and the Z flag is set. If 1s are not found in any bits of the most significant byte of the *rs* register, 0x00000008 is loaded into the *rd* register and the C flag is set.

8 high-order bits in *rs* (bin)	*rd* register (hex)	Flag			
		C	V	Z	N
`1xxx xxxx`	0x00000000	0	0	1	0
`01xx xxxx`	0x00000001	0	0	0	0
`001x xxxx`	0x00000002	0	0	0	0
`0001 xxxx`	0x00000003	0	0	0	0
`0000 1xxx`	0x00000004	0	0	0	0
`0000 01xx`	0x00000005	0	0	0	0
`0000 001x`	0x00000006	0	0	0	0
`0000 0001`	0x00000007	0	0	0	0
`0000 0000`	0x00000008	1	0	0	0

(2) Advanced mode: When the SW flag (bit 22) in the PSR = 1

When the `scan1` instruction is executed after the SW flag (bit 22) in the PSR is set to 1, it functions as a 32-bit `scan1` instruction.

All bits in the *rs* register (bits 31–0) are scanned for a logic 1 and, when a bit that contains a 1 is found, the position of that bit (offset from the MSB) is loaded into the *rd* register. If MSB = 1, data “0” is loaded into the *rd* register and the Z flag is set. If 1s are not found in any bits of the *rs* register, 0x00000020 is loaded into the *rd* register and the C flag is set.

rs register (bin)	*rd* register (hex)	Flag			
		C	V	Z	N
`1xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000000	0	0	1	0
`01xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000001	0	0	0	0
`001x xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000002	0	0	0	0
`0001 xxxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000003	0	0	0	0
`0000 1xxx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000004	0	0	0	0
`0000 01xx xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000005	0	0	0	0
`0000 001x xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000006	0	0	0	0
`0000 0001 xxxx xxxx xxxx xxxx xxxx xxxx`	0x00000007	0	0	0	0
`0000 0000 1xxx xxxx xxxx xxxx xxxx xxxx`	0x00000008	0	0	0	0
`0000 0000 01xx xxxx xxxx xxxx xxxx xxxx`	0x00000009	0	0	0	0
:	:	:	:	:	:
`0000 0000 0000 0000 0000 0000 0000 001x`	0x0000001e	0	0	0	0
`0000 0000 0000 0000 0000 0000 0000 0001`	0x0000001f	0	0	0	0
`0000 0000 0000 0000 0000 0000 0000 0000`	0x00000020	1	0	0	0

(3) Delayed instruction

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

Example

```
psrset  22              ; psr(22) ← 1
scan1   %r1,%r0         ; Bits in r0(31:0) scanned for 1
```

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
Extension 3)	*rd* ← *rs1* << *rs2*

Code

15			12	11			8	7 4	3 0	
1	0	0	1	0	1	0	1	*rs*	*rd*	0x95__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

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

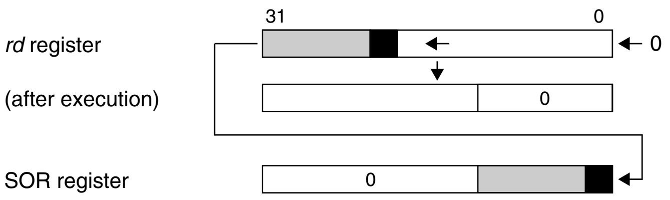
CLK

One cycle

Description

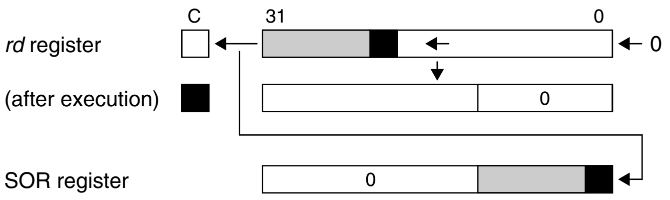
(1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode specified in (1), except that the bit shifted out from the most significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs1
sla   %rd,%rs2
```

The *rs1* register is shifted to the left as many bits as specified by *rs2*. Operation of this instruction is the same as that in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs1` instruction.
Data "0" is placed in the least significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 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
Extension 3)	*rd* ← *rs* << *imm5*

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

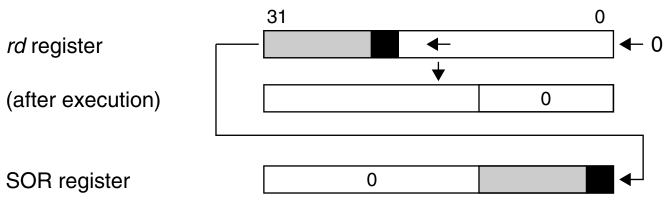
CLK

One cycle

Description

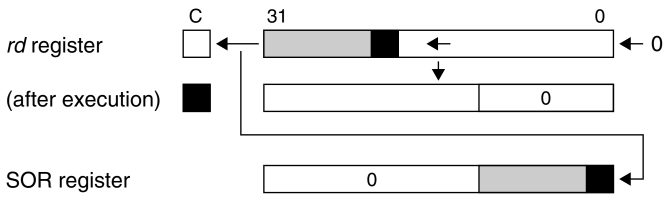
(1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the most significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs
sla   %rd,imm5
```

The *rs* register is shifted to the left as many bits as specified by *imm5*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs` instruction.
Data "0" is placed in the least significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 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
Extension 3)	*rd* ← *rs1* << *rs2*

Code

15	14	13	12	11	10	9	8	7–4	3–0	
1	0	0	0	1	1	0	1	*rs*	*rd*	0x8D__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

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

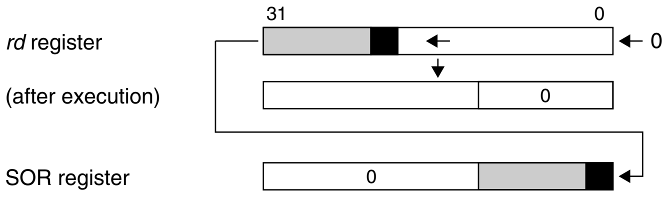
CLK

One cycle

Description

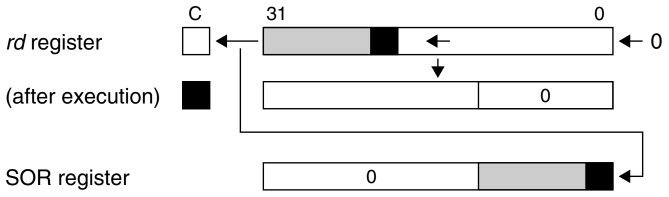
(1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the most significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs1
sll   %rd,%rs2
```

The *rs1* register is shifted to the left as many bits as specified by *rs2*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs1` instruction.
Data “0” is placed in the least significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 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
Extension 3)	*rd* ← *rs* << *imm5*

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

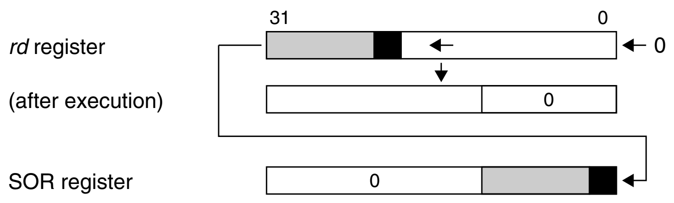
One cycle

Description

(1) Standard: When the SE flag (bit 20) in the PSR = 0

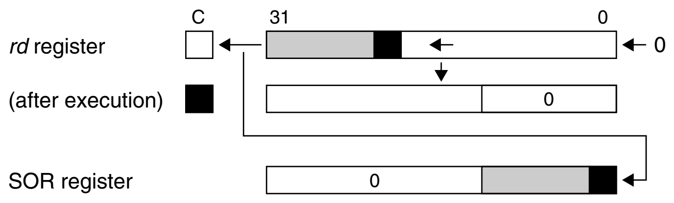
The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. Data “0” is placed in the least significant bit of the *rd* register.

The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the most significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs
sll   %rd,imm5
```

The *rs* register is shifted to the left as many bits as specified by *imm5*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs` instruction.
Data "0" is placed in the least significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 included.

slp

Function

SLEEP

Standard)	Place the CPU in SLEEP mode
Extension 1)	Unusable
Extension 2)	Unusable
Extension 3)	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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

–

CLK

One cycle

Description

Places the CPU in SLEEP mode.
As a result, the CPU and the peripheral circuits in the chip stop operating, thereby significantly reducing the current consumption in the chip.
The CPU is reawaken by an interrupt and, after executing the handler routine for the interrupt, the CPU returns to the instruction next to `slp`.
To keep the CPU in sleep state until an external wakeup cause such as an NMI is received following the execution of `slp`, a CMU control bit (bit 0 at the address 0x48368) must be set to 0. (If the CMU register is to be altered, register protection must be removed by writing data "0x96" to bits 0–7 at the address 0x4836E.)
By setting this control bit to 1, the CPU can automatically wake up from SLEEP mode a certain length of time after executing `slp`. For details, refer to the CMU section in the Technical Manual for each model.

Example

```
slp       ; The CPU 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
Extension 3)	*rd* ← *rs1* >> *rs2*

Code

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

0x91__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

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

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

CLK

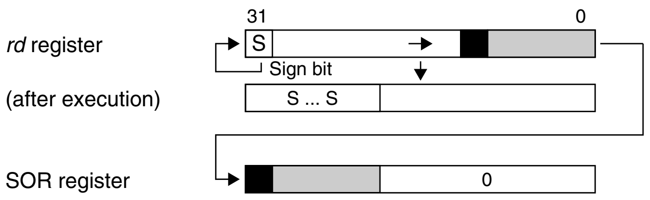
One cycle

Description

(1) Standard: When the SE flag (bit 20) in the PSR = 0

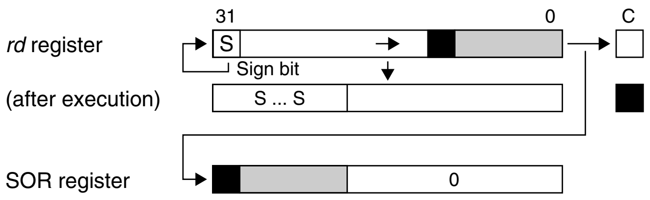
The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5 low-order bits of the *rs* register. The sign bit is copied to the most significant bit of the *rd* register.

The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the least significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0

C ^ N → V = 1

(3) Extension 3

```
ext  %rs1
sra  %rd,%rs2
```

The *rs1* register is shifted to the right as many bits as specified by *rs2*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs1` instruction.
The sign bit is copied to the most significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 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
Extension 3)	*rd* ← *rs* >> *imm5*

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

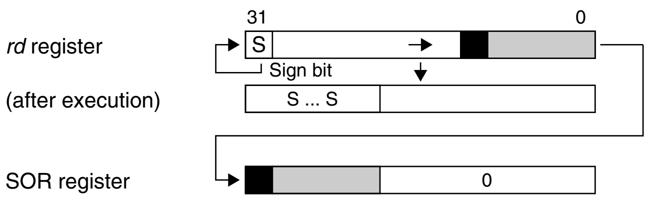
One cycle

Description

(1) Standard: When the SE flag (bit 20) in the PSR = 0

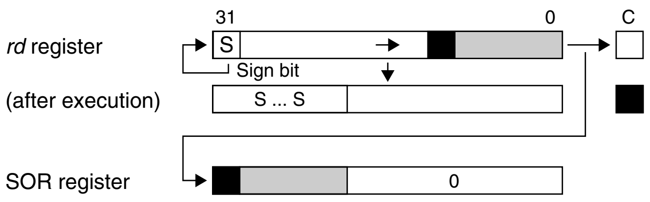
The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. The sign bit is copied to the most significant bit of the *rd* register.

The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the least significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs
sra   %rd,imm5
```

The *rs* register is shifted to the right as many bits as specified by *imm5*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs` instruction.
The sign bit is copied to the most significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 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
Extension 3)	*rd* ← *rs1* >> *rs2*

Code

15			12	11			8	7–4	3–0	
1	0	0	0	1	0	0	1	*rs*	*rd*	0x89__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

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

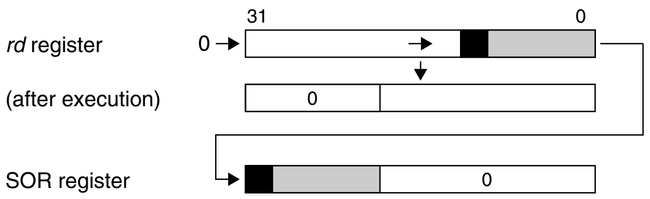
CLK

One cycle

Description

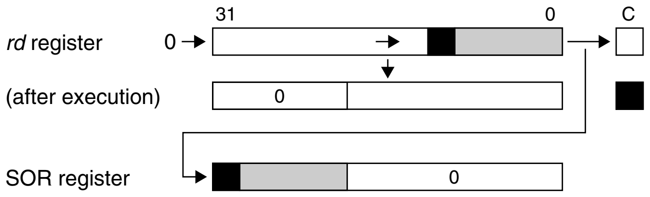
(1) Standard: When the SE flag (bit 20) in the PSR = 0

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. The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the least significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext  %rs1
srl  %rd,%rs2
```

The *rs1* register is shifted to the right as many bits as specified by *rs2*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs1` instruction.
Data “0” is placed in the most significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 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
Extension 3)	*rd* ← *rs* >> *imm5*

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

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

The C and V flags change when the SE flag (bit 20) in the PSR = 1.

Mode

Src: Immediate (unsigned)
Dst: Register direct `%rd` = `%r0` to `%r15`

CLK

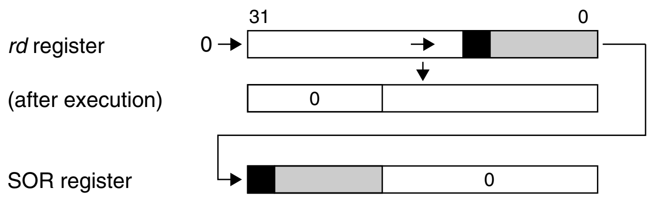
One cycle

Description

(1) Standard: When the SE flag (bit 20) in the PSR = 0

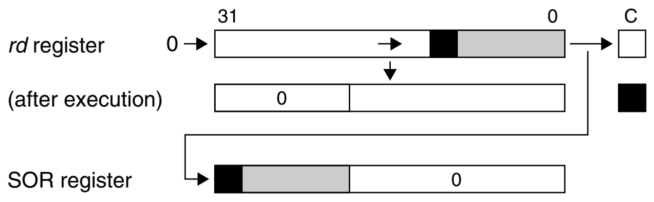
The *rd* register is shifted as shown in the diagram below. The number of bits to be shifted can be specified in the range of 0 to 31 by the 5-bit immediate *imm5*. Data "0" is placed in the most significant bit of the *rd* register.

The bit that has been shifted out can be read from the SOR register.



(2) Advanced mode: When the SE flag (bit 20) in the PSR = 1

Operation is the same as in the standard mode described in (1), except that the bit shifted out from the least significant bit position is placed in the C flag of the PSR.



The V flag changes state depending on the status of the C and Z flags at completion of the shift operation.

C & N | !C & !N → V = 0
C ^ N → V = 1

(3) Extension 3

```
ext   %rs
srl   %rd,imm5
```

The *rs* register is shifted to the right as many bits as specified by *imm5*. Operation of this instruction is the same as in the standard or advanced modes, except that the source register to be shifted is given by the `ext %rs` instruction.
Data “0” is placed in the most significant bit of the *rd* register.
The bit that has been shifted out can be read from the SOR 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 included.

sub %rd, %rs

Function

Subtraction
Standard) *rd* ← *rd* - *rs*
Extension 1) *rd* ← *rs* - *imm13*
Extension 2) *rd* ← *rs* - *imm26*
Extension 3) *rd* ← *rs1* - *rs2* ("*op*, *imm2*" is usable)

Code

15	14	13	12	11	10	9	8	7–4	3–0	
0	0	1	0	0	1	1	0	*rs*	*rd*	0x26__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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) Extension 3

```
ext   %rs2,op,imm2     ; op = sra, srl, sla, imm2 = 0-3
sub   %rd,%rs1         ; rd ← (rs1 - rs2) op imm2
```

The register *rs2* specified by the `ext` instruction is subtracted from the content of the *rs1* register, and the content of the *rs1* register is then shifted as indicated by *op* a number of bits equal to *imm2*, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(5) 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.

(6) Postshift

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `sub` instruction.

Example

```
(1) sub   %r0,%r0          ; r0 = r0 - r0

(2) ext   0x1
    ext   0x1fff
    sub   %r1,%r2          ; r1 = r2 - 0x3fff

(3) ext   %r2,srl,1
    sub   %r3,%r1          ; r3 = (r1 - r2) >> 1
```

sub %rd, imm6

Function

Subtraction

Standard)	*rd* ← *rd* - *imm6*
Extension 1)	*rd* ← *rd* - *imm19*
Extension 2)	*rd* ← *rd* - *imm32*
Extension 3)	Unusable

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	0	0	1	*imm6*	*r d*	0x64__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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.

(5) Postshift ("`ext` *op*, *imm2*" only)

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `sub` instruction.

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
Extension 3)	Unusable

Code

15			12	11	10	9 … 0	
1	0	0	0	0	1	*imm10*	0x84__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	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.

(3) Postshift (“`ext` *`op`*`,`*`imm2`*” only)

The execution result of this instruction may be shifted a maximum of 3 bits by writing it directly after an extension instruction with postshift. In this case, the result is shifted the same way as the `sra`, `srl`, or `sll` instruction. However, because the shift-out register SOR is unused, the SOR does not change. Furthermore, the C, V, Z, and N flags are irrelevant to the shift operation, and are determined only by the result of the `sub` instruction.

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
Extension 3)	Unusable

Code

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

0x92__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

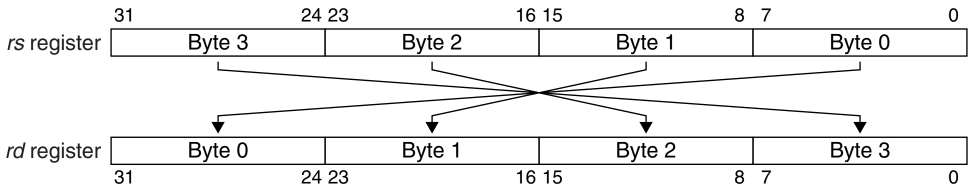
CLK

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
Extension 3)	Unusable

Code

15			12	11			8	7–4	3–0	
1	0	0	1	1	0	1	0	*rs*	*rd*	0x9A__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

Mode

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

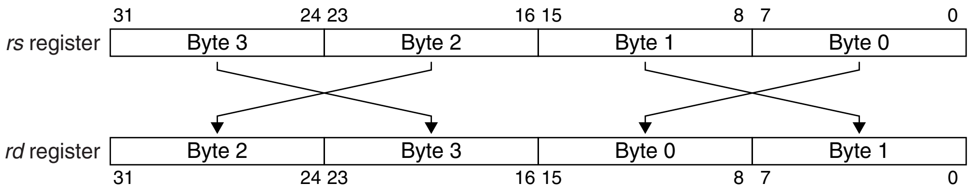
CLK

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*
Extension 3)	*rd* ← *rs1* ^ *rs2*

Code

15			12	11			8	7–4	3–0	
0	0	1	1	1	0	1	0	*rs*	*rd*	0x3A__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	∗1	↔	↔

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) Extension 3

```
ext    %rs2
xor    %rd,%rs1         ; rd ← rs1 ^ rs2
```

The content of the *rs1* register and the register *rs2* specified by the `ext` instruction are exclusively OR'ed, and the result is loaded into the *rd* register. The contents of the *rs1* and *rs2* registers are not altered.

(5) 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.

∗1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `and`, `not`, and `or` instructions, refer to the description of each instruction.)

Example

```
(1) xor   %r0,%r0         ; r0 = r0 ^ r0

(2) ext   0x1
    ext   0x1fff
    xor   %r1,%r2         ; r1 = r2 ^ 0x00003fff

(3) ext   %r5
    xor   %r3,%r4         ; r3 = r4 ^ r5
```

xor %rd, sign6

Function

Exclusive OR

Standard) *rd* ← *rd* ^ *sign6*
Extension 1) *rd* ← *rd* ^ *sign19*
Extension 2) *rd* ← *rd* ^ *sign32*
Extension 3) Unusable

Code

15	14	13	12	11	10	9–4	3–0	
0	1	1	1	1	0	*sign6*	*r d*	0x78__

Flag

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

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.

*1 The V flag is cleared to 0 by executing this instruction after setting the OC flag in the PSR to 1. The same applies to other logical operation instructions. (For the functions of the `and`, `not`, and `or` instructions, refer to the description of each instruction.)

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	Loop	Repeat
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`	1	×	×	○	×
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	`halt`	1	×	×	○	×
0	0	0	0	0	0	1	0	0	0	0	0	*rs*				`pushn   %rs`	N	×	×	○	×
0	0	0	0	0	0	1	0	0	1	0	0	*rd*				`popn    %rd`	N	×	×	○	×
0	0	0	0	0	0	1	0	1	1	0	0	*rb*				**`jpr     %rb`**	4	×	×	*1	×
0	0	0	0	0	0	1	1	1	1	0	0	*rb*				**`jpr.d   %rb`**	3	×	×	*2	×
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	`brk`	7	×	×	*1	×
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	`retd`	6	×	×	×	×
0	0	0	0	0	1	0	0	1	0	0	0	0	0	*imm2*		`int     imm2`	7	×	×	*1	×
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	`reti`	6	×	×	×	×
0	0	0	0	0	1	1	0	0	0	0	0	*rb*				`call    %rb`	3	×	×	*1	×
0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	`ret`	5	×	×	×	×
0	0	0	0	0	1	1	0	1	0	0	0	*rb*				`jp      %rb`	3	×	×	*1	×
0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	**`retm`**	6	×	×	×	×
0	0	0	0	0	1	1	1	0	0	0	0	*rb*				`call.d  %rb`	2	×	×	*2	×
0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	`ret.d`	4	×	×	×	×
0	0	0	0	0	1	1	1	1	0	0	0	*rb*				`jp.d    %rb`	2	×	×	*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	Loop	Repeat
Class			op1				op2			0	1	*imm4,r,s*									
0	0	0	0	0	0	0	0	0	0	0	1	*rs*				**`push    %rs`**	1	×	×	○	○
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`**	N	×	×	○	×
0	0	0	0	0	0	0	0	1	1	0	1	*sd*				**`pops    %sd`**	N	×	×	*3	×
0	0	0	0	0	0	0	1	0	0	0	1	*rs*				**`mac.w   %rs`**	3 + N × 2	×	×	○	×
0	0	0	0	0	0	0	1	0	1	0	1	*rs*				**`mac.hw  %rs`**	2 + N × 2	×	×	○	×
0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	**`macclr`**	1	×	○	○	○
0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	**`ld.cf`**	1	×	○	○	○
0	0	0	0	0	0	1	0	0	0	0	1	*rs*				**`divu.w  %rs`**	35	×	×	○	×
0	0	0	0	0	0	1	0	0	1	0	1	*rs*				**`div.w   %rs`**	35	×	×	○	×
0	0	0	0	0	0	1	0	1	0	0	1	*rc*				**`repeat  %rc`**	4	×	×	×	×
0	0	0	0	0	0	1	0	1	1	0	1	*imm4*				**`repeat  imm4`**	4	×	×	×	×
0	0	0	0	0	0	1	1	0	1	0	1	*rd*				**`add     %rd,%dp`**	1	○	○	○	○

Class 0 (3)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class			op1				d	*sign8*													
0	0	0	0	1	0	0	0	*sign8*								`jrgt    sign8`	1–2	○	×	*1	×
0	0	0	0	1	0	0	1	*sign8*								`jrgt.d  sign8`	1	○	×	*2	×
0	0	0	0	1	0	1	0	*sign8*								`jrge    sign8`	1–2	○	×	*1	×
0	0	0	0	1	0	1	1	*sign8*								`jrge.d  sign8`	1	○	×	*2	×
0	0	0	0	1	1	0	0	*sign8*								`jrlt    sign8`	1–2	○	×	*1	×
0	0	0	0	1	1	0	1	*sign8*								`jrlt.d  sign8`	1	○	×	*2	×
0	0	0	0	1	1	1	0	*sign8*								`jrle    sign8`	1–2	○	×	*1	×
0	0	0	0	1	1	1	1	*sign8*								`jrle.d  sign8`	1	○	×	*2	×
0	0	0	1	0	0	0	0	*sign8*								`jrugt   sign8`	1–2	○	×	*1	×
0	0	0	1	0	0	0	1	*sign8*								`jrugt.d sign8`	1	○	×	*2	×
0	0	0	1	0	0	1	0	*sign8*								`jruge   sign8`	1–2	○	×	*1	×
0	0	0	1	0	0	1	1	*sign8*								`jruge.d sign8`	1	○	×	*2	×
0	0	0	1	0	1	0	0	*sign8*								`jrult   sign8`	1–2	○	×	*1	×
0	0	0	1	0	1	0	1	*sign8*								`jrult.d sign8`	1	○	×	*2	×
0	0	0	1	0	1	1	0	*sign8*								`jrule   sign8`	1–2	○	×	*1	×
0	0	0	1	0	1	1	1	*sign8*								`jrule.d sign8`	1	○	×	*2	×
0	0	0	1	1	0	0	0	*sign8*								`jreq    sign8`	1–2	○	×	*1	×
0	0	0	1	1	0	0	1	*sign8*								`jreq.d  sign8`	1	○	×	*2	×
0	0	0	1	1	0	1	0	*sign8*								`jrne    sign8`	1–2	○	×	*1	×
0	0	0	1	1	0	1	1	*sign8*								`jrne.d  sign8`	1	○	×	*2	×
0	0	0	1	1	1	0	0	*sign8*								`call    sign8`	2	○	×	*1	×
0	0	0	1	1	1	0	1	*sign8*								`call.d  sign8`	1	○	×	*2	×
0	0	0	1	1	1	1	0	*sign8*								`jp      sign8`	2	○	×	*1	×
0	0	0	1	1	1	1	1	*sign8*								`jp.d    sign8`	1	○	×	*2	×

Class 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class			op1			op2		*imm5,rb,rs*				*imm2,rs,rd*									
0	0	1	0	0	0	0	0	*rb*				*rd*				ld.b %*rd*,[%*rb*]	1	○	×	○	○
0	0	1	0	0	0	0	1	*rb*				*rd*				ld.b %*rd*,[%*rb*]+	1	○	×	○	○
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	○	×	○	○
0	0	1	0	0	1	0	1	*rb*				*rd*				ld.ub %*rd*,[%*rb*]+	1	○	×	○	○
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	○	×	○	○
0	0	1	0	1	0	0	1	*rb*				*rd*				ld.h %*rd*,[%*rb*]+	1	○	×	○	○
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	○	×	○	○
0	0	1	0	1	1	0	1	*rb*				*rd*				ld.uh %*rd*,[%*rb*]+	1	○	×	○	○
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	○	×	○	○
0	0	1	1	0	0	0	1	*rb*				*rd*				ld.w %*rd*,[%*rb*]+	1	○	×	○	○
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	○	×	○	○
0	0	1	1	0	1	0	1	*rb*				*rs*				ld.b [%*rb*]+,%*rs*	1	○	×	○	○
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	○	×	○	○
0	0	1	1	1	0	0	1	*rb*				*rs*				ld.h [%*rb*]+,%*rs*	1	○	×	○	○
0	0	1	1	1	0	1	0	*rs*				*rd*				xor %*rd*,%*rs*	1	○	○	○	○
0	0	1	1	1	0	1	1	0	0	0	0	0	1	*imm2*		**ext sra,*imm2***	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	0	0	0	0	1	0	*imm2*		**ext srl,*imm2***	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	0	0	0	0	1	1	*imm2*		**ext sll,*imm2***	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	**ext gt**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	**ext ge**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	0	1	1	0	0	0	0	0	**ext lt**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	0	**ext le**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	**ext ugt**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0	**ext uge**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	1	0	1	0	0	0	0	0	**ext ult**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	1	0	1	1	0	0	0	0	**ext ule**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	1	1	0	0	0	0	0	0	**ext eq**	0–1	×	×	*1	×
0	0	1	1	1	0	1	1	1	1	0	1	0	0	0	0	**ext ne**	0–1	×	×	*1	×
0	0	1	1	1	1	0	0	*rb*				*rs*				ld.w [%*rb*],%*rs*	1	○	×	○	○
0	0	1	1	1	1	0	1	*rb*				*rs*				ld.w [%*rb*]+,%*rs*	1	○	×	○	○
0	0	1	1	1	1	1	0	*rs*				*rd*				not %*rd*,%*rs*	1	×	○	○	○
0	0	1	1	1	1	1	1	*rs*				0	0	0	0	**ext %*rs***	0–1	×	×	*1	×
0	0	1	1	1	1	1	1	*rs*				0	1	*imm2*		**ext %*rs*,sra,*imm2***	0–1	×	×	*1	×
0	0	1	1	1	1	1	1	*rs*				1	0	*imm2*		**ext %*rs*,srl,*imm2***	0–1	×	×	*1	×
0	0	1	1	1	1	1	1	*rs*				1	1	*imm2*		**ext %*rs*,sll,*imm2***	0–1	×	×	*1	×

Class 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class			op1			*imm6*						*rs,rd*									
0	1	0	0	0	0	*imm6*						*rd*				ld.b %*rd*,[%sp+*imm6*]	1	○	×	○	○
0	1	0	0	0	1	*imm6*						*rd*				ld.ub %*rd*,[%sp+*imm6*]	1	○	×	○	○
0	1	0	0	1	0	*imm6*						*rd*				ld.h %*rd*,[%sp+*imm6*]	1	○	×	○	○
0	1	0	0	1	1	*imm6*						*rd*				ld.uh %*rd*,[%sp+*imm6*]	1	○	×	○	○
0	1	0	1	0	0	*imm6*						*rd*				ld.w %*rd*,[%sp+*imm6*]	1	○	×	○	○
0	1	0	1	0	1	*imm6*						*rs*				ld.b [%sp+*imm6*],%*rs*	1	○	×	○	○
0	1	0	1	1	0	*imm6*						*rs*				ld.h [%sp+*imm6*],%*rs*	1	○	×	○	○
0	1	0	1	1	1	*imm6*						*rs*				ld.w [%sp+*imm6*],%*rs*	1	○	×	○	○

Class 3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
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)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class			op1			*imm10*																
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)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class			op1			op2		*imm5,rs*				0,*rd*										
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	0	1	0	*rs*				*rd*				**scan0 %rd,%rs**	1	×	○	○	○
1	0	0	0	1	0	1	1	*rs*				0	0	0	0	div0s %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	0	1	1	1	0	*rs*				*rd*				**scan1 %rd,%rs**	1	×	○	○	○
1	0	0	0	1	1	1	1	*rs*				0	0	0	0	div0u %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	0	1	1	*rs*				0	0	0	0	div1 %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	0	1	1	0	*rs*				*rd*				mirror %rd,%rs	1	×	○	○	○
1	0	0	1	0	1	1	1	*rs*				0	0	0	0	div2s %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	0	1	1	*rs*				0	0	0	0	div3s	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	△	○	○	○
1	0	0	1	1	1	1	0	*rs*				*rd*				**sat.b %rd,%rs**	1	×	○	○	○
1	0	0	1	1	1	1	1	*rs*				*rd*				**sat.ub %rd,%rs**	1	×	○	○	○

Class 5 (1)

15 14 13	12 11 10	9 8	7 6 5 4	3 2 1 0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class	op1	op2	*imm4,r,s*	*0,imm3,r,s*						
1 0 1	0 0 0	0 0	*rs*	*sd*	**ld.w %sd,%rs**	1	×	○	○	○
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	1	×	○	○	○
1 0 1	0 0 0	1 1	*rs*	*rd*	**mlt.hw %rd,%rs**	2	×	×	○	○
1 0 1	0 0 1	0 0	*ss*	*rd*	**ld.w %rd,%ss**	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	1	×	○	○	○
1 0 1	0 0 1	1 1	*rs*	*rd*	**mac1.h %rd,%rs**	1	×	×	○	○
1 0 1	0 1 0	0 0	*rb*	0 *imm3*	btst [%rb],*imm3*	3	○	×	○	○
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	2	×	×	○	○
1 0 1	0 1 0	1 1	*rs*	*rd*	**mac1.hw %rd,%rs**	2	×	×	○	○
1 0 1	0 1 1	0 0	*rb*	0 *imm3*	bclr [%rb],*imm3*	3	○	×	○	○
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	2	×	×	○	○
1 0 1	1 0 0	0 0	*rb*	0 *imm3*	bset [%rb],*imm3*	3	○	×	○	○
1 0 1	1 0 0	0 1	*imm4*	*rd*	**ld.c %rd,*imm4***	1	×	○	○	○
1 0 1	1 0 0	1 0	*rs*	0 0 0 0	mac %rs	2 + N × 2	×	×	○	×
1 0 1	1 0 0	1 1	*rs*	*rd*	**mac1.w %rd,%rs**	2	×	×	○	○
1 0 1	1 0 1	0 0	*rb*	0 *imm3*	bnot [%rb],*imm3*	3	○	×	○	○
1 0 1	1 0 1	0 1	*imm4*	*rs*	**ld.c *imm4*,%rs**	1	×	○	○	○
1 0 1	1 0 1	1 0	*rs*	*rd*	**sat.h %rd,%rs**	1	×	○	○	○
1 0 1	1 0 1	1 1	*rs*	*rd*	**sat.uh %rd,%rs**	1	×	○	○	○
1 0 1	1 1 0	0 0	*rs*	*rd*	**adc %rd,%rs**	1	△	○	○	○
1 0 1	1 1 0	0 1	*ra*	*rc*	**loop %rc,%ra**	5	×	×	×	×
1 0 1	1 1 0	1 0	*imm4*	*rc*	**loop %rc,*imm4***	5	×	×	×	×
1 0 1	1 1 0	1 1	*imm4(addr)*	*imm4(count)*	**loop *imm4*,*imm4***	5	×	×	×	×
1 0 1	1 1 1	0 0	*rs*	*rd*	**sbc %rd,%rs**	1	△	○	○	○
1 0 1	1 1 1	0 1	*rs*	*rd*	**sat.w %rd,%rs**	1	×	○	○	○
1 0 1	1 1 1	1 0	*rs*	*rd*	**sat.uw %rd,%rs**	1	×	○	○	○

Class 5 (2)

15 14 13	12 11 10	9 8	7 6	5 4 3 2 1 0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class	op1	op2	op3	*imm5,imm6*						
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***	4	×	○	○	○
1 0 1	1 1 1	1 1	1 0	0 *imm5*	**psrclr *imm5***	4	×	○	○	○

Class 6

15 14 13	12 11 10 9 8 7 6 5 4 3 2 1 0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class	*imm13*						
1 1 0	*imm13*	ext *imm13*	0–1	×	×	*1	×

Class 7

15 14 13	12 11 10	9 8 7 6 5 4	3 2 1 0	Mnemonic	Cycle	Extension	Delayed S	Loop	Repeat
Class	op1	*imm6*	*rs,rd*						
1 1 1	0 0 0	*imm6*	*rd*	**ld.b %rd,[%dp+*imm6*]**	1	○	×	○	○
1 1 1	0 0 1	*imm6*	*rd*	**ld.ub %rd,[%dp+*imm6*]**	1	○	×	○	○
1 1 1	0 1 0	*imm6*	*rd*	**ld.h %rd,[%dp+*imm6*]**	1	○	×	○	○
1 1 1	0 1 1	*imm6*	*rd*	**ld.uh %rd,[%dp+*imm6*]**	1	○	×	○	○
1 1 1	1 0 0	*imm6*	*rd*	**ld.w %rd,[%dp+*imm6*]**	1	○	×	○	○
1 1 1	1 0 1	*imm6*	*rs*	**ld.b [%dp+*imm6*],%rs**	1	○	×	○	○
1 1 1	1 1 0	*imm6*	*rs*	**ld.h [%dp+*imm6*],%rs**	1	○	×	○	○
1 1 1	1 1 1	*imm6*	*rs*	**ld.w [%dp+*imm6*],%rs**	1	○	×	○	○

Inst	Function-Extended Instructions
▨ **Inst**	Added Instructions
△	Only extension 3 (3-operand operation) is usable.

*1 The instruction cannot be placed at the memory location that is the loop end address (LEA).
*2 The instruction cannot be placed at the memory location that is the loop end address (LEA) or is equal to LEA - 2.
*3 The registers used in the `loop` instruction (LSA, LEA, and LCO) cannot be popped off the stack.

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 OFFICES -

West

1960 E.Grand Avenue Flr 2
El Segundo, CA 90245, U.S.A.
Phone: +1-800-249-7730 Fax: +1-310-955-5400

Central

101 Virginia Street, Suite 290
Crystal Lake, IL 60014, U.S.A.
Phone: +1-800-853-3588 Fax: +1-815-455-7633

Northeast

301 Edgewater Place, Suite 210
Wakefield, MA 01880, U.S.A.
Phone: +1-800-922-7667 Fax: +1-781-246-5443

Southeast

3010 Royal Blvd. South, Suite 170
Alpharetta, GA 30005, U.S.A.
Phone: +1-877-332-0020 Fax: +1-770-777-2637

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

BARCELONA BRANCH OFFICE

Barcelona Design Center

Edificio Testa, C/Alcalde Barnils 64-68, Modulo C 2a planta
E-08190 Sant Cugat del Vallès, SPAIN
Phone: +34-93-544-2490 Fax: +34-93-544-2491

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 TAIWAN TECHNOLOGY & TRADING LTD.

14F, No. 7, Song Ren Road
Taipei 110
Phone: +886-2-8786-6688 Fax: +886-2-8786-6677

HSINCHU OFFICE

No. 99, Jiangong Road
Hsinchu City 300
Phone: +886-3-573-9900 Fax: +886-3-573-9169

EPSON SINGAPORE PTE., LTD.

401 Commonwealth Drive, #07-01
Haw Par Technocentre, SINGAPORE 149598
Phone: +65-6586-3100 Fax: +65-6472-4291

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

6F, Good Morning Securities Bldg., 56 Songjeong-Dong
Gumi-City, Seoul, 730-090, 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 ADV
Core CPU Manual

SEIKO EPSON CORPORATION
SEMICONDUCTOR OPERATIONS DIVISION

■ EPSON Electronic Devices Website

http://www.epsondevice.com

Document code: 410117900
Issue February, 2005
Printed in Japan ⓁB