User's Manual

V830 Family™

32-Bit Microprocessor

Architecture

Document No. U12496EJ4V0UMJ1 (4th edition)
Date Published August 2000 J CP(K)

[MEMO]

SUMMARY OF CONTENTS

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.l.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Madrid Office
Madrid, Spain
Tel: 91-504-2787
Fax: 91-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore
Tel: 65-253-8311
Fax: 65-250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP Brasil
Tel: 55-11-6462-6810
Fax: 55-11-6462-6829

J00.7

Major Revision in This Edition

Page	Description
p. 27	**Section 2.2.6** has been modified.
p. 111	**Table 5-4** has been modified.
p. 115	A note has been added to **Table 6-1**.
p. 120	**Section 6.4.1** has been added.
p. 127	**Section 7.2.2** has been modified.

The mark * shows major revised points.

[MEMO]

PREFACE

Intended readers

This manual is intended for those users who wish to become familiar with the functions of the V830 Family, and those involved in the design of systems based on the V830 Family.

- The V830 Family products
  - V830^{TM}: μPD705100
  - V831^{TM}: μPD705101
  - V832^{TM}: μPD705102

Purpose

The purpose of this manual is to assist users in understanding the architecture of the V830 Family, i.e., the topics listed in “Configuration” below.

Configuration

This manual covers the following:

- Register set
- Data set
- Address space
- Instructions
- Interrupts and exceptions
- Internal memory
- Reset
- Pipeline

How to use this manual

Readers of this manual are assumed to have a general knowledge of electronics, logic circuits, and microcomputers.

For an explanation of the hardware functions
→ Read the **User's Manual - Hardware** of each device.

For an explanation of the instructions
→ Read **Chapter 5**.

For an explanation of the electrical characteristics
→ Read the **Data Sheet** of each device.

To gain an overall understanding of the functions provided by the V830 Family
→ Read this manual in its entirety.

Legend

Significance of a data representation	:	Left high, right low
Representation of active low	:	$\overline{\text{XXX}}$ (bar above a pin or signal name)
Memory map address	:	Top upper, bottom lower
Note	:	Explanation of Note that appears in text

Caution	:	Point to which the user must pay particular attention
Remark	:	Supplementary explanation of the contents of the text
Numeric representations	:	XXXX or XXXXB for a binary number XXXX for a decimal number XXXXH for a hexadecimal number

Prefixes indicating powers of two (address space, memory capacity):

K (kilo)	:	$2^{10} = 1024$
M (mega)	:	$2^{20} = 1024^2$
G (giga)	:	$2^{30} = 1024^3$

Related documents Some related documents may be preliminary editions; if so, however, this is not indicated in this manual.

- Documents for the V830 Family

Product name		Data sheet	User's manual	
Alias name	Product		Hardware	Architecture
V830	μPD705100	U11483E	U10064E	This manual
V831	μPD705101	U12979E	U12273E	
V832	μPD705102	U13675E	U13577E	

- Documents for V830 Family development tools (User's manual)

Document name			Document No.
CA830 (C Compiler)		Operation (UNIX™-based)	U11013E
		Operation (Windows™-based)	U11068E
		Assembly Language	U11014E
		C	U11010E
		Project Manager	U11991E
RX830 (Real-Time OS)	ITRON1	Fundamental	U11730E
		Installation	U11731E
	μITRON Ver 3.0	Fundamental	U13152E
		Installation	U13151E

CONTENTS

LIST OF FIGURES

LIST OF TABLES

[MEMO]

CHAPTER 1 INTRODUCTION

The V830 Family, offered by NEC for built-in control applications, consists of RISC microprocessors having the V830 as their CPU core.

1.1 OVERVIEW

The V830 Family consists of high-performance 32-bit RISC microprocessors. The V830 Family can perform the data processing demanded by multimedia devices in only a few cycles. Besides a high interrupt responsibility and an optimized pipeline structure, a sum-of-products instruction, double-word shift instruction, and high-speed branch instruction using branch predication have been added to support multimedia functions.

Furthermore, by inheriting the V810 Family™ basic instruction set at the object level, V810 Family software can be used as is.

The V830 Family offers high performance for applications which require high-speed data processing, such as image processing, game machines, car navigation, high-performance TVs, color facsimile machines, Internet and intranet devices, office automation equipment, etc.

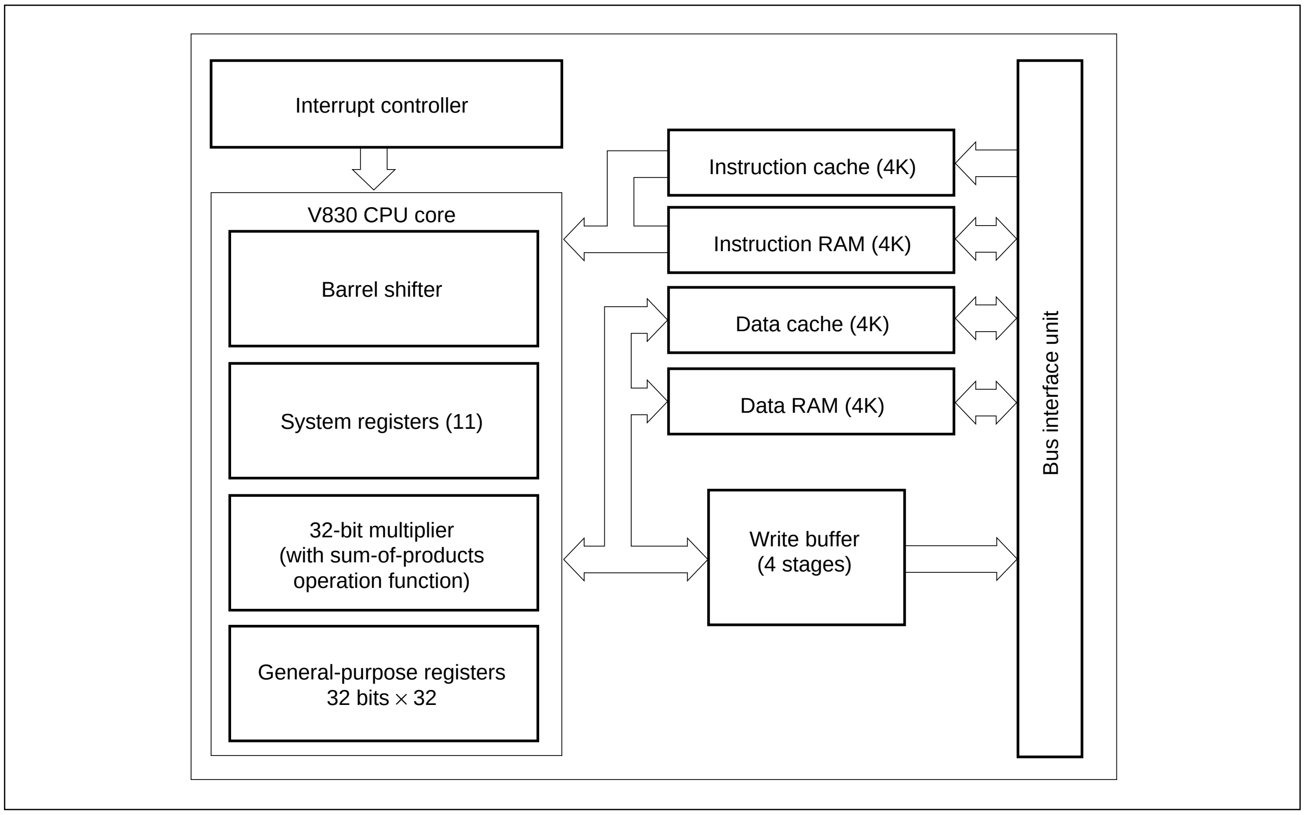
1.2 FEATURES

- Number of instructions: 102
- Minimum number of instruction execution cycles: 1
- General-purpose registers: 32 bits x 32
- Instruction set: V810 basic instruction set
  Sum-of-products operation (32 bits x 32 bits + (upper/lower) 32 bits): 1-3 cycles
  Saturatable arithmetic operation (with a saturation detection function)
  Double-word shift (64-bit data shift): 1-2 cycles
  High-speed branch
  Block transfer instruction
- Memory space
  Memory space, I/O space: 4G-byte linear address
- Internal memory
  Instruction cache (direct mapping): 4K bytes
  Data cache (direct mapping/write-through): 4K bytes
  Instruction RAM: 4K bytes
  Data RAM: 4K bytes
- Power control
  - Stop mode
  - Sleep mode
- CMOS structure

1.3 INTERNAL CONFIGURATION OF THE CPU

Figure 1-1 shows the internal configuration of a V830 Family microprocessor.

Figure 1-1. Internal Configuration



(1) CPU core

Executes the processing of the majority of instructions, including address calculation, arithmetic and logic operations, and data transfer within one cycle, by means of 5-stage pipeline control.
Dedicated hardware, such as an adder with a sum-of-products function (32 bits x 32 bits + (upper/lower) 32 bits) and barrel shifter (capable of 64-bit data shift) are built in to enable the high-speed processing of complicated instructions.

(2) Bus interface

Activates a required bus cycle according to the physical address acquired by the CPU. The bus interface unit supports both 32-bit bus mode, in which the external data bus has a 32-bit configuration, and 16-bit bus mode, in which it has a 16-bit configuration. It outputs appropriate control signals according to the mode set when a bus cycle is activated.

(3) Interrupt controller

Handles received hardware interrupt requests (nonmaskable and maskable interrupt requests) . The handler for maskable interrupts can be placed in the built-in instruction RAM.

(4) Write buffer

Stores data write (up to four data items) when the CPU performs write to external hardware. When data is written into the write buffer, the CPU no longer has to wait for the end of the bus cycle and can continue processing.

(5) Internal memory

16K-byte memory. This memory consists of four 4K-byte blocks, an instruction cache, data cache, instruction RAM, and data RAM. The instruction RAM uses direct mapping, while the data cache uses direct mapping/write-through.

[MEMO]

CHAPTER 2 REGISTER SETS

2.1 PROGRAM REGISTER SET

The V830 Family has two types of register sets: general-purpose register sets which can be used by programmers, and system register sets which control the execution environment. The width of all registers is 32 bits.

2.1.1 General-Purpose Register Set

(1) General-purpose registers

The V830 Family has 32 general-purpose registers, r0-r31, which can be used either as data registers or address registers. Note, however, that r0, r30, and r31 contain values that are fixed by hardware or which are used implicitly by instructions.

(a) Hardware-dependent registers

Hardware-dependent registers contain values that are fixed by hardware or which are used implicitly by instructions.

r0 :	Zero register Always contains 0.
r30 :	Register reserved for operation Serves as an auxiliary register which stores the result of a multiplication or division instruction.
r31 :	Link pointer The JAL instruction stores the return address in this register.

Remark The initial values of r1 to r31 are indefinite.

(b) Software-reserved registers

These registers are used by assemblers and compilers. To use them as registers for variables, first save their contents to guard against data loss or damage. When their use is no longer required restore the saved contents.

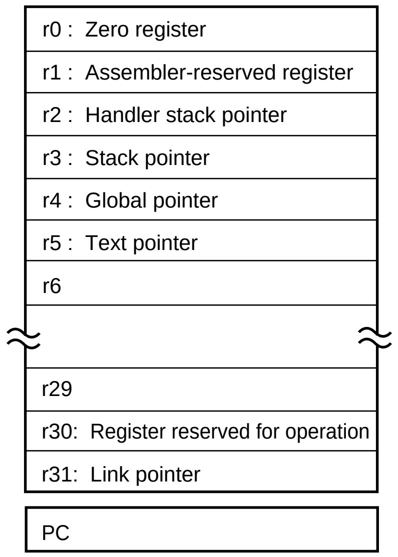
r1 :	Assembler-reserved register Serves as a working register for creating 32 bits of immediate data. It is used implicitly when the assembler calculates an effective address.
r2 :	Handler stack pointer Reserved as the stack pointer for a handler.
r3 :	Stack pointer Reserved for stack frame creation when a function is called.
r4 :	Global pointer Used when accessing a global variable in a data area.
r5 :	Text pointer Points to the beginning of a text area.

2.1.2 Program Counter (PC)

The program counter (PC) is a register which holds the first address of the instruction being executed. Bit 0 of the program counter is fixed to 0, but is forcibly masked to 0 upon a branch to a point other than a halfword boundary (bit 0 of the address is 0).

Upon reset, the program counter is initialized to FFFFFFF0H.

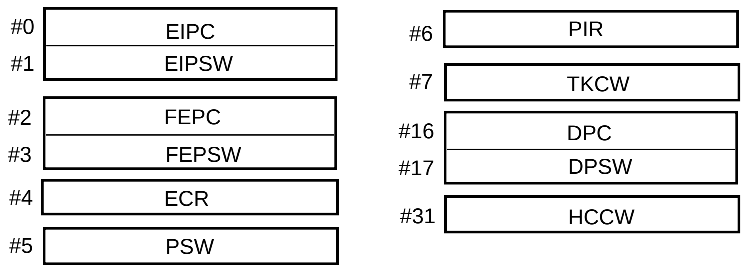
Figure 2-1. Program Registers



2.2 SYSTEM REGISTER SET

System registers are used to control the processor state, save exception/interruption information, and manage tasks. The V830 Family has eleven 32-bit system registers. These registers can be accessed using special instructions (LDSR and STSR instructions).

Figure 2-2. System Registers

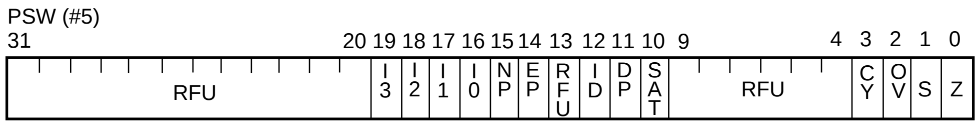


Remark The system register number is preceded by #.

2.2.1 Program Status Word (PSW)

The program status word is a set of flags indicating the program status (results of instruction execution) and the processor status. If the LDSR instruction is used to modify the fields in this register, the modification will become effective immediately after the LDSR instruction is executed.

The initial value is 00008000H.



Bit position	Field name	Meaning
31-20	RFU	Reserved (fixed to 0)
19-16	I3-I0	Interrupt Level Level of maskable interrupt enabled
15	NP	NMI Pending Indicates that an NMI is being handled. When an NMI is accepted, the NP bit is set to mask NMIs so that multiple interrupts will be disabled. NP = 0: NMI processing not in progress NP = 1: NMI processing in progress
14	EP	Exception Pending Indicates that an exception, trap, or interrupt is being handled. When an exception event occurs, this bit is set to mask interrupts. EP = 0: Exception, trap, or interrupt handling is not in progress. EP = 1: Exception, trap, or interrupt handling is in progress.
13	RFU	Reserved (must be fixed to 0)

12	ID	Interrupt Disable
		Indicates whether the V830 is ready to accept an external interrupt.
		ID = 0: Interrupts are enabled.
		ID = 1: Interrupts are disabled.

11	DP	Debug Pending Indicates that a fatal exception is being handled. DP = 1: Fatal exception handling is in progress. DP = 0: Fatal exception handling is not in progress.
10	SAT	Saturate Flag Indicates whether overflow has occurred during a saturatable arithmetic operation. The SAT bit is held until it is cleared. SAT = 1: Overflow has occurred SAT = 0: No overflow has occurred
9-4	RFU	Reserved (must be fixed to 0)

3	CY	Carry
		Indicates whether a carry occurred during an arithmetic operation.
		CY = 0: No carry occurred.
		CY = 1: A carry occurred.

2	OV	Overflow
		Indicates whether an overflow occurred during an arithmetic operation.
		OV = 0: No overflow occurred.
		OV = 1: Overflow occurred.

1	S	Sign Indicates whether the result of an operation is negative. S = 0: The result of the operation is positive or zero. S = 1: The result of the operation is negative.

Bit position	Field name	Meaning
0	Z	Zero Indicates whether the result of an operation is zero. Z = 0: The result of the operation is other than zero. Z = 1: The result of the operation is zero.

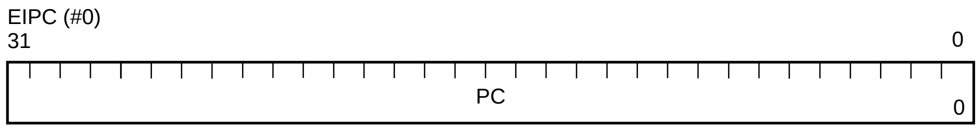
Remark RFU stands for Reserved for Future Use.

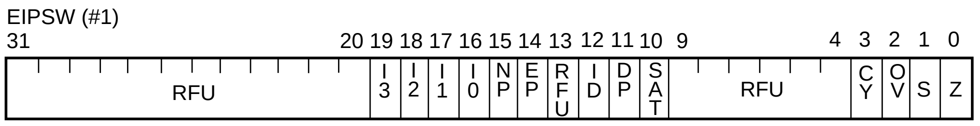
2.2.2 Exception/Interrupt Status Save Registers (EIPC and EIPSW)

EIPC and EIPSW are registers in which the contents of the PC and PSW will be saved when an exception or maskable interrupt occurs — EIPC for PC and EIPSW for PSW. There is only one pair of EIPC and EIPSW. If, therefore, it is necessary to enable multiple exceptions or multiple interrupts, the software designer must ensure that EIPC and EIPSW will be saved.

Bit 0 of EIPC and bits 31-20, 13, and 9-4 of EIPSW are fixed to 0. If an exception occurs when the EP bit of PSW is set (indicating that a double exception has occurred), the PC and PSW are not saved in EIPC and EIPSW, instead being saved in FEPC and FEPSW.

The initial values are indefinite.





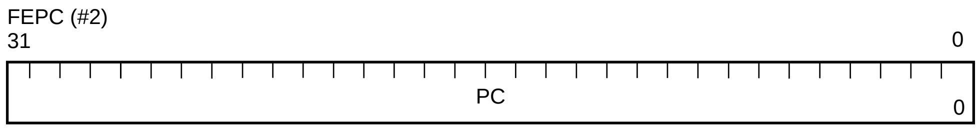
Remark RFU stands for Reserved for Future Use.

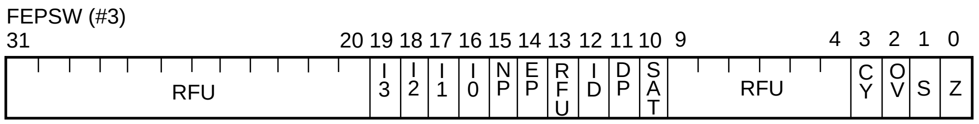
2.2.3 NMI/Double Exception Status Save Registers (FEPC and FEPSW)

When an NMI or double exception (exception that occurs when the EP bit of the PSW is 1) occurs, the PC and PSW are saved in these registers — FEPC for PC and FEPSW for PSW. Since saving to FEPC and FEPSW indicates a serious problem, prompt action is needed.

Bit 0 of FEPC and bits 31-20, 13, and 9-4 of FEPSW are fixed to 0.

The initial values are indefinite.





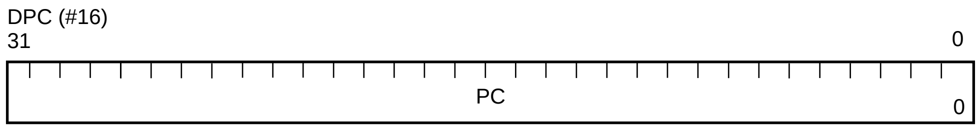
Remark RFU stands for Reserved for Future Use.

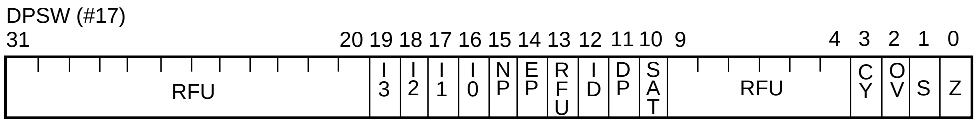
2.2.4 Fatal Exception Status Save Registers (DPC and DPSW)

When a fatal exception (exception that occurs when the NP bit of the PSW is set to 1) occurs, the PC and PSW are saved in these registers — DPC in PC and DPSW in PSW. Since saving to DPC and DPSW indicates a serious problem, prompt action is needed.

Bit 0 of DPC and bits 31-20, 13, and 9-4 of DPSW are fixed to 0.

The initial values are indefinite.





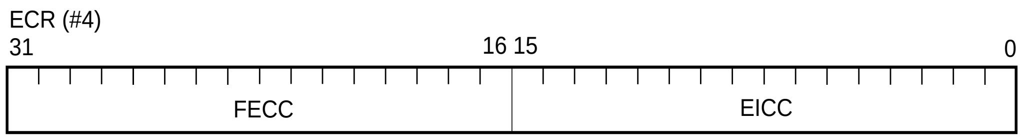
Remark RFU stands for Reserved for Future Use.

2.2.5 Exception Cause Register (ECR)

When an exception, maskable interrupt, or NMI occurs, its cause is stored in this register. The value held in ECR is coded for each cause of exception (see **Chapter 6**).

ECR is read-only. It is impossible to write data in ECR using the LDSR instruction.

The initial value is 0000FFF0H.



Bit position	Field name	Meaning
31-16	FECC	Exception code of NMI or double exception
15-0	EICC	Exception code of exception or interrupt

* 2.2.6 Processor ID Register (PIR)

This register identifies the CPU type. Its value is shown below.

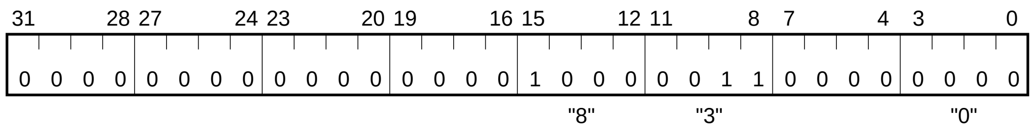
(1) V830: 00008300H
(2) V831: 00008301H
(3) V832: 00008302H

PIR is read-only. It is impossible to write data in PIR using the LDSR instruction.

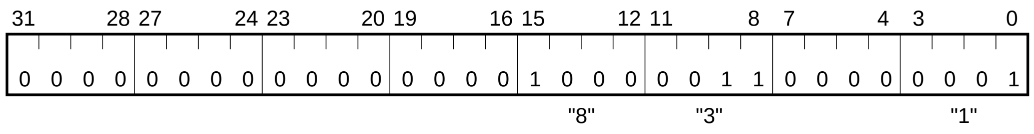
The value of the register is fixed.

PIR(#6)

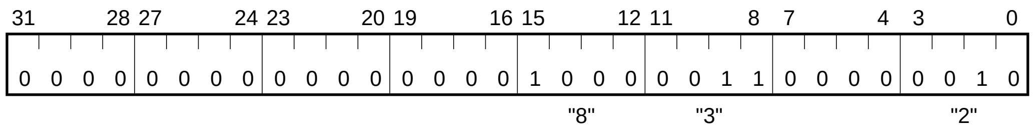
(1) V830



(2) V831



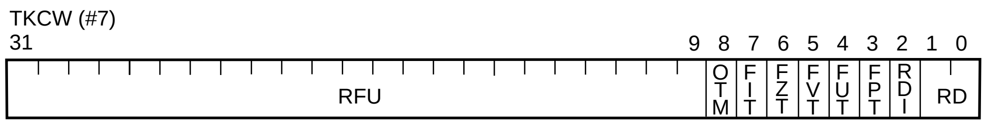
(3) V832



2.2.7 Task Control Word (TKCW)

This register is provided for task control. It is read-only. It is impossible to write data in TKCW using the LDSR instruction. It is currently not used, but is provided to ensure that compatibility is maintained.

The value is fixed to 000000E0H.

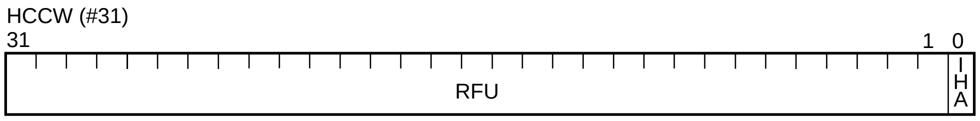


Remark RFU stands for Reserved for Future Use.

2.2.8 Hardware Configuration Control Word (HCCW)

This register specifies the maskable interrupt handler address.

The initial value is 00000000H.



Bit position	Field name	Meaning
31-1	RFU	Reserved (must be fixed to 0)
0	IHA	Interrupt Handler Address Indicates the address of the maskable interrupt handler. IHA = 1: FE0000n0H (built-in instruction RAM) IHA = 0: FFFFFEn0H (external memory) n: Interrupt level

Remark RFU stands for Reserved for Future Use.

2.3 SYSTEM REGISTER NUMBERS

For inputs from and outputs to the system registers, system register numbers are specified in the LDSR and STSR instructions, as follows:

No.	System register	Whether to allow operand specification	
		LDSR	STSR
0	EIPC : Exception/Interrupt PC	○	○
1	EIPSW : Exception/Interrupt PSW	○	○
2	FEPC : Fatal Error PC	○	○
3	FEPSW : Fatal Error PSW	○	○
4	ECR : Exception Cause Register	—	○
5	PSW : Program Status Word	○	○
6	PIR : Processor ID Register	—	○
7	TKCW : Task Control Word	—	○
8-15	Reserved		
16	DPC : Debug PC	○	○
17	DPSW : Debug PSW	○	○
18-30	Reserved		
31	HCCW : Hardware Configuration Control Word	○	○

— : Inhibited (inaccessible)
○ : Allowed (accessible)

[MEMO]

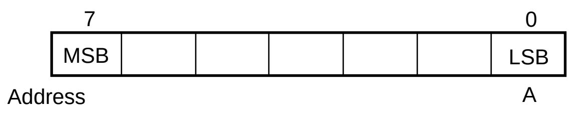
CHAPTER 3 DATA SETS

3.1 DATA TYPES

The V830 Family supports three data types: byte (8 bits), halfword (16 bits), and word (32 bits). Data of these types must be aligned with byte, halfword, or word boundaries, respectively. Addressing is based on little endian.

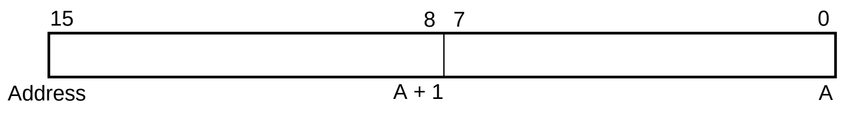
(1) Byte data

One byte of data consists of eight consecutive bits, each of which is named. Bit 0 is the LSB (Least Significant Bit) while bit 7 is the MSB (Most Significant Bit). This data can be placed at any address.



(2) Halfword data

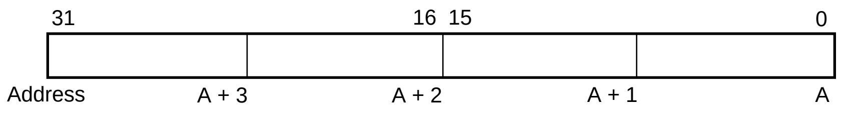
One halfword of data consists of 16 consecutive bits, each of which is named. Bit 0 is the LSB, while bit 15 is the MSB. Halfword data must be aligned with halfword boundaries (in address areas such that bit 0 of the address of the segment containing bit 0 is 0).



A = 2n (where n is a positive integer)

(3) Word data

One word of data consists of 32 consecutive bits, each of which is named. Bit 0 is the LSB and bit 31 is the MSB. Word data must be aligned with word boundaries (in address areas such that bits 0 and 1 of the address of the segment containing bit 0 are 0).



A = 4n (where n is a positive integer)

3.1.1 Integers

In the V830 Family, integers are represented by twos complements. They are expressed by bytes, halfwords, or words. Digit ordering for integers is as follows: Bit 0 is handled as the least significant bit, regardless of the data length. Larger bit numbers correspond to higher orders.

Data length	Range (in decimal notation)
Byte (8 bits)	–128 to +127
Halfword (16 bits)	–32,768 to +32,767
Word (32 bits)	–2,147,483,648 to +2,147,483,647

3.1.2 Unsigned Integers

Unsigned integers are of a data type for which the most significant bit is not handled as a sign bit, but all bits represent a positive integer. Data of this data type is represented by a binary number and has a size of a byte, halfword, or word. Digit ordering for unsigned integers is as follows: Bit 0 is handled as the least significant bit, regardless of the data length. Larger bit numbers correspond to higher orders.

Data length	Range (in decimal notation)
Byte (8 bits)	0 to 255
Halfword (16 bits)	0 to 65,535
Word (32 bits)	0 to 4,294,967,295

3.2 DATA ALIGNMENT

The V830 Family requires that data be aligned with appropriate boundaries: word boundaries for word data, halfword boundaries for halfword data, and byte boundaries for byte data. If a data alignment error is delected, the data address is automatically changed to an accessible address. It is impossible to predict whether this address change will lead to correct or incorrect data access. This change is made as follows:

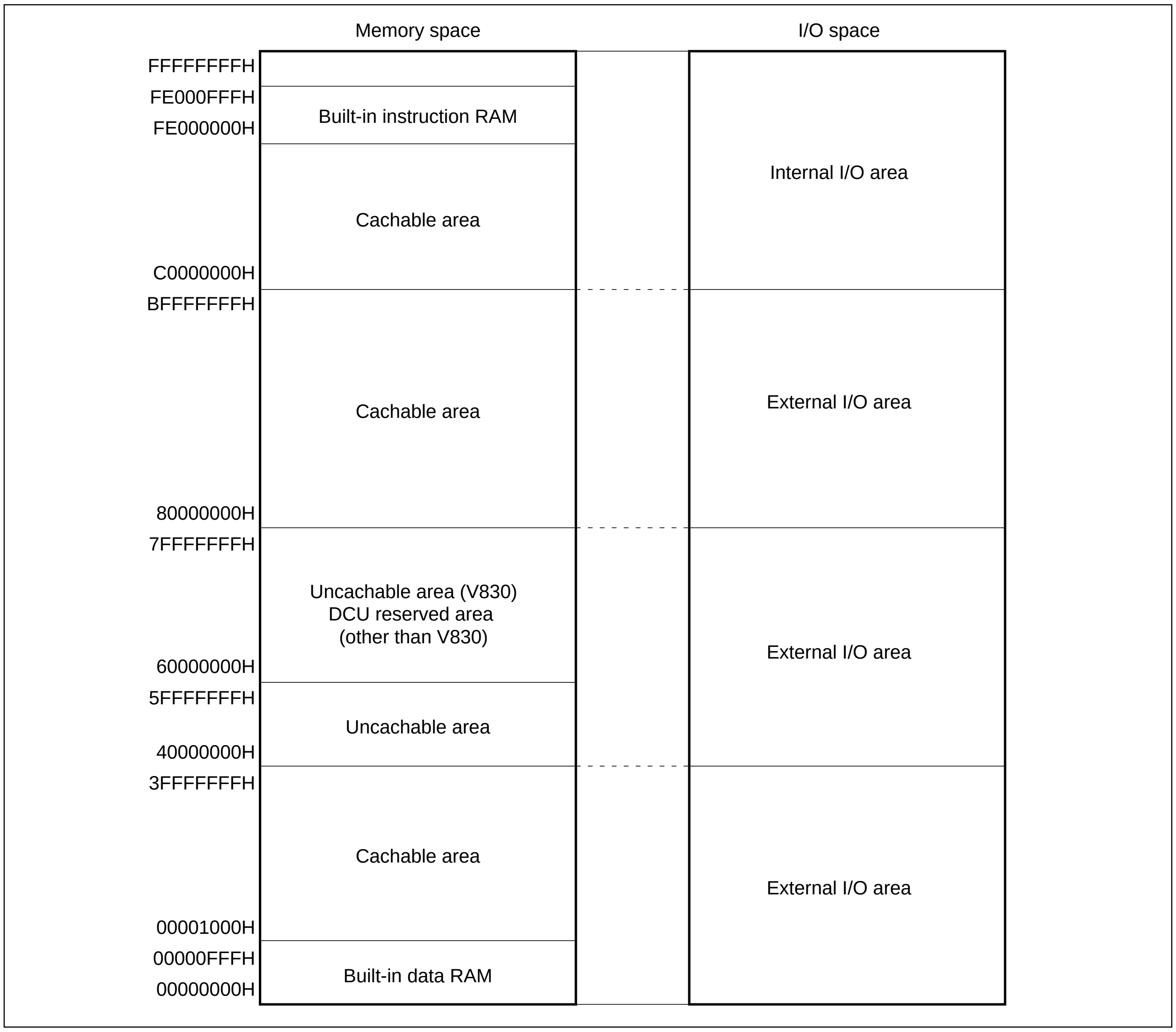
Data size	Method
Byte data	—
Halfword data	Bit 0 is masked to 0.
Word data	Bits 0 and 1 are masked to 0.

CHAPTER 4 ADDRESS SPACE

The V830 Family supports 4G-byte linear address spaces for both the memory space and I/O space. It assigns 32-bit addresses to the memory space. The maximum address is $2^{32} - 1$. It also assigns 32-bit addresses to the I/O space.

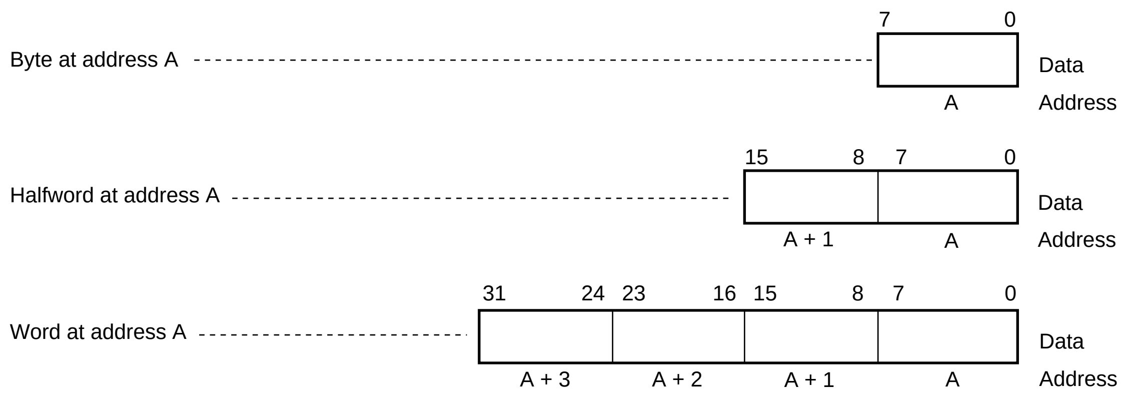
Figure 4-1 shows a memory map for the V830 Family.

Figure 4-1. Memory Map



Byte data aligned with each address is defined such that bits 0 and 7 are the LSB and MSB, respectively. If data consists of multiple bytes, it is defined such that the byte data at the low-order address contains the LSB and that at the high-order address contains the MSB (little-endian ordering), unless specified otherwise.

According to V830 Family terminology, data arranged in two-byte format is called halfword data, while that arranged in four-byte format is called word data. For data consisting of multiple bytes, the low-order address on the right and the high-order address on the left, as indicated below.



4.1 ADDRESSING MODE

The V830 Family generates two types of addresses, as follows:

- Instruction addresses (used by instructions involving branching)
- Operand addresses (used by instructions which access data)

4.1.1 Instruction Addresses

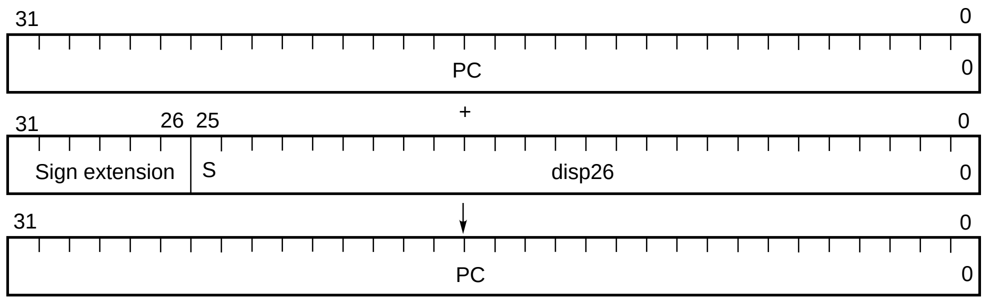
The instruction address is determined by the contents of the program counter (PC). Each time an instruction is executed, it is automatically incremented by 2 or 4, depending on the number of bytes constituting the instruction being fetched. When a branch instruction is executed, the branch address is set in the PC by the following addressing mode:

(1) Relative addressing (to PC)

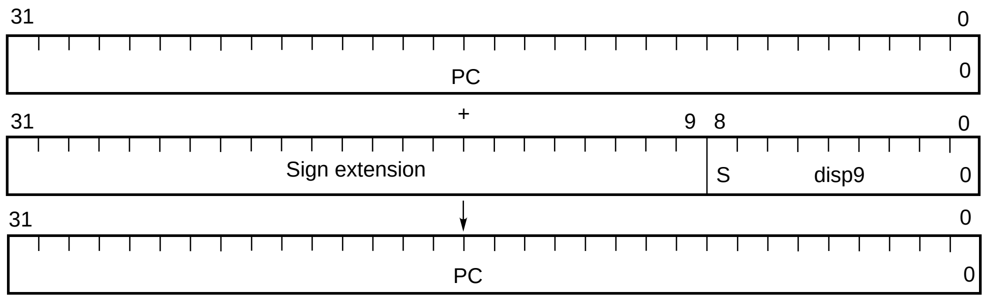
The signed 9 or 26 bits (displacement, or disp) of data contained in the operation code are added to the program counter (PC). For this addition, the displacement is handled as twos complement data. Bit 8 or 25 is the sign bit, respectively.

The JR, JAL, Bcond, and ABcond instructions use this addressing.

Addressing for JR and JAL instructions



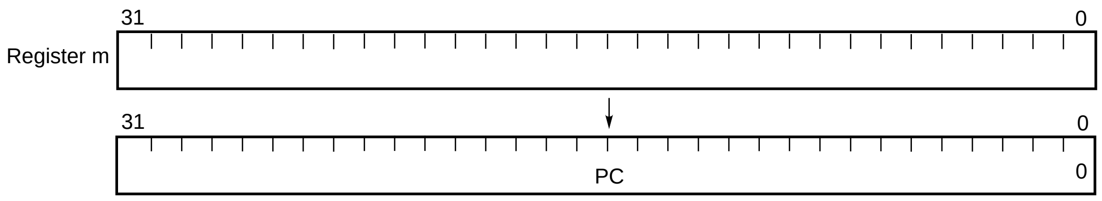
Addressing for Bcond and ABcond instructions



(2) Register addressing (via register)

The contents of the general-purpose register (r0-r31) designated in the instruction are transferred to the program counter (PC).

The JMP instruction uses this addressing.



4.1.2 Operand Addresses

(1) Register addressing

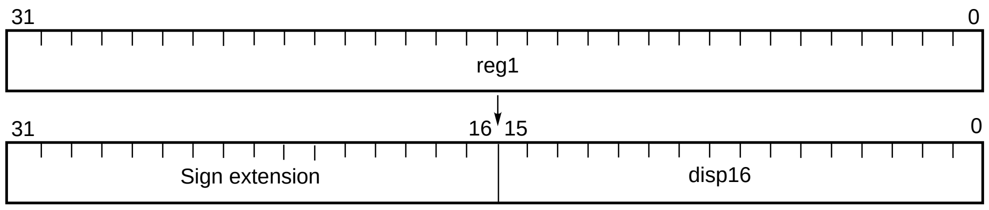
In this addressing mode, the general-purpose register designated in the general-purpose register designation field is accessed as an operand. This addressing is used by instructions whose operand format is reg1 or reg2.

(2) Immediate addressing

In this addressing mode, the 5 or 16 bits of data constituting the operation code are handled as an operand. This addressing is used by those instructions whose operand format is imm5 or imm16.

(3) Based addressing

In this addressing mode, when the memory area containing the operand is accessed, its address is determined from the sum of the contents of the general-purpose register designated by the address designation code and the 16-bit displacement in the instruction. This addressing is used by those instructions having an operand format of disp16[reg1].



CHAPTER 5 INSTRUCTIONS

5.1 INSTRUCTION FORMAT

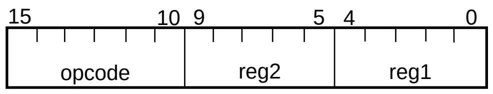
The V830 Family uses two instruction formats: 16-bit and 32-bit. The 16-bit instructions include binary operation, control, and conditional branch instructions, while the 32-bit instructions include load/store and I/O operation instructions, instructions for handling 16 bits of immediate data, and jump-and-link instructions.

Some instructions contain unused fields, which must be fixed to 0, which are provided for future use. When an instruction is actually loaded into memory, its configuration is as follows:

- Low-order part of each instruction format (including bit 0) → Low-order address
- High-order part of each instruction format (including bit 15 or 31) → High-order address

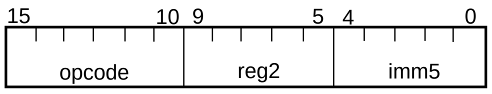
(1) reg-reg instruction format [FORMAT I]

This instruction format has a six-bit operation code field and two general-purpose register designation fields for operand specification, giving a total length of 16 bits.



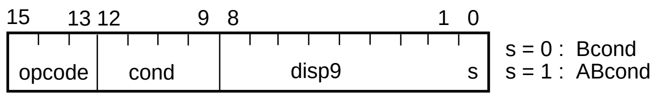
(2) imm-reg instruction format [FORMAT II]

This instruction format has a six-bit operation code field, a five-bit immediate data field, and a general-purpose register designation field, giving a total length of 16 bits.



(3) Conditional branch instruction format [FORMAT III]

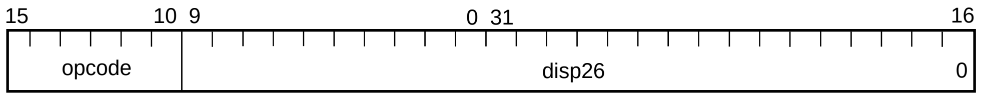
This instruction format has a three-bit operation code field, a four-bit condition code field, a nine-bit branch displacement field (bit 0 is handled as 0 and need not be specified), and a one-bit sub-operation code, giving a total length of 16 bits.



s : sub-opcode

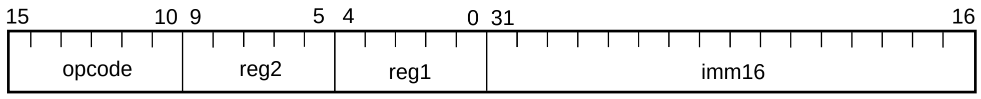
(4) Medium-distance jump instruction format [FORMAT IV]

This instruction format has a six-bit operation code field and a 26-bit displacement field (the lowest-order bit must be 0), giving a total length of 32 bits.



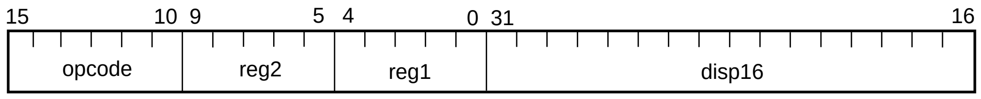
(5) Three-operand instruction format [FORMAT V]

This instruction format has a six-bit operation code field, two general-purpose register designation fields, and a 16-bit immediate data field, giving a total length of 32 bits.



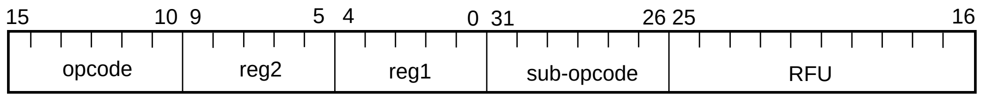
(6) Load/store instruction format [FORMAT VI]

This instruction format has a six-bit operation code field, two general-purpose register designation fields, and a 16-bit displacement field, giving a total length of 32 bits.



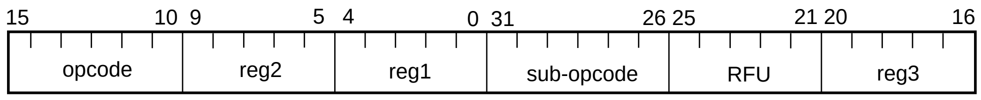
(7) Extended instruction format [FORMAT VII]

This instruction format has a six-bit operation code field, two general-purpose register designation fields, and a six-bit sub-operation code field, giving a total length of 32 bits.



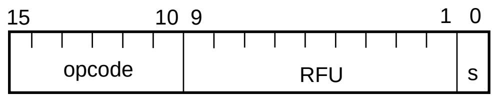
(8) Three-register operand instruction format [FORMAT VIII]

This instruction format has a six-bit operation code field, three general-purpose register designation fields, and a six-bit sub-operation code field, giving a total length of 32 bits.



(9) No-operand instruction format [FORMAT IX]

This instruction format has a six-bit operation code field and a one-bit sub-operation code field, giving a total length of 16 bits.



s : sub-opcode

5.2 OUTLINE OF INSTRUCTIONS

(1) Load/store instructions: For data transfer between memory and register

Mnemonic	Meaning
LD.B	Load Byte
LD.H	Load Halfword
LD.W	Load Word
ST.B	Store Byte
ST.H	Store Halfword
ST.W	Store Word
BILD	Block Instruction Load to built-in instruction RAM
BIST	Block Instruction Store from built-in instruction RAM
BDLD	Block Data Load to built-in data RAM
BDST	Block Data Store from built-in data RAM

(2) I/O instructions: For data transfer between I/O and registers

Mnemonic	Meaning
IN.B	Input Byte from port
IN.H	Input Halfword from port
IN.W	Input Word from port
OUT.B	Output Byte to port
OUT.H	Output Halfword to port
OUT.W	Output Word to port

(3) Arithmetic operation instructions: For addition, subtraction, multiplication, division, data comparison, and register-to-register data transfer

Mnemonic	Meaning
MOV	Move data
MOVHI	Move with addition of High-order Immediate data
ADD	Add
ADDI	Add Immediate data
MOVEA	Move with Addition
SUB	Subtract
MUL	Multiply (signed)
MULU	Multiply Unsigned
DIV	Divide (signed)
DIVU	Divide Unsigned
CMP	Compare
SETF	Set Flag condition
MIN3	Minimum on 3 operands
MAX3	Maximum on 3 operands

(4) Sum-of-products/saturatable operation instructions

Mnemonic	Meaning
MUL3	Multiply on 3 operands
MAC3	Multiply and Accumulate on 3 operands
MULI	Multiply on Immediate and register data
MACI	Multiply and Accumulate on Immediate and register data
MULT3	Multiply with Truncation on 3 operands
MACT3	Multiply and Accumulate with Truncation on 3 operands
SATADD3	Saturatable Addition on 3 operands
SATSUB3	Saturatable Subtraction on 3 operands

(5) Logical operation instructions

Mnemonic	Meaning
OR	OR (disjunction)
ORI	OR of Immediate data and register data
AND	AND (conjunction)
ANDI	AND of Immediate data and register data
XOR	Exclusive OR
XORI	Exclusive OR of Immediate and register data
NOT	NOT (ones compliment)
SHL	Shift Logical to the Left
SHR	Shift Logical to the Right
SAR	Shift Arithmetic to the Right
SHLD3	Shift to the Left of Double word on 3 operands
SHRD3	Shift to the Right of Double word on 3 operands

(6) Branch instructions: Unconditional branch instruction, conditional branch instructions which change control according to the setting of a flag, and high-speed (advanced) branch instructions which make use of branch history

Mnemonic	Meaning
JMP	Jump unconditional (via register)
JR	Jump Relative to PC, unconditional
JAL	Jump and Link
ABGT BGT	Advanced Branch on Greater than signed Branch on Greater than signed
ABGE BGE	Advanced Branch on Greater than or Equal signed Branch on Greater than or Equal signed
ABLT BLT	Advanced Branch on Less than signed Branch on Less than signed
ABLE BLE	Advanced Branch on Less than or Equal signed Branch on Less than or Equal signed
ABH BH	Advanced Branch on Higher Branch on Higher
ABNL BNL	Advanced Branch on Not Lower Branch on Not Lower
ABL BL	Advanced Branch on Lower Branch on Lower
ABNH BNH	Advanced Branch on Not Higher Branch on Not Higher

Mnemonic	Meaning
ABE BE	Advanced Branch on Equal Branch on Equal
ABNE BNE	Advanced Branch on Not Equal Branch on Not Equal
ABV BV	Advanced Branch on Overflow Branch on Overflow
ABNV BNV	Advanced Branch on No Overflow Branch on No Overflow
ABN BN	Advanced Branch on Negative Branch on Negative
ABP BP	Advanced Branch on Positive Branch on Positive
ABC BC	Advanced Branch on Carry Branch on Carry
ABNC BNC	Advanced Branch on No Carry Branch on No Carry
ABZ BZ	Advanced Branch on Zero Branch on Zero
ABNZ BNZ	Advanced Branch on Not Zero Branch on Not Zero
ABR BR	Advanced Branch Always (unconditional) Branch Always (unconditional)
NOP	Not Always (no branching)

(7) Special instructions: Instructions other than those in (1) to (6) above

Mnemonic	Meaning
LDSR	Load to System Register
STSR	Store contents of System Register
TRAP	Software Trap
RETI	Return from Trap or Interrupt
CAXI	Compare and Exchange Interlocked
HALT	Halt
BRKRET	Break Return from fatal exception
EI	Enable maskable Interrupt
DI	Disable maskable Interrupt
STBY	Standby

5.3 INSTRUCTION SET

Format of explanations of each instruction

Instruction mnemonic	Meaning

[Syntax] Explains how to write the instruction, together with the required operands. The following abbreviations are used in the explanations of operands:

Abbreviation	Meaning
reg1	General-purpose register (used as a source register)
reg2	General-purpose register (used mainly as a destination register, but with some instructions, as a source register)
reg3	General-purpose register (used mainly as a destination register, but with some instructions, as a source register)
immx	x bits of immediate data
dispx	x-bit displacement
regID	System register number
vector adr	Trap handler address corresponding to trap vector

[Operation] Explains the function of the instruction. The following abbreviations are used:

Abbreviation	Meaning
←	Assignment
||	Bit concatenation
GR[x]	General-purpose register x
SR[x]	System register x
sign-extend (x)	Value x is subjected to sign extension to the length of one word.
zero-extend (x)	Value x is subjected to zero extension to the length of one word.
Load-Memory (x, y)	Data of size y is read from address x.
Store-Memory (x, y, z)	Data y is written to address x with size z.
Input-Port (x, y)	Data of size y is read from port address x.
Output-Port (x, y, z)	Data y is written to port address x with size z.
adr	Unsigned 32-bit address

[Format]	Identifies an instruction format by its number.
[Operation code]	Gives the operation code of an instruction by showing the bit pattern in the operation code field.
[Flags]	Explains how each flag operates.

Abbreviation	Meaning
—	No change
0	Change to 0
1	Change to 1

[Instruction]	Briefly explains the function of the instruction.
[Description]	Explains the operation of the instruction in detail.
[Supplement]	Gives a supplementary explanation.
[Exception]	Explains exceptions which could occur when the instruction is executed.

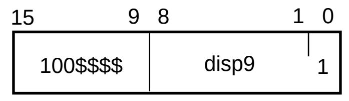
ABcond — Advanced Branch on condition

[Syntax] ABcond disp9

[Operation] if conditions are satisfied
then PC $\leftarrow$ PC + sign-extend(disp9)

[Format] Format III

[Operation code]



The $$$$ field indicates the condition (see **Table 5-1**).

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] ABcond - Advanced branch on condition according to a code having a 9-bit displacement

[Description] The condition flag specified in the instruction is tested. If the condition is satisfied, the instruction sets the PC to the sum of the current PC value and the 9 bits sign-extended to a word, transfers control according to the resulting PC value, and leaves a branch history.
High-speed branching is assured when an instruction with a branch history is executed. However, since only one branch history can be held, the only instruction carrying a branch history is the ABcond instruction executed last.
Bit 0 of the 9-bit displacement is masked to 0. Since the current PC value used for calculation is the start address of the ABcond instruction itself, the branch destination will be the instruction itself if the displacement is 0.

[Supplement] The branch history is erased if one of the following conditions is satisfied:

- Reset
- Execution of BILD instruction (instruction transfer from external memory to built-in instruction RAM)
- Rewriting of IRAMR register (built-in instruction RAM change)
- Clearing of instruction cache
- Rewriting of instruction cache tag

Pay careful attention to the following when loading a program:

- Because the program is loaded into built-in instruction RAM only by the BILD instruction, the branch history is automatically erased.
- When the program is loaded into the cachable area, the branch history is erased by clearing the instruction cache (setting the ICC bit of the cache memory control register (CMCR) to 1).
- When the program is loaded into the uncachable area, erase the previous branch history by executing the ABR instruction. If the user does not erase it, an incorrect branch occurs when the previous branch history points to the program area which was rewritten.

[Exception] None

Table 5-1. Conditional Branch Instructions (ABcond Instructions)

Instruction		Bits 12-9	Status of condition flag	Branch condition
Integer	ABGT	1111	((S xor OV) or Z) = 0	Greater than signed
	ABGE	1110	(S xor OV) = 0	Greater than or equal signed
	ABLT	0110	(S xor OV) = 1	Less than signed
	ABLE	0111	((S xor OV) or Z) = 1	Less than or equal signed
Unsigned integer	ABH	1011	(CY or Z) = 0	Higher (Greater than)
	ABNL	1001	CY = 0	Not lower (Greater than or equal)
	ABL	0001	CY = 1	Lower (Less than)
	ABNH	0011	(CY or Z) = 1	Not higher (Less than or equal)
Common	ABE	0010	Z = 1	Equal
	ABNE	1010	Z = 0	Not equal
Other	ABV	0000	OV = 1	Overflow
	ABNV	1000	OV = 0	No overflow
	ABN	0100	S = 1	Negative
	ABP	1100	S = 0	Positive
	ABC	0001	CY = 1	Carry
	ABNC	1001	CY = 0	No carry
	ABZ	0010	Z = 1	Zero
	ABNZ	1010	Z = 0	Not zero
	ABR	0101	—	Always (unconditional)

ADD   Add

[Syntax]
(1) ADD reg1, reg2
(2) ADD imm5, reg2

[Operation]
(1) GR[reg2] ← GR[reg2] + GR[reg1]
(2) GR[reg2] ← GR[reg2] + sign-extend(imm5)

[Format]
(1) Format I
(2) Format II

[Operation code]

(1)

15 – 10	9 – 5	4 – 0
000001	reg2	reg1

(2)

15 – 10	9 – 5	4 – 0
010001	reg2	imm5

[Flags]

CY	:	Assumes 1 if there is a carry from the MSB. Otherwise, assumes 0.
OV	:	Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S	:	Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z	:	Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction]
(1) ADD - Add the contents of registers
(2) ADD - Add the contents of a register and immediate data (5 bits)

[Description]
(1) The instruction adds the word in reg1 to the word in reg2 and stores the sum in reg2. The contents of reg1 remain unchanged.
(2) The instruction adds the 5 bits of immediate data, sign-extended to a word, to the word in reg2 then stores the sum in reg2.

[Exception]
None

ADDI Add Immediate data

[Syntax] ADDI imm16, reg1, reg2

[Operation] GR[reg2] ← GR[reg1] + sign-extend(imm16)

[Format] Format V

[Operation code]

15 10	9 5	4 0	31 16
101001	reg2	reg1	imm16

[Flags]

CY	:	Assumes 1 if there is a carry from the MSB. Otherwise, assumes 0.
OV	:	Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S	:	Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z	:	Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] ADDI - Add the contents of a register and immediate data (16 bits)

[Description] The instruction adds the 16 bits of immediate data, sign-extended to a word, to the word in reg1 then stores the sum in reg2. The contents of reg1 remain as is.

[Exception] None

AND

AND (conjunction)

[Syntax] AND reg1, reg2

[Operation] GR[reg2] ← GR[reg2] AND GR[reg1]

[Format] Format I

[Operation code]

15 10	9 5	4 0
001101	reg2	reg1

[Flags]

CY : —
OV : 0
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] AND - AND of registers

[Description] The instruction ANDs the words in reg1 and reg2 then stores the result in reg2. The contents of reg1 remain as is.

[Exception] None

ANDI

AND of Immediate data and register data

[Syntax] ANDI imm16, reg1, reg2

[Operation] GR[reg2] ← GR[reg1] AND zero-extend(imm16)

[Format] Format V

[Operation code]

15 10	9 5	4 0	31 0
101101	reg2	reg1	imm16

[Flags]
CY : —
OV : 0
S : 0
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] ANDI - AND contents of a register and immediate data (16 bits)

[Description] The instruction ANDs the 16 bits of immediate data, zero-extended to a word, and the word in reg1 then stores the result in reg2. The contents of reg1 remain as is.

[Exception] None

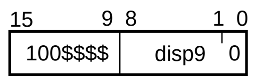
Bcond — Branch on condition

[Syntax] Bcond disp9

[Operation] if condition are satisfied
  then PC ← PC + (sign-extend)disp9

[Format] Format III

[Operation code]



The $$$$ field indicates the condition (see **Table 5-2**).

[Flags]

CY	: —
OV	: —
S	: —
Z	: —

[Instruction] Bcond - Branch on condition according to a code having a 9-bit displacement

[Description] The condition flag specified in the instruction is tested. If the condition is satisfied, the instruction sets the PC to the sum of the current PC value and the 9-bit displacement, sign-extended to a word, then transfers control according to the resulting PC value. Bit 0 of the 9-bit displacement is masked to 0. Since the current PC value used for calculation is the start address of the Bcond instruction itself, the branch destination will be the instruction itself if the displacement is 0.

[Exception] None

Table 5-2. Conditional Branch Instructions (Bcond Instructions)

Instruction		Bits 12-9	Status of condition flag	Branch condition
Integer	BGT	1111	((S xor OV) or Z) = 0	Greater than signed
	BGE	1110	(S xor OV) = 0	Greater than or equal signed
	BLT	0110	(S xor OV) = 1	Less than signed
	BLE	0111	((S xor OV) or Z) = 1	Less than or equal signed
Unsigned integer	BH	1011	(CY or Z) = 0	Higher (Greater than)
	BNL	1001	CY = 0	Not lower (Greater than or equal)
	BL	0001	CY = 1	Lower (Less than)
	BNH	0011	(CY or Z) = 1	Not higher (Less than or equal)
Common	BE	0010	Z = 1	Equal
	BNE	1010	Z = 0	Not equal
Other	BV	0000	OV = 1	Overflow
	BNV	1000	OV = 0	No overflow
	BN	0100	S = 1	Negative
	BP	1100	S = 0	Positive
	BC	0001	CY = 1	Carry
	BNC	1001	CY = 0	No carry
	BZ	0010	Z = 1	Zero
	BNZ	1010	Z = 0	Not zero
	BR	0101	—	Always (unconditional)
	NOP	1101	—	Not Always (no branch)

BDLD — Block Data Load to built-in data RAM

[Syntax] BDLD [reg1], [reg2]

[Operation] Store-internal-data-Memory(GR[reg2], Load-Memory(GR[reg1], 16 bytes), 16 bytes)

[Format] Format VII

[Operation code]

15 10	9 5	4 0	31 26	25 16
111110	reg2	reg1	100001	RFU

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] BDLD - Block data load to internal memory

[Description] The instruction transfers four words (16 bytes) of data from external memory to built-in data RAM. In the instruction, reg1 indicates the external memory address, while reg2 indicates the built-in data RAM offset address.
Bits 0-3 of reg1 and reg2 (addresses) must be 0.

[Exception] None

BDST — Block Data Store from built-in data RAM

[Syntax] BDST [reg2], [reg1]

[Operation] Store-Memory(GR[reg1], Load-internal-data-Memory(GR[reg2], 16 bytes), 16 bytes)

[Format] Format VII

[Operation code]

15 10	9 5	4 0	31 26	25 16
111110	reg2	reg1	100011	RFU

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] BDST - Block data store from internal data memory to external memory

[Description] The instruction transfers four words (16 bytes) of data from built-in data RAM to external memory. In the instruction, reg2 indicates the built-in data RAM offset address, while reg1 indicates the external memory address.
Bits 0-3 of reg1 and reg2 (addresses) must be 0.

[Exception] None

BILD — Block Instruction Load to built-in instruction RAM

[Syntax] BILD [reg1], [reg2]

[Operation] Store-internal-instruction-Memory(GR[reg2], Load-Memory(GR[reg1], 16 bytes), 16 bytes)

[Format] Format VII

[Operation code]

15 … 10	9 … 5	4 … 0	31 … 26	25 … 16
111110	reg2	reg1	100000	RFU

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] BILD - Block instruction load to internal memory

[Description] The instruction transfers four words (16 bytes) of data from external memory to built-in instruction RAM. In the instruction, reg1 indicates the external memory address, while reg2 indicates the built-in instruction RAM offset address.
Bits 0-3 of reg1 and reg2 (addresses) must be 0.

[Supplement] When the BILD instruction is executed, the branch history for the ABcond instruction (high-speed branching) is erased.

[Exception] None

BIST — Block Instruction Store from built-in instruction RAM

[Syntax] BIST [reg2], [reg1]

[Operation] Store-Memory(GR[reg1], Load-internal-instruction-Memory(GR[reg2], 16 bytes), 16 bytes)

[Format] Format VII

[Operation code]

15–10	9–5	4–0	31–26	25–16
111110	reg2	reg1	100010	RFU

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] BIST - Block instruction store from internal instruction memory to external memory

[Description] The instruction transfers four words (16 bytes) of data from built-in instruction RAM to external memory. In the instruction, reg2 indicates the built-in instruction RAM offset address, while reg1 indicates the external memory address.
Bits 0-3 of reg1 and reg2 (addresses) must be 0.

[Exception] None

BRKRET — Break Return from fatal exception

[Syntax] BRKRET

[Operation] PC ← DPC
PSW ← DPSW

[Format] Format IX

[Operation code]

15 10	9 1	0
011001	RFU	1

[Flags] CY : —
OV : —
S : —
Z : —

[Instruction] BRKRET - Break return

[Description] The instruction effects a return from a fatal exception by fetching the PC and PSW from the DPC and DPSW system registers.
When the instruction is executed, the return PC and PSW are retrieved from the DPC and DPSW. The retrieved return PC and PSW are set in the PC and PSW so that program execution will jump to the PC.

[Supplement] Use this instruction only when processing is needed for a return from a fatal exception.

[Exception] None

CAXI — Compare And Exchange Interlocked

[Syntax] CAXI disp16[reg1], reg2

[Operation]

```
locked
adr ← GR[reg1] + (sign-extend)disp16
tmp ← Load-Memory(adr,Word)
if GR[reg2] = tmp(comparison;result ← GR[reg2] - tmp)
      then   Store-Memory(adr, GR[30], Word)
             GR[reg2] ← tmp
      else   Store-Memory(adr, tmp, Word)
             GR[reg2] ← tmp
unlocked
```

[Format] Format VI

[Operation code]

15 10	9 5	4 0	31 16
111010	reg2	reg1	disp16

[Flags]

CY : Assumes 1 if comparison involves a borrow from the MSB. Otherwise, assumes 0.
OV : Assumes 1 if comparison has encountered overflow. Otherwise, assumes 0.
S : Assumes 1 if the comparison result is negative. Otherwise, assumes 0.
Z : Assumes 1 if the comparison result is zero. Otherwise, assumes 0.

[Instruction] CAXI - Compare and exchange interlocked

[Description] The instruction synchronizes the processors of a multi-processor system. The data specified by disp16[reg1] is used for synchronization (a lock word, for example). The condition prior to the execution of the instruction is as follows:

Newly set lock word	GR[30]
Previously read lock word	GR[reg2]
Lock word	The lock word is the word at the address specified by GR[reg1] + (sign-extend)disp16. Bits 0 and 1 of the address are masked to 0.

In this condition, the CAXI instruction performs the following:

(1) Locks the bus to prevent access by other processors.
(2) Fetches the lock word.
(3) Compares the lock word with the previously read lock word and sets the flags such that they reflect the result of the comparison.
(4) If the new and old lock words match, it indicates that the conditions under which the previous access was made are still effective (no lock due to access by a program running on another processor).
Since execution of the CAXI instruction changes the condition, the instruction sets the lock word in GR[30] (new lock word).
(5) If the new and old lock words do not match, it indicates that the conditions under which the previous access was made are no longer effective (lock due to access by a program running on another processor). Therefore, the instruction sets the lock word in GR[reg2] to determine the condition assumed by the lock word.
(6) Unlocks the bus.

[Exception] None

CMP — Compare

[Syntax] (1) CMP reg1, reg2
(2) CMP imm5, reg2

[Operation] (1) result ← GR[reg2] - GR[reg1]
(2) result ← GR[reg2] - sign-extend(imm5)

[Format] (1) Format I
(2) Format II

[Operation code]

(1)

15 10	9 5	4 0
000011	reg2	reg1

(2)

15 10	9 5	4 0
010011	reg2	imm5

[Flags] CY : Assumes 1 if there is a borrow from the MSB. Otherwise, assumes 0.
OV : Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S : Assumes 1 if the result is negative. Otherwise, assumes 0.
Z : Assumes 1 if the result is zero. Otherwise, assumes 0.

[Instruction] (1) CMP - Compare registers
(2) CMP - Compare register and immediate data (5 bits)

[Description] (1) The instruction compares the words in reg2 and reg1 and sets the condition flag according to the result. This comparison involves subtracting the contents of reg1 from those of reg2. The contents of reg1 and reg2 remain as is.
(2) The instruction compares the word in reg2 with the five bits of immediate data, sign-extended to a word, and sets the condition flag according to the result. This comparison involves subtracting the five bits of immediate data, sign-extended to a word, from the word in reg2. The contents of reg2 remain as is.

[Exception] None

DI — Disable maskable Interrupt

[Syntax]	DI
[Operation]	Sets the ID bit in the PSW to disable maskable interrupts.
[Format]	Format II
[Operation code]	see below

15 … 10	9 … 0
011110	RFU

[Flags]	CY : — OV : — S : — Z : —
[Instruction]	DI - Disable interrupts
[Description]	The instruction disables maskable interrupts by setting the ID bit in the PSW to 1. This has the same effect as when the LDSR instruction is used to set the PSW ID bit to 1.
[Supplement]	The DI instruction cannot disable nonmaskable interrupts. To disable nonmaskable interrupts, use the LDSR instruction to rewrite the PSW.
[Exception]	None

DIV — Divide (signed)

[Syntax]	DIV reg1, reg2

[Operation]	GR[30] ← GR[reg2] MOD GR[reg1](signed)
	GR[reg2] ← GR[reg2] ÷ GR[reg1](signed)

[Format]	Format I

[Operation code]

15 10	9 5	4 0
001001	reg2	reg1

[Flags]

CY :	—
OV :	Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S :	Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z :	Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] DIV - Divide

[Description] The instruction divides the word in reg2 by the word in reg1 (signed operands) and stores the quotient in reg2 and the remainder in r30. This division is conducted such that the sign of the remainder matches the sign of the dividend. The contents of reg1 remain as is. If r30 is designated as reg2, the quotient is stored in r30. Overflow occurs when the negative maximum (80000000H) is divided by -1 (FFFFFFFFH). In this case, reg2 contains the negative maximum and r30 contains 0.

[Exception] Division-by-zero exception

[Caution] If the word in reg1 is 0, a division-by-zero exception occurs, causing a trap to the exception handler. In this case, the contents of reg2, r30, and the flags remain as is.

DIVU — Divide Unsigned

[Syntax] DIVU reg1, reg2

[Operation] GR[30] $\leftarrow$ GR[reg2] MOD GR[reg1](unsigned)
GR[reg2] $\leftarrow$ GR[reg2] $\div$ GR[reg1](unsigned)

[Format] Format I

[Operation code]

15 … 10	9 … 5	4 … 0
001011	reg2	reg1

[Flags]
CY : —
OV : 0
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] DIVU - Divide unsigned value

[Description] The instruction divides the word in reg2 by the word in reg1 (unsigned operands) and stores the quotient in reg2 and the remainder in r30. The contents of reg1 remain as is. If r30 is designated as reg2, the quotient is stored in r30. The flags are set as if the results were signed data.

[Exception] Division-by-zero exception

[Caution] If the word in reg1 is 0, a division-by-zero exception occurs, causing a trap to the exception handler. In this case, the contents of reg2, r30, and the flags remain as is.

EI — Enable maskable Interrupt

[Syntax] EI

[Operation] Clears the ID bit in the PSW to enable maskable interrupts.

[Format] Format II

[Operation code]

15 … 10	9 … 0
010110	RFU

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] EI - Enable interrupts

[Description] The instruction enables maskable interrupts by resetting the ID bit in the PSW to 0. This produces the same effect as when the LDSR instruction is used to reset the PSW ID bit to 0.

[Supplement] The EI instruction cannot enable nonmaskable interrupts. To enable nonmaskable interrupts, use the LDSR instruction to rewrite the PSW.

[Exception] None

HALT

Halt

[Syntax]	HALT
[Operation]	Stops program execution.
[Format]	Format IX

[Operation code]

15 … 10	9 … 1	0
011010	RFU	0

[Flags]	CY : —
	OV : —
	S : —
	Z : —
[Instruction]	HALT - Halt
[Description]	The instruction stops the CPU and places it in sleep mode.
[Exception]	None

IN — Input from port

[Syntax]
(1) IN.B disp16[reg1], reg2
(2) IN.H disp16[reg1], reg2
(3) IN.W disp16[reg1], reg2

[Operation]
(1) adr ← GR[reg1] + (sign-extend)disp16

GR[reg2] $\xleftarrow{\text{zero-extend}}$ Input-Port(adr, Byte)

(2) adr ← GR[reg1] + (sign-extend)disp16

GR[reg2] $\xleftarrow{\text{zero-extend}}$ Input-Port(adr, Halfword)

(3) adr ← GR[reg1] + (sign-extend)disp16

GR[reg2] $\longleftarrow$ Input-Port(adr, Word)

[Format]
Format VI

[Operation code]

15 10	9 5	4 0	31 16
1110*$	reg2	reg1	disp16

(*$: 00 = (1), 01 = (2), 11 = (3))

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction]
(1) IN.B - Input byte from port
(2) IN.H - Input halfword from port
(3) IN.W - Input word from port

[Description]
(1) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit port address. It reads a byte of data from the resulting port address, zero-extends the read byte to a word, then stores the result in reg2.
(2) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit port address. It reads a halfword of data from the resulting port address, zero-extends the read halfword to a word, then stores the result in reg2. Bit 0 of the unsigned 32-bit address is masked to 0.

(3) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit port address. It reads a word of data from the resulting port address then stores the word in reg2. Bits 0 and 1 of the unsigned 32-bit address are masked to 0.

[Exception] None

JAL

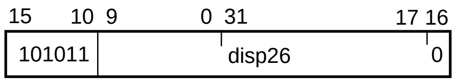
Jump and Link

[Syntax] JAL disp26

[Operation] GR[31] ← PC + 4
PC ← PC + (sign-extend)disp26

[Format] Format IV

[Operation code]



[Flags] CY : —
OV : —
S : —
Z : —

[Instruction] JAL - Jump and link

[Description] The instruction adds 4 to the current PC, saves the sum in r31, adds the 26-bit displacement, sign-extended to a word, to the current PC, sets the sum in the PC, then transfers control according to the newly set PC. The lowest-order bit of the 26-bit displacement is masked to 0. Since the current PC value used for calculation is the start address of the JAL instruction itself, the branch destination will be the instruction itself if the displacement is 0.

[Exception] None

JMP Jump unconditional (via register)

[Syntax] JMP [reg1]

[Operation] PC ← GR[reg1]

[Format] Format I

[Operation code]

15 10	9 5	4 0
000110	—	reg1

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] JMP - Jump to register-specified address

[Description] The instruction transfers control to the address specified by reg1. Bit 0 of the address is masked to 0.

[Exception] None

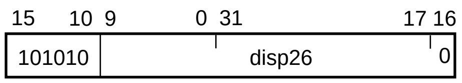
JR — Jump Relative to PC, unconditional

[Syntax] JR disp26

[Operation] PC ← PC + (sign-extend)disp26

[Format] Format IV

[Operation code]



[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] JR - Jump to relative address

[Description] The instruction sets the PC to the sum of the current PC and the 26-bit displacement, sign-extended to a word, then transfers control according to the newly set PC. Bit 0 of the 26-bit displacement is masked to 0.
Since the current PC value used for calculation is the start address of the JR instruction itself, the branch destination will be the instruction itself if the displacement is 0.

[Exception] None

LD Load

[Syntax]
(1) LD.B disp16[reg1], reg2
(2) LD.H disp16[reg1], reg2
(3) LD.W disp16[reg1], reg2

[Operation]
(1) adr ← GR[reg1] + (sign-extend)disp16

$$\text{GR[reg2]} \xleftarrow{\text{sign-extend}} \text{Load-Memory(adr, Byte)}$$

(2) adr ← GR[reg1] + (sign-extend)disp16

$$\text{GR[reg2]} \xleftarrow{\text{sign-extend}} \text{Load-Memory(adr, Halfword)}$$

(3) adr ← GR[reg1] + (sign-extend)disp16

$$\text{GR[reg2]} \longleftarrow \text{Load-Memory(adr, Word)}$$

[Format]
Format VI

[Operation code]

15 10	9 5	4 0	31 16
1100∗$	reg2	reg1	disp16

(∗$: 00 = (1), 01 = (2), 11 = (3))

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction]
(1) LD.B - Load byte
(2) LD.H - Load halfword
(3) LD.W - Load word

[Description]
(1) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit address. It reads a byte of data from the resulting address, sign-extends the read byte to a word, then stores the result in reg2.
(2) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit address. It reads a halfword of data from the resulting address, sign-extends the read halfword to a word, then stores the result in reg2. Bit 0 of the unsigned 32-bit address is masked to 0.

(3) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit address. It reads a word of data from the resulting address then stores the word in reg2. Bits 0 and 1 of the unsigned 32-bit address are masked to 0.

[Exception] None

LDSR — Load to System Register

[Syntax] LDSR reg2, regID

[Operation] SR[regID] ← GR[reg2]

[Format] Format II

[Operation code]

15 10	9 5	4 0
011100	reg2	regID

[Flags]
CY : — (See Supplement)
OV : — (See Supplement)
S : — (See Supplement)
Z : — (See Supplement)

[Instruction] LDSR - Load to system register

[Description] The instruction loads the word contained in reg2 to the system register designated by the system register number (regID). The contents of reg2 remain as is. System register numbers uniquely identify system registers. If the LDSR instruction is executed on a reserved system register or write-disabled system register, the operation of the instruction will be unpredictable.

[Exception] None

[Supplement] If the specified system register number (regID) is 5 (PSW), each flag assumes the value of the corresponding bit in reg2.

MAC3

Multiply and Accumulate on 3 operands (saturatable operation on signed 32-bit operands)

[Syntax] MAC3 reg1, reg2, reg3

[Operation] GR[reg3] ← saturate(GR[reg3] + GR[reg2] x GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	011101	RFU	reg3

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] MAC3 - Multiply and accumulate

[Description] The instruction multiplies the word in reg1 by that in reg2 as signed 32-bit integers, and adds the product to the data in reg3 as signed integers. If the sum falls outside the range of signed 32-bit integers that can be represented, it is regarded as causing an overflow (the low-order 32 bits of the 64 bits of the product are valid).

[If no overflow has occurred:]

The sum is stored into reg3.

[If an overflow has occurred:]

The SAT flag is set to 1. If the sum is positive, the positive maximum (7FFFFFFFH) is stored into reg3; if the sum is negative, the negative maximum (80000000H) is stored into reg3.

The contents of reg1 and reg2 remain as is.

[Supplement] A timing restriction is imposed on MAC3 instruction input operand reg3. If an instruction to update reg3 is not issued within three cycles before the issue of the MAC3 instruction, the MAC3 instruction will begin after a one-cycle halt (stall).

The flags (CY, OV, S, and Z) do not change. The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even if the result of a subsequent operation instruction is not saturated. To reset the SAT flag, use the LDSR instruction to rewrite the PSW.

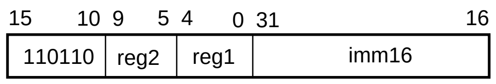
[Exception] None

MACI

Multiply and Accumulate on Immediate and register data

[Syntax]	MACI imm16, reg1, reg2
[Operation]	GR[reg2] ← saturate(GR[reg2] + GR[reg1] x sign-extend(imm16))
[Format]	Format V

[Operation code]



[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] MACI - Multiply and accumulate immediate and register data

[Description] The instruction multiplies the word in reg1 by the immediate data (16 bits, sign-extended to 32 bits) as signed integers then adds together the product and the data in reg2 as signed integers. If the sum falls outside the range of signed 32-bit integers that can be represented, it is regarded as causing an overflow (the low-order 32 bits of the 64 bits of the product are valid).

[If no overflow has occurred:]

The sum is stored into reg2.

[If an overflow has occurred:]

The SAT flag is set to 1. If the sum is positive, the positive maximum (7FFFFFFFH) is stored into reg2; if the sum is negative, the negative maximum (80000000H) is stored into reg2.

The contents of reg1 remain as is.

[Supplement] The flags (CY, OV, S, and Z) do not change. The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even if the result of a subsequent operation instruction is not saturated. To reset the SAT flag, use the LDSR instruction to rewrite the PSW.

[Exception] None

MACT3

Multiply and Accumulate with Truncation on 3 operands
(saturatable operation on signed 32-bit operands)

[Syntax] MACT3 reg1, reg2, reg3

[Operation] GR[reg3] ← saturate(GR[reg3] + high-order-32-bits(GR[reg2] x GR[reg1]))

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	011100	RFU	reg3

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] MACT3 - Multiply and accumulate with truncation

[Description] The instruction multiplies the word in reg1 by that in reg2 as signed integers, truncates the 64-bit product to discard the low-order 32 bits, then adds the high-order 32 bits of the product to the data in reg3 as signed integers.

[If no overflow has occurred:]

The sum is stored into reg3.

[If an overflow has occurred:]

The SAT flag is set to 1. If the sum is positive, the positive maximum (7FFFFFFFH) is stored into reg3; if the sum is negative, the negative maximum (80000000H) is stored into reg3.

The contents of reg1 and reg2 remain as is.

[Supplement] A timing restriction is imposed on the MACT3 instruction input operand reg3. If an instruction to update reg3 is not issued within three cycles before the issue of the MACT3 instruction, the MACT3 instruction will begin after a one-cycle halt (stall). The flags (CY, OV, S, and Z) do not change. The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even if the result of a subsequent operation instruction is not saturated. To reset the SAT flag, use the LDSR instruction to rewrite the PSW.

[Exception] None

MAX3 Maximum on 3 operands

[Syntax] MAX3 reg1, reg2, reg3

[Operation] GR[reg3] ← max(GR[reg2],GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	010011	RFU	reg3

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] MAX3 - Maximum

[Description] The instruction compares the words in reg1 and reg2 as signed integers and stores the larger value into reg3. The contents of reg1 and reg2 remain as is.

[Exception] None

MIN3 — Minimum on 3 operands

[Syntax] MIN3 reg1, reg2, reg3

[Operation] GR[reg3] $\leftarrow$ min(GR[reg2], GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	010010	RFU	reg3

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] MIN3 - Minimum

[Description] The instruction compares the words in reg1 and reg2 as signed integers and stores the smaller value into reg3. The contents of reg1 and reg2 remain as is.

[Exception] None

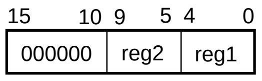
MOV

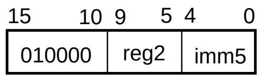
Move data

[Syntax]
(1) MOV reg1, reg2
(2) MOV imm5, reg2

[Operation]
(1) GR[reg2] ← GR[reg1]
(2) GR[reg2] ← sign-extend(imm5)

[Format]
(1) Format I
(2) Format II

[Operation code]
(1)


(2)


[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction]
(1) MOV - Move register data
(2) MOV - Move immediate data (5 bits)

[Description]
(1) The instruction copies the word in reg1 to reg2. The contents of reg1 remain as is.
(2) The instruction copies and transfers the 5 bits of immediate data, sign-extended to a word, to reg2.

[Exception]
None

MOVEA Move with Addition

[Syntax] MOVEA imm16, reg1, reg2

[Operation] GR[reg2] ← GR[reg1] + sign-extend(imm16)

[Format] Format V

[Operation code]

15 10	9 5	4 0	31 16
101000	reg2	reg1	imm16

[Flags] CY : —
OV : —
S : —
Z : —

[Instruction] MOVEA - Move with addition of 16-bit immediate data

[Description] The instruction adds the word in reg1 to the 16 bits of immediate data, sign-extended to a word, then stores the sum into reg2. The contents of reg1 remain as is. The flags do not change.

[Exception] None

MOVHI

Move with addition of High-order Immediate data

[Syntax] MOVHI imm16, reg1, reg2

[Operation] $GR[reg2] \leftarrow GR[reg1] + (imm16 \text{ II } 0^{16})$

[Format] Format V

[Operation code]

15 10	9 5	4 0	31 16
101111	reg2	reg1	imm16

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] MOVHI - Move with high-order immediate data addition

[Description] The instruction adds the word in reg1 to a word consisting of the high-order 16 bits of immediate data and the low-order 16 bits of 0 then stores the sum into reg2. The contents of reg1 remain as is. The flags do not change.

[Exception] None

MUL

Multiply (signed)

[Syntax] MUL reg1, reg2

[Operation]

result ← GR[reg2] x GR[reg1] (signed)
GR[30] ← result (high-order 32 bits)
GR[reg2] ← result (low-order 32 bits)

[Format] Format I

[Operation code]

15 10	9 5	4 0
001000	reg2	reg1

[Flags]

CY : —
OV : Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] MUL - Multiply

[Description] The instruction multiplies the word in reg1 by that in reg2 as signed data and stores the high-order 32 bits of the result (double word) in r30 and the low-order 32 bits in reg2. The contents of reg1 remain as is. If r30 is designated as reg2, the low-order 32 bits of the result are stored in r30. Overflow occurs when the double-word result is not equal to the low-order 32 bits, sign-extended to a double word.

[Exception] None

MUL3 Multiply on 3 operands (saturatable operation on signed 32-bit operands)

[Syntax] MUL3 reg1, reg2, reg3

[Operation] GR[reg3] ← saturate(GR[reg2] x GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	011111	RFU	reg3

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] MUL3 - Multiplication on 3 operands

[Description] The instruction multiplies the word in reg1 by that in reg2 as signed 32-bit integers. If the product falls outside the range of signed 32-bit integers that can be represented, it is regarded as causing an overflow (the low-order 32 bits of the 64 bits of the product are valid).

[If no overflow has occurred:]

The product is stored into reg3.

[If an overflow has occurred:]

The SAT flag is set to 1. If the product is positive, the positive maximum (7FFFFFFFH) is stored into reg3; if the product is negative, the negative maximum (80000000H) is stored into reg3.

The contents of reg1 and reg2 remain as is.

[Supplement] The flags (CY, OV, S, and Z) do not change. The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even if the result of a subsequent operation instruction is not saturated. To reset the SAT flag, use the LDSR instruction to rewrite the PSW.

[Exception] None

MULI

Multiply on Immediate and register data
(saturatable operation on signed 32-bit operands)

[Syntax] MULI imm16, reg1, reg2

[Operation] GR[reg2] ← saturate(GR[reg1] x sign-extend(imm16))

[Format] Format V

[Operation code]

15 10	9 5	4 0	31 16
110010	reg2	reg1	imm16

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] MULI - Multiplication involving immediate data

[Description] The instruction multiplies the word in reg1 by the 16 bits of immediate data (sign-extended to 32 bits) as signed integers. If the product fulls outside the range of signed 32-bit integers that can be represented, it is regarded as causing an overflow (the low-order 32 bits of the 64 bits of the product are valid).

[If no overflow has occurred:]

The product is stored into reg2.

[If an overflow has occurred:]

The SAT flag is set to 1. If the product is positive, the positive maximum (7FFFFFFFH) is stored into reg2; if the product is negative, the negative maximum (80000000H) is stored into reg2.

The contents of reg1 remain as is.

[Supplement] The flags (CY, OV, S, and Z) do not change. The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even it the result of a subsequent operation instruction is not saturated. To reset the SAT flag, use the LDSR instruction to rewrite the PSW.

[Exception] None

MULT3

Multiply with Truncation on 3 operands
(operation on signed 32-bit operands)

[Syntax] MULT3 reg1, reg2, reg3

[Operation] GR[reg3] ← high-order-32-bits(GR[reg2] x GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	011110	RFU	reg3

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] MULT3 - Multiplication on 3 operands with truncation

[Description] The instruction multiplies the word in reg1 by that in reg2 as signed integers, truncates the 64-bit product to discard the low-order into 32 bits, and stores only the high-order 32 bits into reg3. The contents of reg1 and reg2 remain as is.

[Exception] None

MULU — Multiply Unsigned

[Syntax] MULU reg1, reg2

[Operation] result ← GR[reg2] x GR[reg1] (unsigned)
GR[30] ← result (high-order 32 bits)
GR[reg2] ← result (low-order 32 bits)

[Format] Format I

[Operation code]

15 10	9 5	4 0
001010	reg2	reg1

[Flags]
CY : —
OV : Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] MULU - Multiply unsigned values

[Description] The instruction multiplies the word in reg1 by that in reg2 as unsigned data and stores the high-order 32 bits of the result (double word) into r30 and the low-order 32 bits into reg2. The contents of reg1 remain as is. If r30 is designated as reg2, the low-order 32 bits of the result are stored into r30. The flags are set as if the result were signed data. Overflow occurs when the double-word result is not equal to the low-order 32 bits, zero-extended to a double word.

[Exception] None

NOT

Not (ones compliment)

[Syntax] NOT reg1, reg2

[Operation] GR[reg2] $\leftarrow$ NOT(GR[reg1])

[Format] Format I

[Operation code]

15 10	9 5	4 0
001111	reg2	reg1

[Flags]
CY : —
OV : 0
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise assumes 0.

[Instruction] NOT - NOT

[Description] The instruction takes the NOT (ones complement) of the word in reg1 and stores the result into reg2. The contents of reg1 remain as is.

[Exception] None

OR

OR (disjunction)

[Syntax]	OR reg1, reg2
[Operation]	GR[reg2] ← GR[reg2] OR GR[reg1]
[Format]	Format I
[Operation code]	see below

15 … 10	9 … 5	4 … 0
001100	reg2	reg1

[Flags]	CY : —
	OV : 0
	S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
	Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.
[Instruction]	OR - OR
[Description]	The instruction ORs the words in reg1 and reg2 and stores the result into reg2. The contents of reg1 remain as is.
[Exception]	None

ORI

OR of Immediate data and register data

[Syntax] ORI imm16, reg1, reg2

[Operation] GR[reg2] ← GR[reg1] OR zero-extend(imm16)

[Format] Format V

[Operation code]

15 … 10	9 … 5	4 … 0	31 … 16
101100	reg2	reg1	imm16

[Flags]
CY : —
OV : 0
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] ORI - OR of immediate data and register (16 bits)

[Description] The instruction ORs the word in reg1 and the 16 bits of immediate data, zero-extended to a word, and stores the result into reg2. The contents of reg1 remain as is.

[Exception] None

OUT

Output to port

[Syntax]

(1) OUT.B reg2, disp16[reg1]
(2) OUT.H reg2, disp16[reg1]
(3) OUT.W reg2, disp16[reg1]

[Operation]

(1) adr ← GR[reg1] + (sign-extend)disp16
Output-Port(adr, GR[reg2], Byte)
(2) adr ← GR[reg1] + (sign-extend)disp16
Output-Port(adr, GR[reg2], Halfword)
(3) adr ← GR[reg1] + (sign-extend)disp16
Output-Port(adr, GR[reg2], Word)

[Format]

Format VI

[Operation code]

15 10	9 5	4 0	31 16
1111∗$	reg2	reg1	disp16

(∗$: 00 = (1), 01 = (2), 11 = (3))

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction]

(1) OUT.B - Output byte to port
(2) OUT.H - Output halfword to port
(3) OUT.W - Output word to port

[Description]

(1) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit port address. It outputs the low-order one byte of data in reg2 to the resulting port address.
(2) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit port address. It outputs the low-order two bytes of data in reg2 to the resulting port address. Bit 0 of the unsigned 32-bit address is masked to 0.
(3) The instruction adds the data in reg1 and the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit port address. It outputs the word in reg2 to the resulting port address. Bits 0 and 1 of the unsigned 32-bit address are masked to 0.

[Exception]

None

RETI — Return from Trap or Interrupt

[Syntax] RETI

[Operation]

```
if PSW.NP = 1
   then PC ← FEPC
        PSW ← FEPSW
   else PC ← EIPC
        PSW ← EIPSW
```

[Format] Format IX

[Operation code]

15 10	9 1	0
011001	RFU	0

[Flags]

CY	:	Will contain the read value.
OV	:	Will contain the read value.
S	:	Will contain the read value.
Z	:	Will contain the read value.

[Instruction] RETI - Return from trap or interrupt

[Description] The instruction takes the return PC and PSW out of the system registers to enable return from a trap or interrupt routine. Its operation is as follows:

(1) The instruction retrieves the return PC and PSW from FEPC and FEPSW if the PSW NP flag is set to 1, or from EIPC and EIPSW if the NP flag is set to 0.

(2) The instruction sets the retrieved return PC and PSW in the PC and PSW, causing a jump to the PC.

[Exception] None

SAR — Shift Arithmetic to the Right

[Syntax]
(1) SAR reg1, reg2
(2) SAR imm5, reg2

[Operation]
(1) GR[reg2] ← GR[reg2] arithmetically shift right by GR[reg1]
(2) GR[reg2] ← GR[reg2] arithmetically shift right by zero-extend(imm5)

[Format]
(1) Format I
(2) Format II

[Operation code]

(1)

15 10	9 5	4 0
000111	reg2	reg1

(2)

15 10	9 5	4 0
010111	reg2	imm5

[Flags]

CY	:	Assumes 1 if the last shift-out bit is 1. Otherwise, assumes 0. If the amount of the shift is 0, the CY flag is set to 0.
OV	:	0
S	:	Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z	:	Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction]
(1) SAR - Shift arithmetic right by amount specified by register
(2) SAR - Shift arithmetic right by amount specified by immediate data (5 bits)

[Description]
(1) The instruction arithmetically shifts the word in reg2 to the right (copies the MSB value at each position to the MSB in sequence) by the amount specified by the low-order five bits in reg1, then writes the result into reg2. If the amount is 0, the reg2 value is not changed by the shift. The amount may be 0 to +31, being represented by five bits.
(2) The instruction arithmetically shifts the word in reg2 to the right (copies the MSB value at each position to the MSB in sequence) by the amount specified by the five bits of immediate data, zero-extended to a word, and writes the result into reg2. If the amount is 0, the reg2 value is not changed by the shift. The amount may be 0 to +31.

[Exception]
None

SATADD3

Saturatable Addition on 3 operands

[Syntax] SATADD3 reg1, reg2, reg3

[Operation] GR[reg3] ← saturate(GR[reg2] + GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	010000	RFU	reg3

[Flags]

CY : Assumes 1 if there is a carry from the MSB. Otherwise, assumes 0.
OV : Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S : Assumes 1 if GR[reg3] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg3] is zero. Otherwise, assumes 0.

[Instruction] SATADD3 - Saturatable addition on 3 operands

[Description] The instruction adds together the words in reg1 and reg2 as signed integers.

[If no overflow has occurred:]

The sum is stored into reg3.

[If an overflow has occurred:]

The SAT flag is set to 1. If the sum is positive, the positive maximum (7FFFFFFFH) is stored into reg3; if the sum is negative, the negative maximum (80000000H) is stored into reg3.

The contents of reg1 and reg2 remain as is.

[Supplement] The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even if the result of a subsequent operation instruction is not saturated. To reset the SAT flag to 0, use the LDSR instruction to rewrite the PSW. If the result of an operation performed by this instruction is saturated, the flags do not indicate the magnitudes of the reg1 and reg2 values. This means that the ABGT, ABGE, ABLT, ABLE, BGT, BGE, BLT, and BLE instructions do not assure normal branching. Instead, therefore, use the ABE, ABNE, ABN, ABP, BE, BNE, BN, or BP instruction.

[Exception] None

SATSUB3

Saturatable Subtraction on 3 operands

[Syntax] SATSUB3 reg1, reg2, reg3

[Operation] GR[reg3] ← saturate(GR[reg2] - GR[reg1])

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	010001	RFU	reg3

[Flags]

CY :	Assumes 1 if there is a carry from the MSB. Otherwise, assumes 0.
OV :	Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S :	Assumes 1 if GR[reg3] is negative. Otherwise, assumes 0.
Z :	Assumes 1 if GR[reg3] is zero. Otherwise, assumes 0.

[Instruction] SATSUB3 - Saturatable subtraction on 3 operands

[Description] The instruction subtracts the word in reg1 from that in reg2 as signed integers.

[If no overflow has occurred:]

The difference is stored into reg3.

[If an overflow has occurred:]

The SAT flag is set to 1. If the difference is positive, the positive maximum (7FFFFFFFH) is stored into reg3; if the difference is negative, the negative maximum (80000000H) is stored into reg3.

The contents of reg1 and reg2 remain as is.

[Supplement] The SAT flag is cumulative, meaning that once the result of a saturatable operation instruction is saturated, the flag is set to 1 and is not reset to 0 even if the result of a subsequent operation instruction is not saturated. To reset the SAT flag to 0, use the LDSR instruction to rewrite the PSW. If the result of the operation performed by this instruction is saturated, the flags do not indicate the magnitudes of the reg1 and reg2 values. This means that the ABGT, ABGE, ABLT, ABLE, BGT, BGE, BLT, and BLE instructions do not assure normal branching. Instead, therefore, use the ABE, ABNE, ABN, ABP, BE, BNE, BN, or BP instruction.

[Exception] None

SETF

Set Flag condition

[Syntax] SETF imm5,reg2

[Operation]
```
if conditions are satisfied
  then GR[reg2] ← 00000001H
  else GR[reg2] ← 00000000H
```

[Format] Format II

[Operation code]

15 10	9 5	4 0
010010	reg2	imm5

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] SETF - Set flag condition

[Description] If the condition specified by the low-order four of the five bits of the immediate data is satisfied, the instruction writes 1 into reg2; otherwise, it writes 0 into reg2. The low-order four of the five bits of immediate data indicate one of the condition codes listed in Table 5-3. The high-order one bit is ignored.

[Exception] None

Table 5-3. Condition Codes

Condition code	Name	Conditional expression
0000	V	OV = 1
1000	NV	OV = 0
0001	C/L	CY = 1
1001	NC/NL	CY = 0
0010	Z	Z = 1
1010	NZ	Z = 0
0011	NH	(CY or Z) = 1
1011	H	(CY or Z) = 0
0100	S/N	S = 1
1100	NS/P	S = 0
0101	T	always 1
1101	F	always 0
0110	LT	(S xor OV) = 1
1110	GE	(S xor OV) = 0
0111	LE	((S xor OV) or Z) = 1
1111	GT	((S xor OV) or Z) = 0

SHL — Shift Logical to the Left

[Syntax]
(1) SHL reg1, reg2
(2) SHL imm5, reg2

[Operation]
(1) GR[reg2] ← GR[reg2] logically shift left by GR[reg1]
(2) GR[reg2] ← GR[reg2] logically shift left by zero-extend(imm5)

[Format]
(1) Format I
(2) Format II

[Operation code]

(1)

15 10	9 5	4 0
000100	reg2	reg1

(2)

15 10	9 5	4 0
010100	reg2	imm5

[Flags]
CY : Assumes 1 if the last shift-out bit is 1. Otherwise, assumes 0. If the amount of the shift is 0, the CY flag is 0.
OV : 0
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction]
(1) SHL - Shift logical left by amount specified by register
(2) SHL - Shift logical left by amount specified by immediate data (5 bits)

[Description]
(1) The instruction logically shifts the word in reg2 to the left (puts 0 on the LSB) by the amount specified by the low-order five bits in reg1, then writes the result into reg2. If the amount is 0, the reg2 value is not changed by the shift. The amount may be 0 to +31, being represented by five bits.
(2) The instruction logically shifts the word in reg2 to the left (puts 0 on the LSB) by the amount specified by the five bits of immediate data, zero-extended to a word, and writes the result into reg2. If the amount is 0, the reg2 value is not changed by the shift. The amount may be 0 to +31.

[Exception]
None

SHLD3 Shift to the Left of Double word on 3 operands

[Syntax] SHLD3 reg1, reg2, reg3

[Operation] GR[reg3] ← (GR[reg3], GR[reg2]) << reg1

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	011000	RFU	reg3

[Flags] CY : —
OV : —
S : —
Z : —

[Instruction] SHLD3 - Shift left double word

[Description] The instruction logically shifts the 64 bits of data obtained by concatenating reg3 (high order) and reg2 (low order) to the left by the amount specified by the low-order five bits in reg1, then outputs the high-order 32 bits of the result into reg3. If reg1 is 0, the reg3 data remains as is. The high-order 27 bits in reg1 are ignored. The contents of reg1 and reg2 remain as is.

[Supplement] A timing restriction is imposed on SHLD3 instruction input operand reg3. If an instruction to update reg3 is not issued within three cycles before the issue of the SHLD3 instruction, the SHLD3 instruction will begin after a one-cycle halt (stall).

[Exception] None

SHR

Shift Logical to the Right

[Syntax] (1) SHR reg1, reg2
(2) SHR imm5, reg2

[Operation] (1) GR[reg2] ← GR[reg2] logically shift right by GR[reg1]
(2) GR[reg2] ← GR[reg2] logically shift right by zero-extend(imm5)

[Format] (1) Format I
(2) Format II

[Operation code]

(1)

15 10	9 5	4 0
000101	reg2	reg1

(2)

15 10	9 5	4 0
010101	reg2	imm5

[Flags]

CY :	Assumes 1 if the last shift-out bit is 1. Otherwise, assumes 0. If the amount of the shift is 0, the CY flag is 0.
OV :	0
S :	Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z :	Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] (1) SHR - Shift logical right by amount specified by register
(2) SHR - Shift logical right by amount specified by immediate data (5 bits)

[Description] (1) The instruction logically shifts the word in reg2 to the right (puts 0 on the MSB) by the amount specified by the low-order five bits in reg1, then writes the result into reg2. If the amount is 0, the reg2 value is not changed by the shift. The amount may be 0 to +31, being represented by five bits.
(2) The instruction logically shifts the word in reg2 to the right (puts 0 on the MSB) by the amount specified by the five bits of immediate data, zero-extended to a word, and writes the result into reg2. If the amount is 0, the reg2 value is not changed by the shift. The amount may be 0 to +31.

[Exception] None

SHRD3 Shift to the Right of Double word on 3 operands

[Syntax] SHRD3 reg1, reg2, reg3

[Operation] GR[reg3] ← (GR[reg3], GR[reg2]) >> reg1

[Format] Format VIII

[Operation code]

15 10	9 5	4 0	31 26	25 21	20 16
111110	reg2	reg1	011001	RFU	reg3

[Flags]

CY :	—
OV :	—
S :	—
Z :	—

[Instruction] SHRD3 - Shift right double word

[Description] The instruction logically shifts the 64 bits of data obtained by concatenating reg3 (high order) and reg2 (low order) to the right by the amount specified by the low-order five bits in reg1, then outputs the low-order 32 bits of the result into reg3. If reg1 is 0, the reg2 data is stored into reg3. The high-order 27 bits in reg1 are ignored. The contents of reg1 and reg2 remain as is.

[Supplement] A timing restriction is imposed on SHRD3 instruction input operand reg3. If an instruction to update reg3 is not issued within three cycles before the issue of the SHRD3 instruction, the SHRD3 instruction will begin after a one-cycle halt (stall).

[Exception] None

ST — Store

[Syntax]

(1) ST.B reg2, disp16[reg1]
(2) ST.H reg2, disp16[reg1]
(3) ST.W reg2, disp16[reg1]

[Operation]

(1) adr ← GR[reg1] + (sign-extend)disp16
Store-Memory(adr, GR[reg2], Byte)
(2) adr ← GR[reg1] + (sign-extend)disp16
Store-Memory(adr, GR[reg2], Halfword)
(3) adr ← GR[reg1] + (sign-extend)disp16
Store-Memory(adr, GR[reg2], Word)

[Format]

Format VI

[Operation code]

15 10	9 5	4 0	31 16
1101∗$	reg2	reg1	disp16

(∗$: 00 = (1), 01 = (2), 11 = (3))

[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction]

(1) ST.B - Store byte
(2) ST.H - Store halfword
(3) ST.W - Store word

[Description]

(1) The instruction adds the data in reg1 to the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit address. It stores the low-order one byte of reg2 data at the resulting address.
(2) The instruction adds the data in reg1 to the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit address. It stores the low-order two bytes of reg2 data at the resulting address. Bit 0 of the unsigned 32-bit address is masked to 0.
(3) The instruction adds the data in reg1 to the 16-bit displacement, sign-extended to a word, to produce an unsigned 32-bit address. It stores the word from reg2 at the resulting address. Bits 0 and 1 of the unsigned 32-bit address are masked to 0.

[Exception]

None

STBY Standby

[Syntax]	STBY
[Operation]	Stop
[Format]	Format IX

[Operation code]

15 10	9 1	0
011010	RFU	1

[Flags]	CY : —
	OV : —
	S : —
	Z : —
[Instruction]	STBY - Standby
[Description]	The instruction stops the CPU and places the system in stop mode.
[Exception]	None

STSR Store contents of System Register

[Syntax] STSR regID, reg2

[Operation] GR[reg2] ← SR[regID]

[Format] Format II

[Operation code]

15 10	9 5	4 0
011010	reg2	regID

[Flags]
CY : —
OV : —
S : —
Z : —

[Instruction] STSR - Store contents of system register

[Description] The instruction writes the contents of the system register identified by the system register number (regID) into reg2. There is no influence on the system register. System register numbers uniquely identify system registers. If the STSR instruction is executed on a reserved system register, however, the operation of the instruction will be unpredictable.

[Exception] None

SUB

Subtract

[Syntax] SUB reg1, reg2

[Operation] GR[reg2] ← GR[reg2] - GR[reg1]

[Format] Format I

[Operation code]

15 10	9 5	4 0
000010	reg2	reg1

[Flags]

CY : Assumes 1 if there is a borrow from the MSB. Otherwise, assumes 0.
OV : Assumes 1 if overflow has occurred. Otherwise, assumes 0.
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] SUB - Subtract

[Description] The instruction subtracts the word in reg1 from that in reg2 and stores the difference into reg2. The contents of reg1 remain as is.

[Exception] None

TRAP

Software Trap

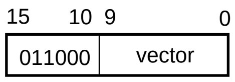
[Syntax] TRAP vector

[Operation]

```
if PSW.NP = 1
    then fatal exception (MACHINE FAULT)
else if PSW.EP = 1
    then  FEPC       ← return PC
          FEPSW      ← PSW
          ECR.FECC   ← exception code
          PSW.NP     ← 1
          PSW.ID     ← 1
          PC         ← <NMI handler address>
    else  EIPC       ← return PC
          EIPSW      ← PSW
          ECR.EICC   ← exception code
          PSW.EP     ← 1
          PSW.ID     ← 1
          PC         ← <vector adr>
```

[Format] Format II

[Operation code]



[Flags]

CY : —
OV : —
S : —
Z : —

[Instruction] TRAP - Trap

[Description] If the PSW NP flag is set to 1, it indicates a fatal exception. The processor performs fatal exception handling.
If the PSW NP flag is set to 0 and the EP flag to 1, it indicates a double exception. In this case, the instruction saves the return PC and PSW into FEPC and FEPSW and sets the exception code (FECC in the ECR) and the PSW flags (the NP and ID flags). Program execution then jumps to the NMI handler address to begin exception handling. There is no influence on the condition flags.
If both the PSW NP and EP flags are set to 0, the instruction saves the return PC and PSW into EIPC and EIPSW and sets the exception code (EICC in the ECR) and the PSW flags (the EP and ID flags). Program execution then jumps to the trap handler address corresponding to the trap vector (0-31) identified by vector to begin exception handling. There is no influence on the condition flags.

The return PC gives the address of the instruction subsequent to the TRAP instruction.

[Exception]	None

XOR

Exclusive OR

[Syntax]	XOR reg1, reg2
[Operation]	GR[reg2] ← GR[reg2] XOR GR[reg1]
[Format]	Format I

[Operation code]

15 10	9 5	4 0
001110	reg2	reg1

[Flags]

CY	:	—
OV	:	0
S	:	Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z	:	Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction]	XOR - Exclusive OR
[Description]	The instruction takes the exclusive OR of the words in reg1 and reg2 and stores the result into reg2. The contents of reg1 remain as is.
[Exception]	None

XORI

Exclusive OR of Immediate and register data

[Syntax] XORI imm16, reg1, reg2

[Operation] GR[reg2] ← GR[reg1] XOR zero-extend(imm16)

[Format] Format V

[Operation code]

15 10	9 5	4 0	31 16
101110	reg2	reg1	imm16

[Flags]

CY : —
OV : 0
S : Assumes 1 if GR[reg2] is negative. Otherwise, assumes 0.
Z : Assumes 1 if GR[reg2] is zero. Otherwise, assumes 0.

[Instruction] XORI - Exclusive OR of immediate data (16 bits) and register data

[Description] The instruction takes the exclusive OR of the word in reg1 and the 16 bits of immediate data, zero-extended to a word, and stores the result into reg2. The contents of reg1 remain as is.

[Exception] None

5.4 INSTRUCTION EXECUTION CYCLES

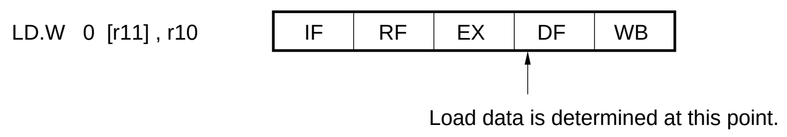
This section lists the execution cycles for each instruction. The number of actual execution cycles will fall between the repeat and the latency.

(1) Latency

The latency is defined as the period between an instruction beginning to run and its ending.

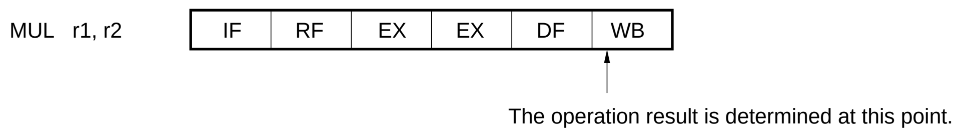
[Example 1] LD instruction

The latency of the LD instruction is 2. This instruction is executed in two cycles for the EX and DF stages.



[Example 2] MUL instruction

The latency of the MUL instruction is 4. This instruction is executed in four cycles for the EX, DF, and WB stages.



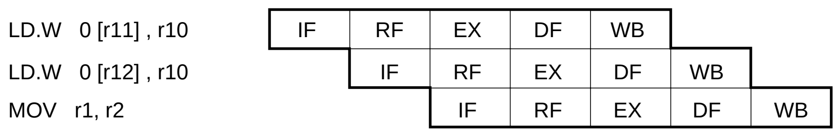
Remark For details of the pipeline flow, see **Chapter 9**.

(2) Repeat

The repeat is defined as the period between the current instruction beginning to run and the subsequent instruction becoming ready to run when the current and subsequent instructions use the same arithmetic/logic unit. Instructions begin to run as soon as they receive their required operands.

[Example 1] LD instruction

The repeat of the LD instruction is 1. This instruction uses the EX stage in one cycle. Therefore, the subsequent instruction can use the EX stage one cycle after it is used by the LD instruction.



[Example 2] **MUL instruction**

The repeat of the MUL instruction is 2. This instruction uses the EX stage in two cycles. Therefore, the subsequent instruction can use the EX stage two cycles after it is used by the MUL instruction. That is, the subsequent instruction cannot always use the EX stage without a wait.

MUL r1, r2	IF	RF	EX	EX	DF	WB				
MUL r3, r4		IF	–	RF	EX	EX	DF	WB		
MOV r5, r6			IF	–	–	RF	EX	EX	DF	WB

Remark For details of the pipeline flow, see **Chapter 9**.

(3) Latency - repeat (difference between the latency and repeat)

The difference between the latency and repeat indicates the pipelined stage in which the operation result is output.

When the difference is 0: The operation result is output in the EX stage.
When the difference is 1: The operation result is output in the DF stage.
When the difference is 2: The operation result is output in the WB stage.

The meanings of the abbreviations and other quantities used in Table 5-4 are as follows:

<1>:	Data cache hit or internal RAM access
<2>:	Data cache miss
<3>:	External RAM (uncachable area) access
B:	Number of clock cycles for burst bus cycle execution (external clock)
S:	Number of clock cycles for single bus cycle execution (external clock)
n:	Frequency ratio between internal and external clocks (n = 2 or n = 3)

s:	Wait time for synchronization with external clock
	s = 0 or 1 if n = 2.
	s = 0, 1, or 2 if n = 3

Table 5-4. Instruction Execution Cycles (1/3)

*

	Mnemonic	Operand	Instruction length in bytes	Latency	Repeat
Load/store	LD.B	disp16[reg1], reg2	4	**<1>** 2	**<1>** 1
	LD.H	disp16[reg1], reg2	4	**<2>** n x B + 10 + s[Note 1]	**<2>** n x B + 9 + s
	LD.W	disp16[reg1], reg2	4	**<3>** n x S + 9 + s[Note 1]	**<3>** n x S + 8 + s
	ST.B	reg2, disp16[reg1]	4	**<1>** 3	1
	ST.H	reg2, disp16[reg1]	4	**<2>** n x S + 5 + s[Note 1]	
	ST.W	reg2, disp16[reg1]	4	**<3>** n x S + 5 + s[Note 1]	
	BILD	[reg1], [reg2]	4	n x B + 10 + s[Note 1]	n x B + 10 + s
	BIST	[reg2], [reg1]	4	n x B + 7 + s[Note 1]	n x (B – 1) + 10 + s
	BDLD	[reg1], [reg2]	4	n x B + 10 + s[Note 1]	n x B + 10 + s
	BDST	[reg2], [reg1]	4	n x B + 7 + s[Note 1]	n x (B – 1) + 10 + s
Input/output	IN.B	disp16[reg1], reg2	4	n x S + 10 + s[Note 1]	n x S + 9 + s
	IN.H	disp16[reg1], reg2	4		
	IN.W	disp16[reg1], reg2	4		
	OUT.B	reg2, disp16[reg1]	4	n x S + 6 + s[Note 1]	n x S + 9 + s
	OUT.H	reg2, disp16[reg1]	4		
	OUT.W	reg2, disp16[reg1]	4		
Arithmetic operation	MOV	reg1, reg2	2	1	1
		imm5, reg2			
	MOVHI	imm16, reg1, reg2	4	1	1
	ADD	reg1, reg2	2	1	1
		imm5, reg2			
	ADDI	imm16, reg1, reg2	4	1	1
	MOVEA	imm16, reg1, reg2	4	1	1
	SUB	reg1, reg2	2	1	1
	MUL	reg1, reg2	2	4[Note 2]	2
	MULU	reg1, reg2	2	4[Note 2]	2
	DIV	reg1, reg2	2	37	37
	DIVU	reg1, reg2	2	35	35
	CMP	reg1, reg2	2	1	1
		imm5, reg2			
	SETF	imm5, reg2	2	2	1
	MIN3	reg1, reg2, reg3	4	2	1
	MAX3	reg1, reg2, reg3	4	2	1

Notes 1. A write bus cycle may be added because the write buffer is emptied for execution.

2. The flag requires three latency cycles. If the next instruction references the flag (as in the case of a conditional branch instruction), a flag hazard will result.

Table 5-4. Instruction Execution Cycles (2/3)

	Mnemonic	Operand	Instruction length in bytes	Latency	Repeat
Sum-of-products/ saturatable operation	MUL3	reg1, reg2, reg3	4	3	1
	MAC3	reg1, reg2, reg3	4	3 Note 1	1 Note 1
	MULI	imm16, reg1, reg2	4	3	1
	MACI	imm16, reg1, reg2	4	3	1
	MULT3	reg1, reg2, reg3	4	3	1
	MACT3	reg1, reg2, reg3	4	3 Note 1	1 Note 1
	SATADD3	reg1, reg2, reg3	4	2	1
	SATSUB3	reg1, reg2, reg3	4	2	1
Logical operation	OR	reg1, reg2	2	1 Note 2	1
	ORI	imm16, reg1, reg2	4	1 Note 2	1
	AND	reg1, reg2	2	1 Note 2	1
	ANDI	imm16, reg1, reg2	4	1 Note 2	1
	XOR	reg1, reg2	2	1 Note 2	1
	XORI	imm16, reg1, reg2	4	1 Note 2	1
	NOT	reg1, reg2	2	1 Note 2	1
	SHL	reg1, reg2	2	2 Note 2	1
		imm5, reg2			
	SHR	reg1, reg2	2	2 Note 2	1
		imm5, reg2			
	SAR	reg1, reg2	2	2 Note 2	1
		imm5, reg2			
	SHLD3	reg1, reg2, reg3	4	2 Note 1	1 Note 1
	SHRD3	reg1, reg2, reg3	4	2 Note 1	1 Note 1

Notes 1. A one-cycle halt occurs unless an instruction which acts on reg3 as its destination is executed up to three cycles before the issue of this instruction.

2. The flag requires two latency cycles. If the next instruction references the flag (as in the case of a conditional branch instruction), a flag hazard will result.

Table 5-4. Instruction Execution Cycles (3/3)

	Mnemonic	Operand	Instruction length in bytes	Latency	Repeat
Branch	JMP	[reg1]	2	3Note 1	3
	JR	disp26	4	3Note 1	3
	JAL	disp26	4	3Note 1	3
	Bcond	disp9	2	3 (taken)Note 1 1 (not taken)Note 2	3 (taken) 1 (not taken)
	ABcond	disp9	2	1 (History available)Note 1 3 (History unavailable)	1 (History available) 3 (History unavailable)
Special	LDSR	reg2, regID	2	5	5
	STSR	regID, reg2	2	5	2
	TRAP	vector	2	5	5
	RETI	—	2	5Note 1	5
	CAXI	disp16[reg1], reg2	4	n x S + 18 + sNote 3	n x S + 18 + s
	HALT	—	2	5Note 3	—
	STBY	—	2	5Note 3	—
	BRKRET	—	2	5Note 1	5
	EI	—	2	4	4
	DI	—	2	4	4

Notes 1. If the branch address is not a multiple of 4 and a 32-bit instruction exists at the branch address, a one-cycle halt occurs.

2. If the instruction next to the high-speed (advanced) branch (ABcond) instruction is 32 bits long and its address is not a multiple of 4, a one-cycle halt will occur when program execution exits from the loop.

3. Since the execution is preceded by the emptying of the write buffer, a write bus cycle could be added.

[MEMO]

CHAPTER 6 INTERRUPTS AND EXCEPTIONS

Interrupts are events occur independently of program execution. They are classified into maskable and nonmaskable interrupts. In contrast, exceptions are events which are directly related to program execution. Interrupts and exceptions do not differ greatly in their control flow, but interrupts are assigned higher handling priorities than exceptions. Fatal exceptions, however, are assigned higher priorities than interrupts.

Under the V830 Family architecture, the following interrupts and exceptions may occur. When an exception, maskable interrupt, or nonmaskable interrupt occurs, control is passed to a handler at an address which is predetermined a given cause. The cause of an exception can be identified by means of the exception code stored in the ECR (Exception Cause Register). The pertinent handler analyzes the contents of the ECR so that it can handle the exception or interrupt appropriately.

Table 6-1. Exception/Interrupt Source Codes

Exception/interrupt		Category	Exception code ECR[Note 1]	Interrupt request name	Handler address[Note1]	Return PC
Reset		Interrupt	FFF0H	RESET	FFFFFFF0H	Indefinite
Fatal exception		Exception	—	FAULT	FFFFFFE0H	Current PC
NMI		Interrupt	FFD0H	NMI	FFFFFFD0H	Next PC
Double exception		Exception	**Note 2**	NMI	FFFFFFD0H	Current PC
TRAP instruction (parameter 0x1n)		Exception	FFBnH	TRAP1n	FFFFFFB0H	Next PC
TRAP instruction (parameter 0x0n)		Exception	FFAnH	TRAP0n	FFFFFFA0H	Next PC
Invalid operation code		Exception	FF90H	I_OPC	FFFFFF90H	Current PC
Division by zero		Exception	FF80H	DIV0	FFFFFF80H	Current PC
Interrupt level n (n = 0-15)[Note 3]	HWCC.IHA = 0	Interrupt	FEn0H	INT0n	FFFFFEn0H	Next PC
	HWCC.IHA = 1			INT1n	FE0000n0H	

Notes 1. Level n is represented by a hexadecimal number (n = 0-F).

2. Exception code of the exception which caused the double exception

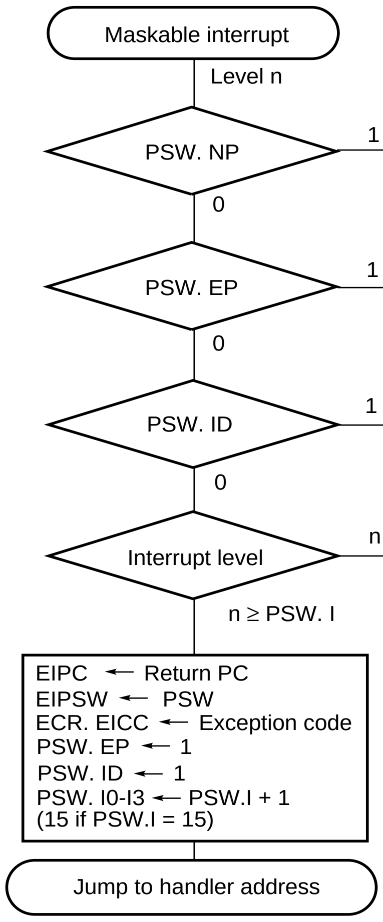
* **3.** V831 and V832 contain an interrupt controller. They allocate internal and external interrupt sources to INT0n and INT1n. Refer to **Chapter 4** in the **User's Manual - Hardware** of each product for more information.

6.1 INTERRUPT HANDLING

6.1.1 Maskable Interrupts

When a maskable interrupt occurs, the processor performs the following processing and passes control to the handler routine. It uses EIPC and EIPSW as status save registers.

Maskable interrupts are masked according to the OR of the NP, EP, and ID bits of the PSW. In addition, if interrupt level n indicated by $\overline{\text{INTV0}}$-$\overline{\text{INTV3}}$ is lower than the PSW-permitted interrupt level indicated by PSW bits I0-I3 (n < I0-I3), the interrupt is not accepted. It is therefore impossible to inhibit interrupts at the highest level (n = 15) by assigning a permitted interrupt level.



<1> Save the return PC in EIPC.

<2> Save the current PSW in EIPSW.

<3> Write the exception code into the low-order 16 bits (EICC) of the ECR.

<4> Set the PSW EP bit.

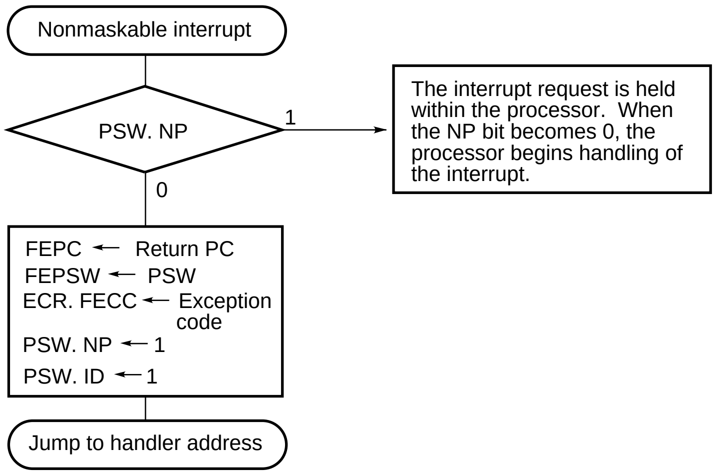
<5> Set the PSW ID bit.

<6> Set the accepted interrupt level n plus 1 (n + 1) in the PSW I (I0-I3) field. If the accepted interrupt level is the highest (n = 15), 15 is set.

<7> Jump to the handler address.

6.1.2 Nonmaskable Interrupts

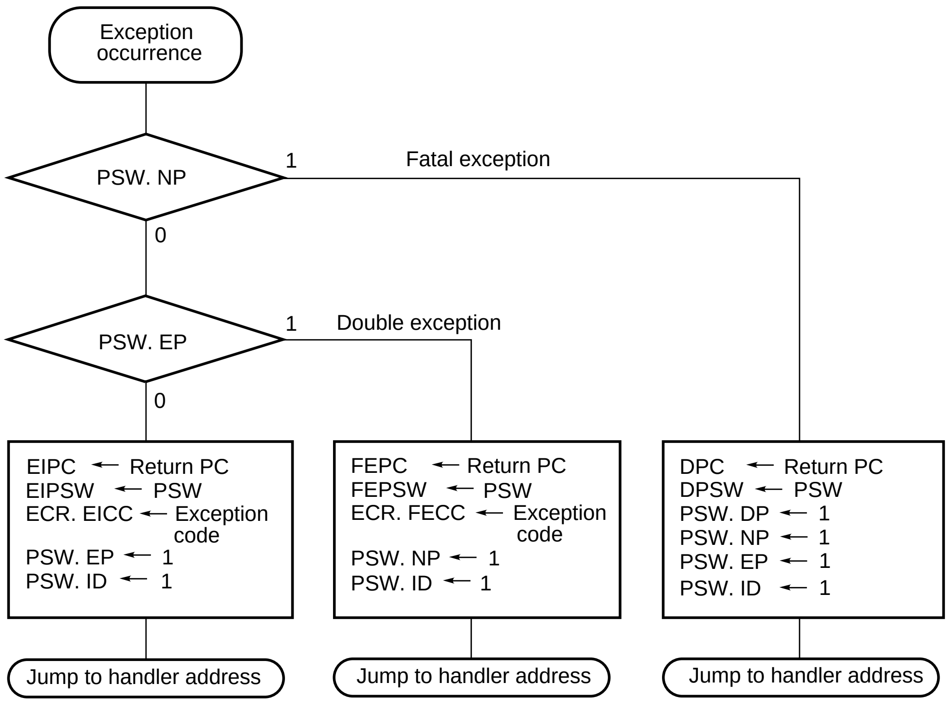
If a nonmaskable interrupt caused by the $\overline{\text{NMI}}$ input occurs, the processor performs the following processing and passes control to the handler routine. It uses FEPC and FEPSW as status save registers. If a nonmaskable interrupt request is issued while a nonmaskable interrupt is being handled (the PSW NP bit is 1), the request is held within the processor (if a nonmaskable interrupt request is issued during the period of internal processing for clearing the latch immediately after the beginning of nonmaskable interrupt handling, the request is not held with the latch within the processor). The processor detects a nonmaskable interrupt at the falling edge of the $\overline{\text{NMI}}$ input. Therefore, when issuing a nonmaskable interrupt request, deactivate then reactivate the $\overline{\text{NMI}}$ input.


Nonmaskable interrupt
PSW. NP
1
The interrupt request is held within the processor. When the NP bit becomes 0, the processor begins handling of the interrupt.
0
FEPC ← Return PC
FEPSW ← PSW
ECR. FECC ← Exception code
PSW. NP ← 1
PSW. ID ← 1
Jump to handler address

<1> Save the return PC in FEPC.
<2> Save the current PSW in FEPSW.
<3> Write the exception code into the high-order 16 bits (FECC) of the ECR.
<4> Set the PSW NP bit.
<5> Set the PSW ID bit.
<6> Jump to the handler address (FFFFFFD0H).

6.2 EXCEPTION HANDLING

When an exception occurs, the processor performs the following processing and passes control to the handler routine.

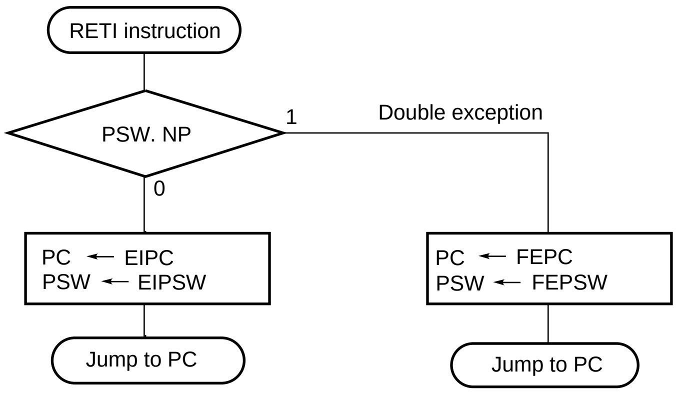


<1> If the PSW NP bit has already been set, go to **<8>**.
<2> If the PSW EP bit has already been set, go to **<9>**.
<3> Save the return PC in EIPC.
<4> Save the current PSW in EIPSW.
<5> Write the exception code into the low-order 16 bits (EICC) of the ECR.
<6> Set the EP and ID bits of the PSW.
<7> Jump to the handler address.
<8> Fatal exception handling
(a) Save the return PC in DPC.
(b) Save the current PSW in DPSW.
(c) Set the DP, NP, EP, and ID bits of the PSW.
(d) Jump to the handler address (FFFFFFE0H).
<9> Double exception handling
(a) Save the return PC in FEPC.
(b) Save the current PSW in FEPSW.
(c) Write the exception code into the high-order 16 bits (FECC) of the ECR.
(d) Set the NP and ID bits of the PSW.
(e) Jump to the handler address (FFFFFFD0H).

6.3 RETURN FROM EXCEPTION/INTERRUPT

6.3.1 Return from Exception/Interrupt

The RETI instruction is used for return from any exception and interrupt events other than fatal exceptions.

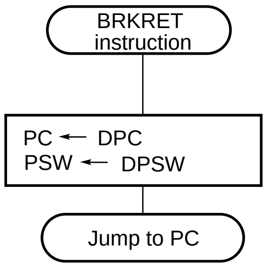


<1> Read the return PC and PSW from FEPC and FEPSW when the PSW NP bit is 1 or from EIPC and EIPSW when the PSW NP bit is 0.

<2> Restore the return PC and PSW and jump to the PC.

6.3.2 Return from Fatal Exception Handling Routine

The BRKRET instruction is used for return from fatal exception handling.



<1> Read the return PC and PSW from DPC and DPSW.

<2> Restore the return PC and PSW and jump to the PC.

6.4 PRIORITIES OF INTERRUPTS AND EXCEPTIONS

The priorities assigned to interrupts and exceptions are given below. If multiple interrupts and/or exceptions occur at the same time, they are handled according to their priorities.

	RESET	NMI	INT	Trap instruction	Invalid operation code exception	Division by zero exception
RESET		*	*	*	*	*
NMI	x		←	←	←	←
INT	x	↑		←	←	←
Trap instruction	x	↑	↑		—	—
Invalid operation code exception	x	↑	↑	—		—
Division by zero exception	x	↑	↑	—	—	

* :	The event on the left overrides that at the top.
x :	The event on the left is overridden by that at the top.
— :	The events on the left and at the top do not occur at that time.
← :	The event on the left is assigned a higher priority than that at the top.
↑ :	The event at the top is assigned a higher priority than that on the left.

*

6.4.1 Priorities of Maskable Interrupts

V831 and V832 incorporate an interrupt controller to control multiple interrupt sources, according to their priorities. Refer to **Chapter 4** in the **User's Manual - Hardware** of each product for more information.

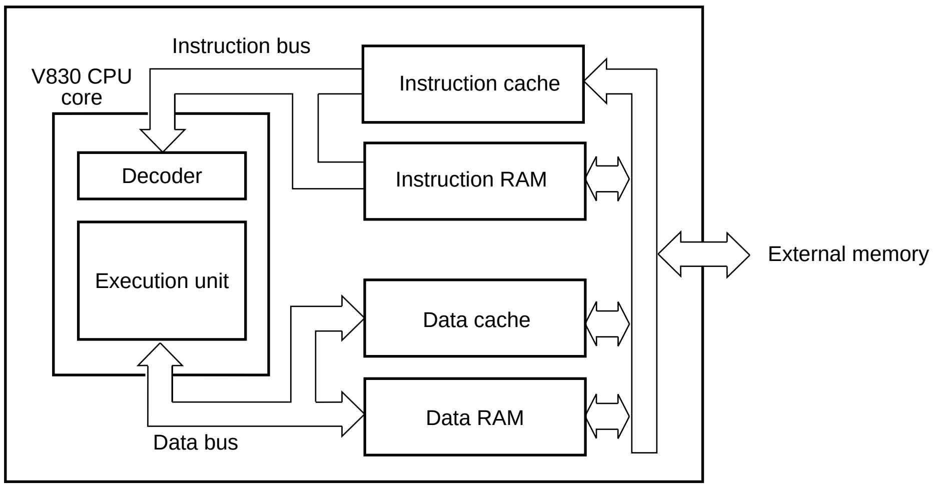
CHAPTER 7 INTERNAL MEMORY

This chapter describes the functions of the built-in cache memory and RAM devices, as well as their retrieval function.

7.1 BUILT-IN CACHE

The V830 Family has a 4K-byte x 4 internal memory, consisting of four blocks (instruction cache, data cache, instruction RAM, and data RAM). The V830 Family allows any of these internal memory blocks to be accessed in one cycle.

Figure 7-1. Built-In Cache Configuration



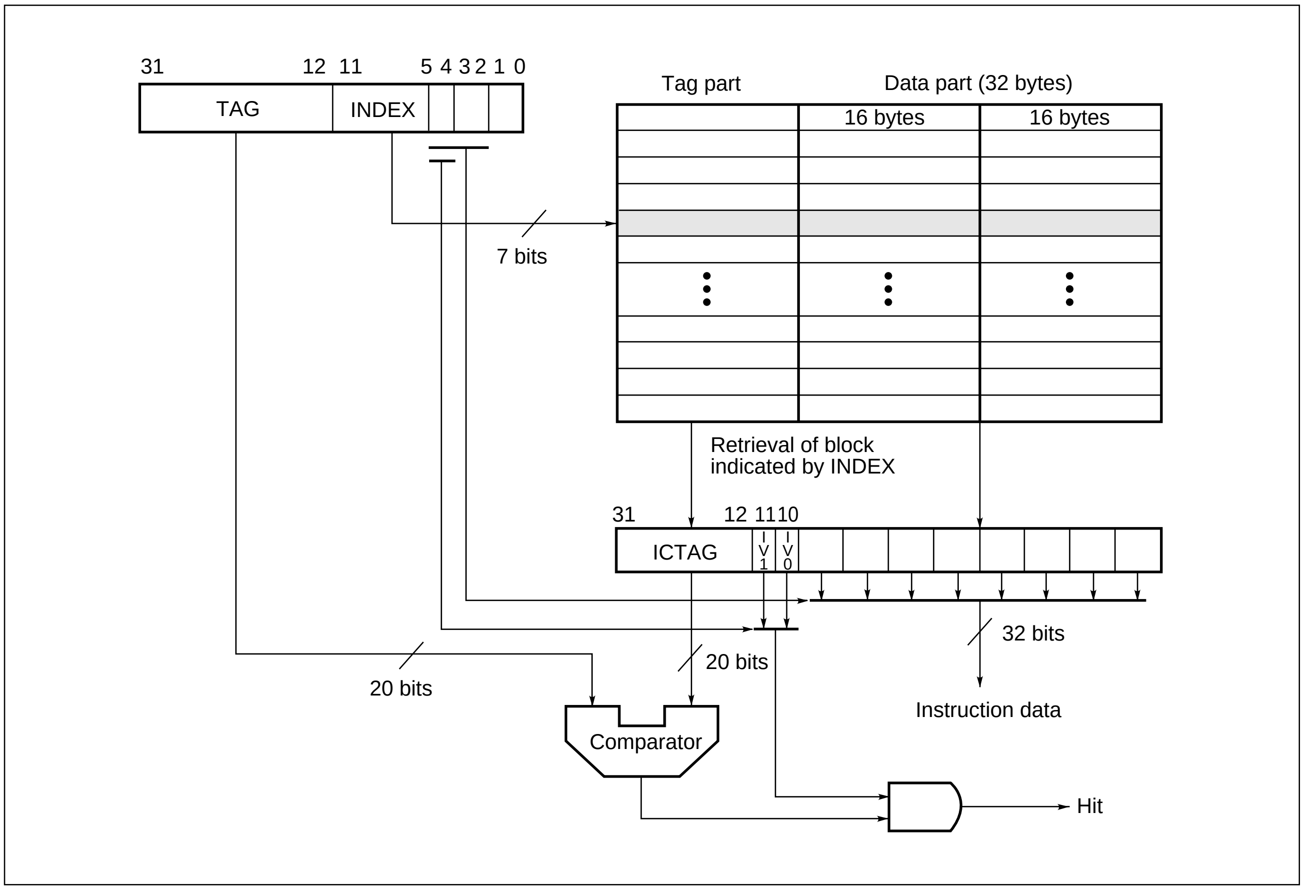
**Caution Data cannot be written into the instruction cache or instruction RAM.
A instruction cannot be written into the data cache or data RAM.**

7.1.1 Instruction Cache

The instruction cache memory consists of 128 32-byte blocks, having a total capacity of 4K bytes. Each block consists of two sub-blocks (16-byte), and has a tag and two valid bits, namely, IV1 (for the high-order 16 bytes of each 32-byte block) and IV0 (for the low-order 16 bytes). These valid bits indicate whether the contents of each sub-block are valid or invalid. If a cache error occurs, the memory is refilled in units of sub-blocks.

Those instructions that can be cached in the instruction cache are limited to an instruction string fetched from a cachable area. No instructions in the built-in instruction RAM are cached, however.

Figure 7-2. Instruction Cache Configuration



7.1.2 Instruction Cache Tag Retrieval

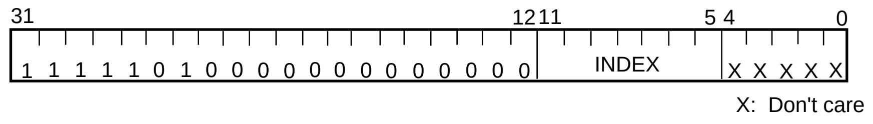
The V830 Family can retrieve the tags of those instructions cached in the instruction cache. The V830 Family recognizes an instruction string that has been cached by generating the addresses of the cached instructions from the tags.

The ICTR registers are used for tag retrieval. There are 128 ICTR registers. These ICTR registers are mapped in the I/O space (FA000000H-FA000FFFH). Numbers ICTR0 to ICTR127 are assigned to the registers, each of which is mapped to an address where bits 4 to 0 are 0s. These numbers also correspond to the block numbers of the cache.

(1) Instruction cache tag register

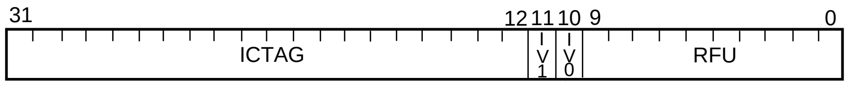
The instruction cache tag registers are used to retrieve the tags of the instructions cached in the instruction cache. To access these registers, use the IN.W or OUT.W instruction.

ICTR addressing method (FA000XXXH)



Bit position	Field name	Meaning
11-5	INDEX	Index Specifies the address of a built-in cache tag.

ICTR contents



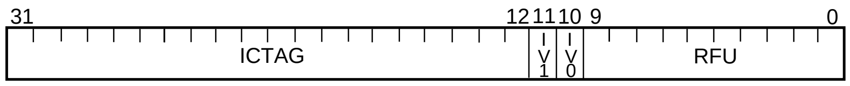
Bit position	Field name	Meaning
31-12	ICTAG	Instruction Cache Tag Tag of a block specified by the index of the instruction cache.
11	IV1	Instruction Cache Valid Bit Indicates that the high-order sub-block specified by the index is valid. IV1 = 0: Invalid IV1 = 1: Valid (The sub-block matches the contents of the external memory specified by ICTAG.)
10	IV0	Instruction Cache Valid Bit Indicates that the low-order sub-block specified by the index is valid. IV0 = 0: Invalid IV0 = 1: Valid (The sub-block matches the contents of the external memory specified by ICTAG.)
9-0	RFU	Reserved field (must be fixed to 0)

(2) Reading cache tags

The V830 Family reads a register, ICTRn, for an instruction cache block to be retrieved. Bits 31 to 12 of the data thus read indicate a tag, while bits 11 and 10 correspond to the valid bits of the related sub-blocks.

To read register ICTRn, use the IN.W instruction.

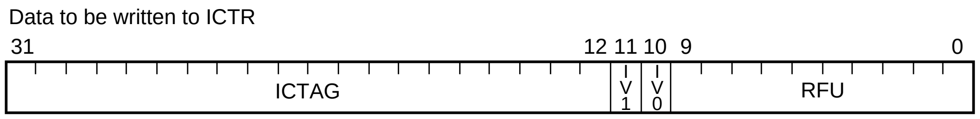
Data read from ICTR



(3) Writing cache tags

The V830 Family writes data, with a specified cache tag and valid bits, to ICTRn for the instruction cache block to be retrieved. This operation enables modification of the cache tag. The branch history (with instruction ABcond) of the written block is then erased.

To write in a cache tag, use OUT.W instruction.

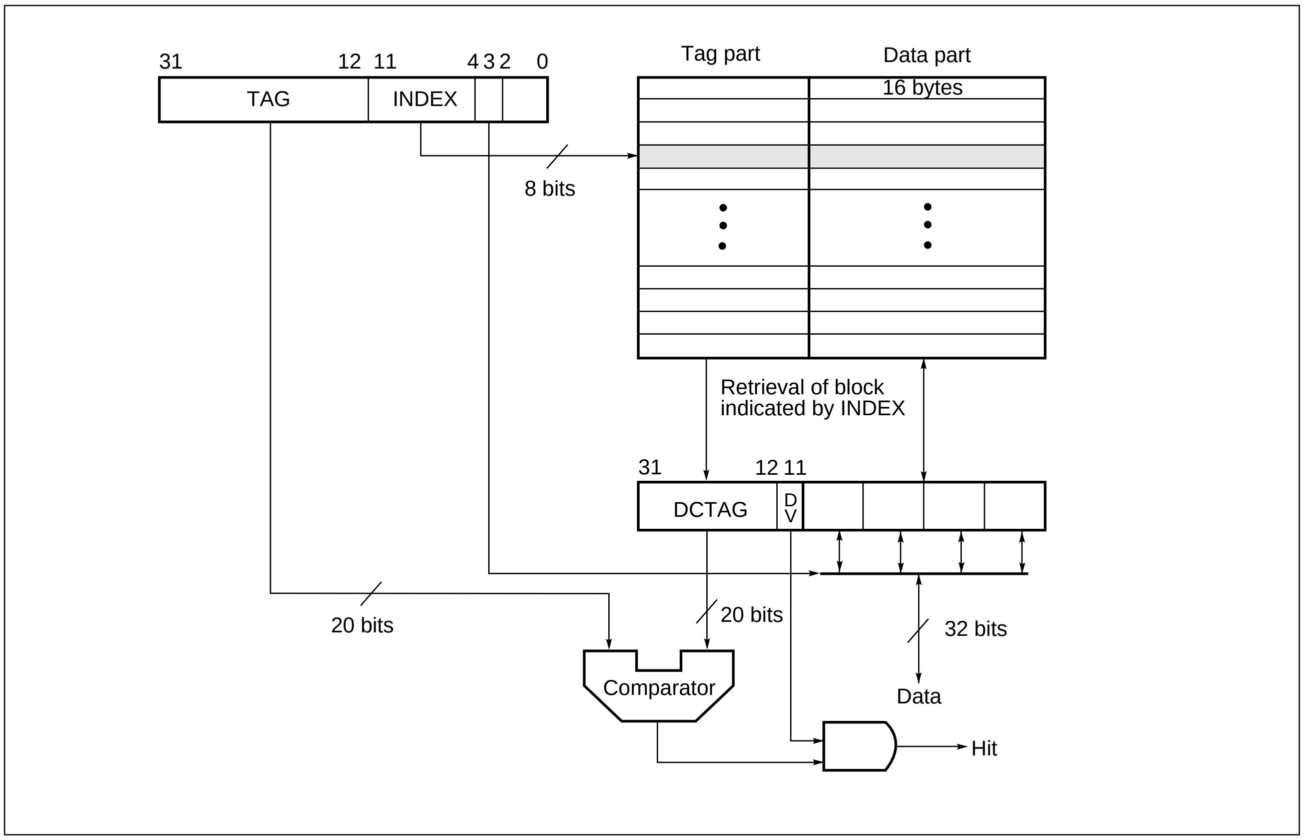


7.1.3 Data Cache

The data cache memory consists of 256 16-byte blocks, having a capacity of 4K bytes. Each block has a tag and valid bits. The valid bits indicate whether the contents of each block are valid or invalid. If a cache error occurs, the memory is refilled in units of blocks. The memory is refilled only when the V830 Family makes a cache error while reading data (write-through mode). Memory is not refilled when writing data.

Also, the data to be cached in the data cache is limited to that data in a cachable area. Data in data RAM or uncachable area is not cached.

Figure 7-3. Data Cache Configuration



7.1.4 Data Cache Tag Retrieval

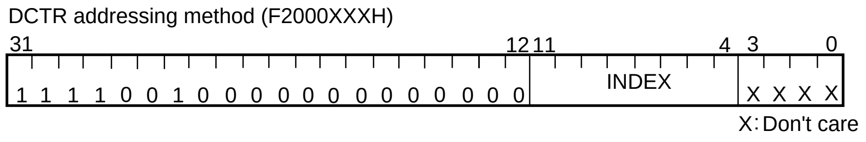
The V830 Family can retrieve the tags of data cached in the data cache. The V830 Family generates the addresses of the cached data from these tags to locate the cached data.

The DCTR registers are used for tag retrieval. There are 256 DCTR registers, which are mapped to the I/O space (F2000000H-F2000FFFH). Numbers DCTR0 to DCTR255 are assigned to these registers, which are each mapped to an address where bits 3 to 0 are 0s. These numbers also correspond to the block numbers of the cache.

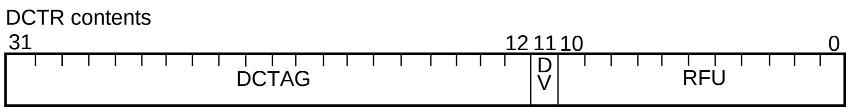
(1) Data cache tag registers

These registers are used for data cache tag retrieval.

To retrieve tags, use the IN.W or OUT.W instruction.



Bit position	Field name	Meaning
11-4	INDEX	Index Specifies the address of a built-in data cache tag.



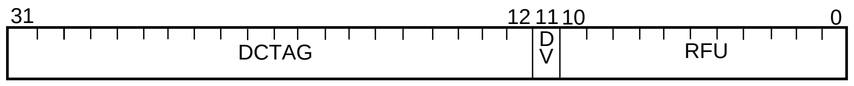
Bit position	Field name	Meaning
31-12	DCTAG	Data Cache Tag Tag of a block specified by the index of the data cache.
11	DV	Data Cache Valid Bit Indicates that the block specified by the index is valid. DV = 0: Invalid DV = 1: Valid (The block matches the contents of the external memory specified by DCTAG.)
10-0	RFU	Reserved field (must be fixed to 0)

(2) Reading cache tags

The V830 Family reads the register, DCTRn, for the data cache block to be retrieved. Bits 31 to 12 of the read data indicate the tag, while bit 11 corresponds to the valid bit.

To read DCTRn, use the IN.W instruction.

Data read from DCTR

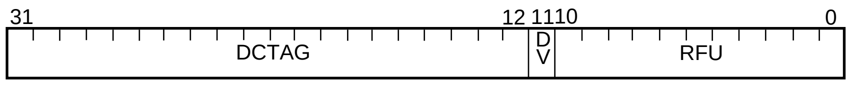


(3) Writing cache tags

The V830 Family writes data with a specified cache tag and valid bits to the register, DCTRn, for the data cache block to be retrieved. This operation enables modification of the cache tag.

To write data to DCTRn, use the OUT.W instruction.

Data to be written to DCTR

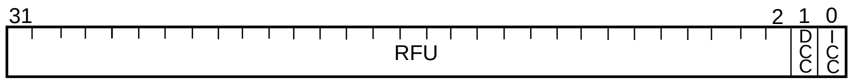


7.1.5 Cache Memory Control Register

The cache memory control register is used for cache clear control. This is a write-only register. If an attempt is made to read from this register, 0 will be read.

To access this register, use the OUT.W instruction.

CMCR (FFFFFFF4H)



Bit position	Field name	Meaning
31-2	RFU	Reserved field (must be fixed to 0)
1	DCC	Data Cache Clear If this bit is set to 1, the data cache is cleared. After the data is transferred to external memory by the external bus master (DMA), clear the data cache before accessing the data. When the DMA destination is the uncachable area, the data cache need not be cleared.
0	ICC	Instruction Cache Clear If this bit is set to 1, the instruction cache is cleared. After the program is transferred to external memory, clear the instruction cache before executing the program. Clear the branch history, too.

7.2 BUILT-IN RAM

7.2.1 Instruction RAM

The built-in instruction RAM is allocated to addresses FE000000H to FE000FFFH in the memory space. An instruction can be fetched from this space in one cycle. If an instruction string is stored into the built-in instruction RAM, instruction fetching can be effected without accessing external memory. The BILD and BIST instructions are used to transfer instructions between external memory and built-in instruction RAM.

Also, the built-in instruction RAM cannot be accessed with instructions LD or ST. If instructions LD or ST are used to access the RAM, the operation cannot be guaranteed.

* 7.2.2 Instruction RAM Retrieval (V830 and V831)

V830 and V831 support a function for accessing instructions stored in instruction RAM. This function enables instructions to be read from or written to RAM, albeit at low speed.

Instruction RAM can be referenced from addresses FE000000H to FE000FFFH of the I/O space. If, however, an instruction is written into this space, the branch history of the written part is erased.

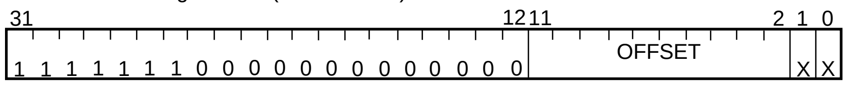
Caution V832 cannot use the instruction RAM retrieval function to read from or write to the instruction RAM by IN.W and OUT.W instructions. Use the BILD and BIST instructions to access the instruction RAM in four-word units by software for other than instruction fetches.

(1) Instruction RAM registers

The instruction RAM registers are used to read the contents of instruction RAM.

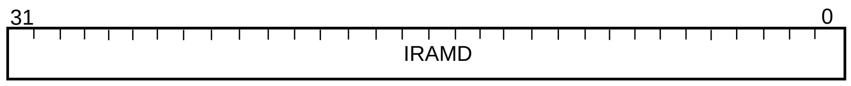
To access these registers, use the IN.W or OUT.W instruction.

IRAMR addressing method (FE000XXXH)



Bit position	Field name	Meaning
11-2	OFFSET	Offset Specifies an address in built-in instruction RAM.

IRAMR contents



Bit position	Field name	Meaning
31-0	IRAMD	Instruction RAM Data The Contents of built-in instruction RAM

(2) Instruction RAM retrieval

Using an IN.W instruction, read the desired instruction from the corresponding address in instruction RAM.

(3) Writing to instruction RAM

Using an OUT.W instruction, write the desired instruction to the corresponding address in instruction RAM.

7.2.3 Data RAM

The built-in data RAM is allocated to addresses 00000000H to 00000FFFH. Data loading/storing can be effected from/to this space in one cycle. Data transfer between external memory and built-in RAM can be performed at high speed by using a BDLD or BDST instruction.

Also, an instruction cannot be fetched from internal data RAM. If an instruction is fetched from internal data RAM, operation cannot be guaranteed.

Caution External memory at the addresses assigned to the built-in RAM area cannot be used.

CHAPTER 8 RESET

The system is reset when the $\overline{\text{RESET}}$ input goes low. The on-chip hardware is initialized.

8.1 INITIALIZATION

When the $\overline{\text{RESET}}$ input goes low, the system is reset to cause the system registers and internal registers to assume the conditions listed in Table 8-1.

When the $\overline{\text{RESET}}$ input goes high, the system is released from the reset state and starts program execution. The registers must be set appropriately by software.

Table 8-1. Conditions of Registers after Reset

	Register	Abbreviation	Condition after reset
System registers	Program counter	PC	FFFFFFF0H
	Exception/interrupt status save registers	EIPC	Unpredictable
		EIPSW	Unpredictable
	NMI/double exception status save registers	FEPC	Unpredictable
		FEPSW	Unpredictable
	Exception cause register	ECR	0000FFF0H
	Program status word	PSW	00008000H
	Processor ID register	PIR	00008300H
	Task control word	TKCW	000000E0H
	Debug exception status save register	DPC	Unpredictable
		DPSW	Unpredictable
	Hardware configuration control word	HCCW	00000000H
Internal registers	PLL control register**Note**	PLLCR	0000000XH
	Cache memory control register	CMCR	00000000H
	Instruction cache tag register	ICTR	XXXXX000H
	Data cache tag register	DCTR	XXXXX000H
	Instruction RAM register	IRAMR	Unpredictable

Note The condition after reset varies depending on CMODE.

8.2 START-UP

When the V830 Family is reset, it starts program execution at FFFFFFF0H. Immediately after a reset, the processor cannot accept interrupt requests. Before an interrupt can be used, the NP bit of the program status word (PSW) must be set to 0.

Caution For the V830, no instructions must be located at FFFFFFFBH and after.

CHAPTER 9 PIPELINE

The V830 Family, the design of which is based on the RISC architecture, executes most instructions within one clock, by means of 5-stage pipeline control.

The processor has a 5-stage pipeline structure.

The pipelined stages are listed below:

IF (instruction fetch)	:	Fetches an instruction and increments the fetch pointer.
RF (register fetch)	:	Decodes an instruction, creates immediate data, and reads registers.
EX (execute)	:	Executes a decoded instruction.
DF (data fetch)	:	Generates operation flags and read memory (cache).
WB (write back)	:	Writes the execution result into the register files and memory (cache).

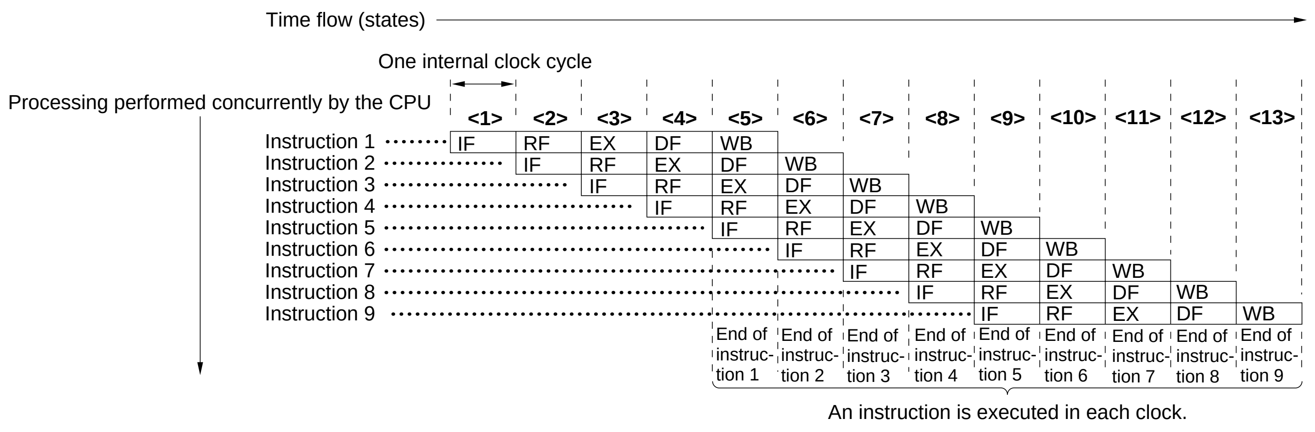
9.1 OUTLINE OF OPERATION

The instruction execution procedure of the V830 Family consists of five stages from fetch to write back.

The execution time of each stage differs according to the type of instruction and the type of memory to be accessed.

As an example of pipeline operation, Figure 9-1 illustrates the CPU processing that is performed when nine standard instructions are successively executed.

Figure 9-1. Example of Executing Nine Standard Instructions Successively



Time flow (states)
One internal clock cycle
Processing performed concurrently by the CPU
<1> <2> <3> <4> <5> <6> <7> <8> <9> <10> <11> <12> <13>
Instruction 1 IF RF EX DF WB
Instruction 2 IF RF EX DF WB
Instruction 3 IF RF EX DF WB
Instruction 4 IF RF EX DF WB
Instruction 5 IF RF EX DF WB
Instruction 6 IF RF EX DF WB
Instruction 7 IF RF EX DF WB
Instruction 8 IF RF EX DF WB
Instruction 9 IF RF EX DF WB
End of instruction 1
End of instruction 2
End of instruction 3
End of instruction 4
End of instruction 5
End of instruction 6
End of instruction 7
End of instruction 8
End of instruction 9
An instruction is executed in each clock.

<1> to **<13>** indicate CPU states. In each state, write-back for instruction n, memory access for instruction n + 1, execution for instruction n + 2, decode for instruction n + 3, and fetch for instruction n + 4 are performed concurrently. The processing of a standard instruction requires five clocks from fetch to write-back. The V830 Family can execute a standard instruction in an average of one clock because it can concurrently process five instructions.

9.2 PIPELINE FLOW WHEN EACH INSTRUCTION IS EXECUTED

This section explains the pipeline flow when each instruction is executed.

For the pipeline used for the explanation, the frequency ratio of the internal block to the external block is assumed to be 2 to 1.

In the explanation, waits for the write buffer and waits to synchronize the internal block with the external block are not considered. The pipeline may be placed in the wait state due to the frequency of access to the external bus or some combinations of instructions. Check the pipeline operation of a specific program using a simulator.

9.2.1 Load Instructions

[Related instructions] LD.B, LD.H, and LD.W

[Pipeline] The basic flow of a load instruction is shown below:

	<1>	**<2>**	**<3>**	**<4>**	**<5>**	**<6>**
Load instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The address is calculated in the EX stage. Load data is read from the data cache or memory in the DF stage. In the DF stage, data cache hit/miss is also determined.

9.2.2 Store Instructions

[Related instructions] ST.B, ST.H, and ST.W

[Pipeline] The basic flow of a store instruction is shown below:

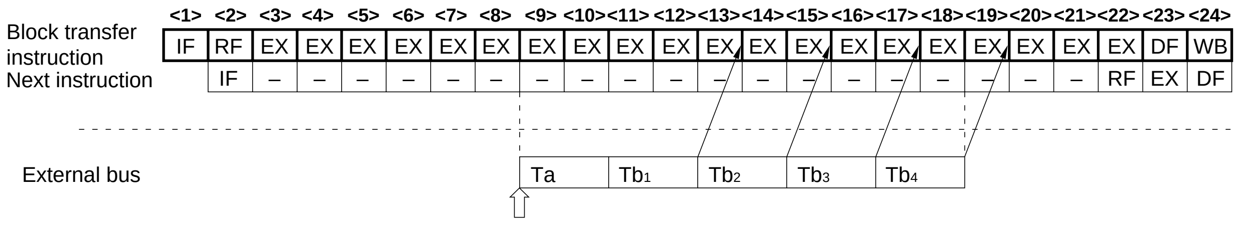
	<1>	**<2>**	**<3>**	**<4>**	**<5>**	**<6>**
Store instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The address is calculated in the EX stage. Data cache hit/miss is determined in the DF stage. Data is stored into the data cache or memory in the WB stage.

9.2.3 Block Transfer Instructions

(1) BILD and BDLD

[Pipeline] The basic flow of a block transfer instruction is shown below:



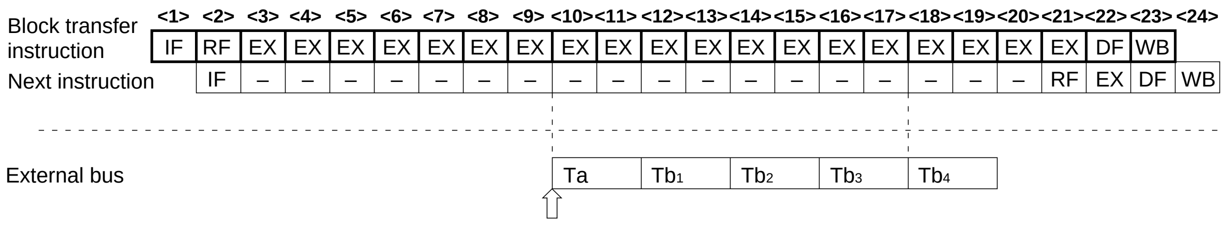
	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>	<9>	<10>	<11>	<12>	<13>	<14>	<15>	<16>	<17>	<18>	<19>	<20>	<21>	<22>	<23>	<24>
Block transfer instruction	IF	RF	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	DF	WB
Next instruction		IF	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	RF	EX	DF
External bus									Ta		Tb1		Tb2		Tb3		Tb4							

To establish synchronization with the bus clock, there may be another delay of one or two cycles.

[Explanation] In EX stage **<14>**, data read in Tb_1 is written into the built-in RAM.
In EX stage **<16>**, data read in Tb_2 is written into the built-in RAM.
In EX stage **<18>**, data read in Tb_3 is written into the built-in RAM.
In EX stage **<20>**, data read in Tb_4 is written into the built-in RAM.
The pipeline hold state is released three clocks after the end of the last bus cycle (Tb_4) in which data is fetched from the external bus.

(2) BIST and BDST

[Pipeline] The basic flow of a block transfer instruction is shown below:



	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>	<9>	<10>	<11>	<12>	<13>	<14>	<15>	<16>	<17>	<18>	<19>	<20>	<21>	<22>	<23>	<24>
Block transfer instruction	IF	RF	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	DF	WB	
Next instruction		IF	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	RF	EX	DF	WB
External bus										Ta		Tb1		Tb2		Tb3		Tb4						

To establish synchronization with the bus clock, there may be another delay of one or two cycles.

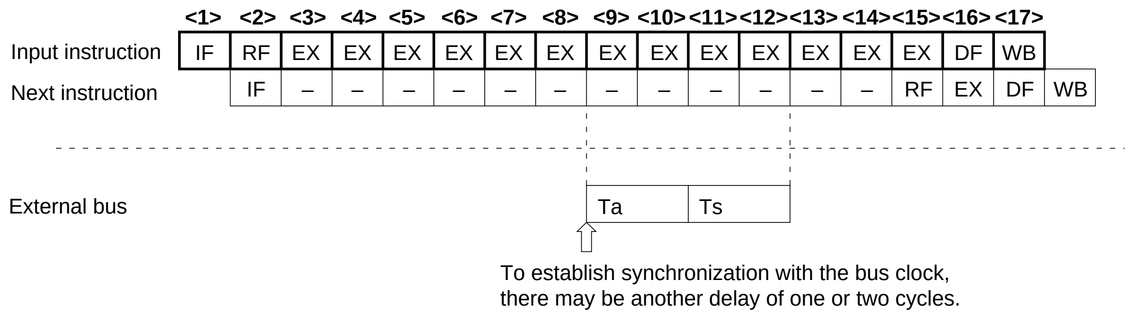
[Explanation] The pipeline hold state is released three clocks after the beginning of the last bus cycle (Tb_4) in which data is output to the external bus.

9.2.4 I/O Instructions

(1) Input instructions

[Related instructions] IN.B, IN.H, and IN.W

[Pipeline] The basic flow of an input instruction is shown below:



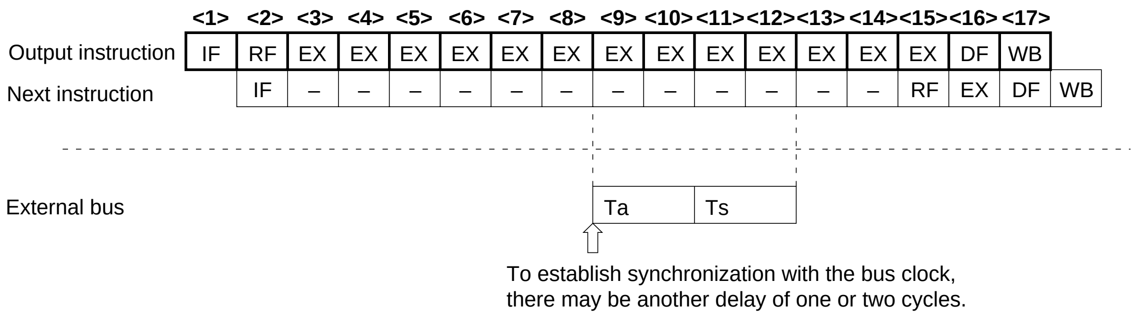
<1> <2> <3> <4> <5> <6> <7> <8> <9> <10> <11> <12> <13> <14> <15> <16> <17>
Input instruction: IF RF EX EX EX EX EX EX EX EX EX EX EX EX EX DF WB
Next instruction: IF – – – – – – – – – – – – RF EX DF WB
External bus
Ta
Ts
To establish synchronization with the bus clock, there may be another delay of one or two cycles.

[Explanation] The pipeline is held until data is fetched from the external bus.
The pipeline hold state is released two clocks after the end of the bus cycle in which data is fetched from the external bus.
In the DF stage, data read from the external bus is fetched. In the WB stage, the register files are written.

(2) Output instructions

[Related instructions] OUT.B, OUT.H, and OUT.W

[Pipeline] The basic flow of an output instruction is shown below:



<1> <2> <3> <4> <5> <6> <7> <8> <9> <10> <11> <12> <13> <14> <15> <16> <17>
Output instruction: IF RF EX EX EX EX EX EX EX EX EX EX EX EX EX DF WB
Next instruction: IF – – – – – – – – – – – – RF EX DF WB
External bus
Ta
Ts
To establish synchronization with the bus clock, there may be another delay of one or two cycles.

[Explanation] The pipeline hold state is released two clocks after the end of the bus cycle in which data is output to the external bus.

9.2.5 Arithmetic Operation Instructions (Other Than the Multiply and Divide Instructions)

[Related instructions] ADD, SUB, ADDI, CMP, MOV, MOVHI, and MOVEA

[Pipeline] The basic flow of an arithmetic operation instruction (other than a multiply/divide instruction) is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>
Arithmetic operation instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The operation result is obtained in the EX stage. The flags are also generated in the EX stage (for other than MOV, MOVHI, and MOVEA).
The register files are written in the WB stage (for other than CMP).

9.2.6 Multiply Instructions

[Related instructions] MUL and MULU

[Pipeline] The basic flow of a multiply instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>	<7>
Multiply instruction	IF	RF	EX	EX	DF	WB	
Next instruction		IF	–	RF	EX	DF	WB

[Explanation] When the operation result is output, it is divided into the high- and low-order words.
In the DF stage, the high-order word is output and written into the register.
The flags are also generated in the DF stage. In the WB stage, the low-order word is output and written into the register.

9.2.7 Divide Instructions

(1) DIV

[Pipeline] The basic flow of the DIV instruction is shown below:

	<1>	<2>	<3>	<4>			<37>	<38>	<39>	<40>	<41>	<42>
DIV instruction	IF	RF	EX	EX			EX	EX	EX	DF	WB	
Next instruction		IF	–				–	–	RF	EX	DF	WB

[Explanation] The DIV instruction stops processing of the next instruction until the 36th cycle (**<38>**) of the EX stage. In the 37th cycle (**<39>**) of the EX stage, the next instruction processing restarts from the RF stage.
The remainder is output in the 35th cycle (**<37>**) of the EX stage and is written in the 37th cycle (**<39>**) of the EX stage. The quotient is output in the 37th cycle (**<39>**) of the EX stage and is written in the WB stage. The flags are generated in the DF stage.

(2) DIVU

[Pipeline] The basic flow of the DIVU instruction is shown below:

	<1>	<2>	<3>	<4>		<35>	<36>	<37>	<38>	<39>	<40>
DIVU instruction	IF	RF	EX	EX		EX	EX	EX	DF	WB	
Next instruction		IF	–			–	–	RF	EX	DF	WB

[Explanation] The DIVU instruction stops processing of the next instruction until the 34th cycle (**<36>**) of the EX stage. In the 35th cycle (**<37>**) of the EX stage, the next instruction processing restarts from the RF stage.
The remainder is output in the 34th cycle (**<36>**) of the EX stage and is written in the DF stage. The quotient is output in the 35th cycle (**<37>**) of the EX stage and is written in the WB stage. The flags are generated in the DF stage.

9.2.8 Multiply/Sum-of-Products Instructions

[Related instructions] MUL3, MULI, MULT3, MAC3, MACI, and MACT3

[Pipeline] The basic flow of a multiply/sum-of-products instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>
Multiply/sum-of-products instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The operation result is generated in the WB stage.
In the RF stage, the register files specified for the first and second operands are read. If the value of the third operand is not forwarded, a structure hazard (2) occurs. For details, see **Section 9.3**.

9.2.9 Signal-Processing Operation Instructions

[Related instructions] SATADD3, SATSUB3, MIN3, and MAX3

[Pipeline] The basic flow of a signal-processing operation instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>
Signal-processing operation instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The flags are generated in the EX stage. The MIN3 and MAX3 instructions generate no flags, however. The operation result is obtained in the DF stage.

9.2.10 Logical Operation Instructions

[Related instructions] OR, AND, XOR, NOT, ORI, ANDI, and XORI

[Pipeline] The basic flow of a logical operation instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>
Logical operation instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The operation result is generated in the EX stage. The flags are generated in the DF stage.

9.2.11 Shift Operation Instructions

[Related instructions] SHL, SHR, SAR, SHLD3, and SHRD3

[Pipeline] The basic flow of a shift operation instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>
Shift operation instruction	IF	RF	EX	DF	WB	
Next instruction		IF	RF	EX	DF	WB

[Explanation] The flags and operation result are generated in the DF stage. The SHLD3 and SHRD3 instructions generate no flags, however.

9.2.12 Branch/Jump Instructions

[Related instructions] Bcond instructions (BGT, BGE, BLT, BLE, BH, BNL, BL, BNH, BE, BNE, BV, BNV, BN, BP, BC, BNC, BZ, BNZ, BR, and NOP), JMP, and JR

[Pipeline] The basic flow of a branch/jump instruction is shown below:

(a) When the branch condition of a jump or Bcond instruction is satisfied

	<1>	<2>	<3>	<4>	<5>	<6>		
Branch/jump instruction	IF	RF	EX	DF	WB			
Next instruction 1		IF	RF					
Next instruction 2			IF					
Branch destination instruction				IF	RF	EX	DF	WB

These instructions are invalidated due to a branch.

(b) When the branch condition of a Bcond instruction is not satisfied

	<1>	<2>	<3>	<4>	<5>	<6>	<7>
Bcond instruction	IF	RF	EX	DF	WB		
Next instruction 1		IF	RF	EX	DF	WB	
Next instruction 2			IF	RF	EX	DF	WB

[Explanation] **(a) When the branch condition of a jump or Bcond instruction is satisfied**
In the EX stage, the branch address is stored in the PC and a branch is made.
(b) When the branch condition of a Bcond instruction is not satisfied
In the EX stage, it is determined that the branch condition is not satisfied, and the subsequent instructions are executed as specified.

9.2.13 Jump and Link Instruction

[Related instruction] JAL

[Pipeline] The basic flow of the JAL instruction is shown below:

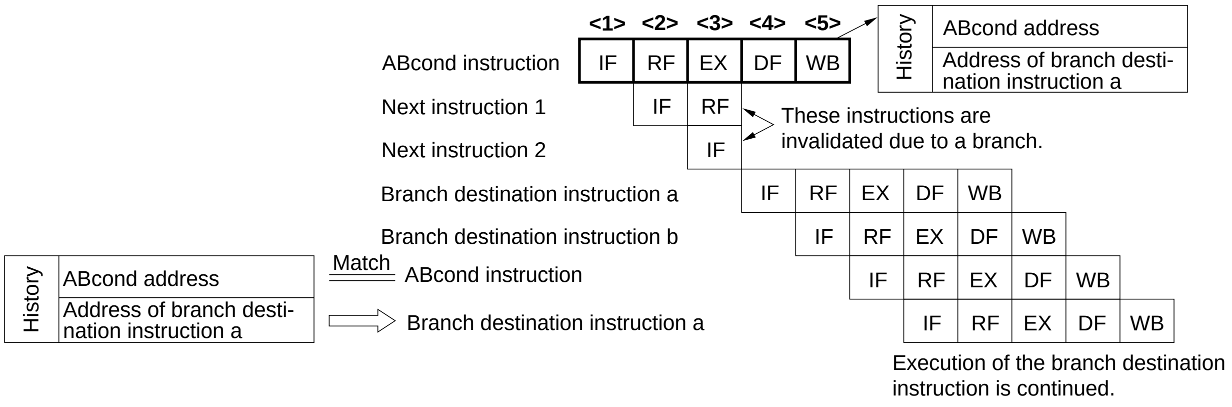
	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>
JAL instruction	IF	RF	EX	EX	DF	WB		
Next instruction		IF	–	Held.				
Branch destination instruction				IF	RF	EX	DF	WB

[Explanation] In the first cycle of the EX stage, the branch address is stored into the PC. In the second cycle of the EX stage, the link PC is calculated. In the WB stage, the value is written into r31.

9.2.14 High-Speed Branch Instructions

[Related instructions] ABcond instructions (ABGT, ABGE, ABLT, ABLE, ABH, ABNL, ABL, ABNH, ABE, ABNE, ABV, ABNV, ABN, ABP, ABC, ABNC, ABZ, ABNZ, and ABR)

[Pipeline] The basic flow of an ABcond instruction is shown below:



<1> <2> <3> <4> <5>
History
ABcond address
Address of branch destination instruction a
ABcond instruction
IF RF EX DF WB
Next instruction 1
Next instruction 2
These instructions are invalidated due to a branch.
Branch destination instruction a
Branch destination instruction b
Match
ABcond instruction
Branch destination instruction a
Execution of the branch destination instruction is continued.

[Explanation] When the condition is satisfied and it is determined that a branch is to be made in the EX stage of the ABcond instruction, for the first branch, the ABcond instruction operates in the same way as an ordinary branch instruction. This is because no branch history has been created. In the WB stage, the addresses of the ABcond and branch destination instructions are written into the branch history. When the second or subsequent branch is to be made for the ABcond instruction (the branch history is not cleared) and the branch history address and PC value match, the address of the branch destination instruction stored in the branch history is set as the next PC value of the ABcond instruction.

9.2.15 Special Instructions

(1) LDSR

[Pipeline] The basic flow of the LDSR instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>	<9>	<10>
LDSR instruction	IF	RF	EX	EX	EX	EX	EX	DF	WB	
Next instruction		IF	–	–	–	–	RF	EX	DF	WB

[Explanation] The new value of the system register is used for the next and subsequent instructions. The next instruction is held until the 4th cycle (**<6>**) of the EX stage. No flag hazard occurs regardless of whether a conditional branch instruction immediately follows the LDSR instruction.

(2) STSR

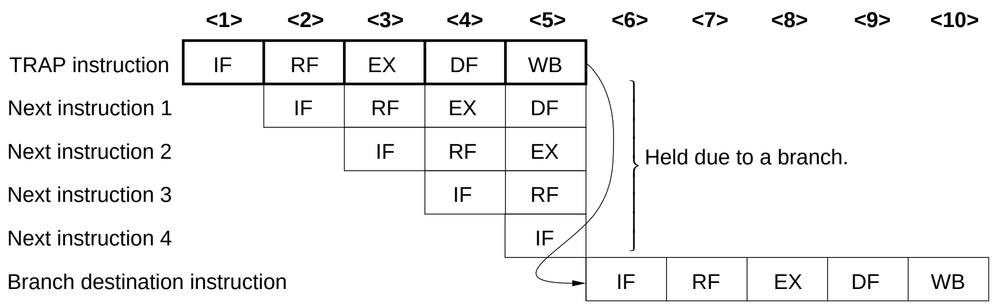
[Pipeline] The basic flow of the STSR instruction is shown below:

	<1>	<2>	<3>	<4>	<5>	<6>	<7>
STSR instruction	IF	RF	EX	EX	DF	WB	
Next instruction		IF	–	RF	EX	DF	WB

[Explanation] In the first cycle of the EX stage (**<3>**), the system register (EIPC, EIPSW, FEPC, FEPSW, PPC, or PPSW) is read. In the WB stage, the system register (ECR, PSW, PIR, TKCW, or HCCW) is read. If the result of the STSR instruction is used immediately after execution of the instruction, a register hazard occurs because the register file is written in the WB stage. For details, see **Section 9.3**.

(3) TRAP

[Pipeline] The basic flow of the TRAP instruction is shown below:

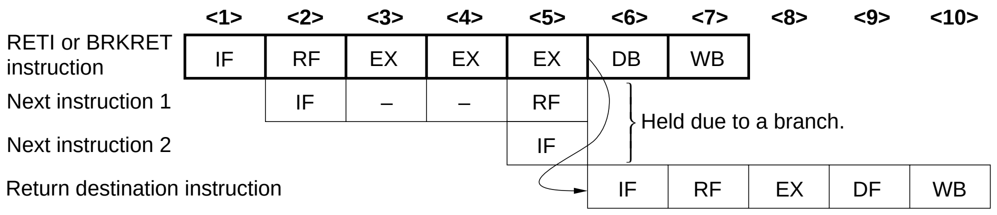


	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>	<9>	<10>
TRAP instruction	IF	RF	EX	DF	WB					
Next instruction 1		IF	RF	EX	DF					
Next instruction 2			IF	RF	EX					
Next instruction 3				IF	RF					
Next instruction 4					IF					
Branch destination instruction						IF	RF	EX	DF	WB

[Explanation] A branch is made after the end of the WB stage for the TRAP instruction to confirm that no exception occurs in an instruction preceding the TRAP instruction before the branch is made.

(4) RETI and BRKRET

[Pipeline] The basic flow of the RETI or BRKRET instruction is shown below:

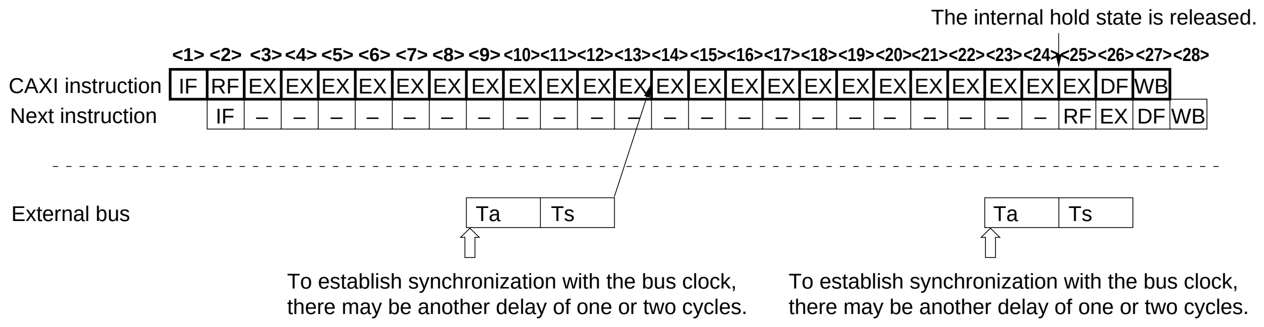


	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>	<9>	<10>
RETI or BRKRET instruction	IF	RF	EX	EX	EX	DB	WB			
Next instruction 1		IF	–	–	RF					
Next instruction 2					IF					
Return destination instruction						IF	RF	EX	DF	WB

[Explanation] The return address is set in the PC and a branch is taken in the 3rd cycle (**<5>**) of the EX stage. The next instruction is held until the 2nd cycle (**<4>**) of the EX stage.

(5) CAXI

[Pipeline] The basic flow of the CAXI instruction is shown below:

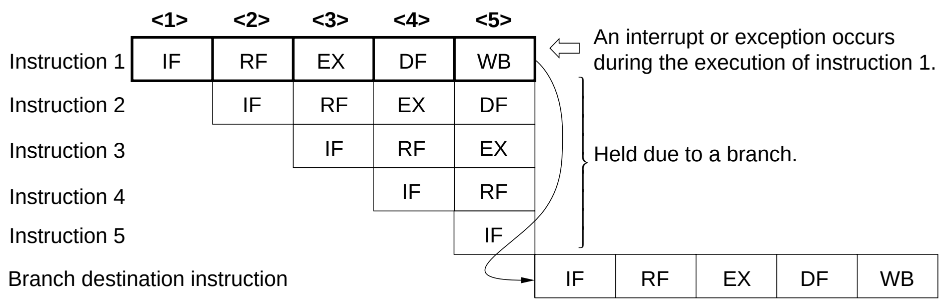

The internal hold state is released.
<1> <2> <3> <4> <5> <6> <7> <8> <9> <10> <11> <12> <13> <14> <15> <16> <17> <18> <19> <20> <21> <22> <23> <24> <25> <26> <27> <28>
CAXI instruction IF RF EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX EX DF WB
Next instruction IF – RF EX DF WB
External bus
Ta Ts
Ta Ts
To establish synchronization with the bus clock, there may be another delay of one or two cycles.
To establish synchronization with the bus clock, there may be another delay of one or two cycles.

[Explanation] The pipeline hold state is released two cycles (**<13>**) after the first bus cycle in which data is output to the external bus.
In the cycle of the EX stage immediately after the hold state is released (**<14>**), data read from the external bus is fetched. In the next cycle of the EX stage (**<15>**), the old data is compared with the read data. In the following cycle of the EX stage (**<16>**), the comparison result is stored.

9.2.16 Address Traps and Interrupts

[Related handling] Interrupt, invalid code exception, and division-by-zero exception

[Pipeline] The basic flow of interrupt/exception handling is shown below:


<1> <2> <3> <4> <5>
Instruction 1 IF RF EX DF WB
An interrupt or exception occurs during the execution of instruction 1.
Instruction 2 IF RF EX DF
Instruction 3 IF RF EX
Instruction 4 IF RF
Instruction 5 IF
Held due to a branch.
Branch destination instruction IF RF EX DF WB

[Explanation] If an interrupt occurs during the execution of instruction 1, processing up to the WB stage for instruction 1 is performed. Then, control is passed to interrupt handling.
If an exception occurs during the execution of instruction 1, processing up to the WB stage of instruction 1 in which the exception occurs is also performed. Then, control is passed to exception handling.

9.3 DISRUPTIONS IN PIPELINE OPERATION

9.3.1 Structure Hazard (1)

[Related processing] A structure hazard (1) occurs if the subsequent instruction may access the same hardware component simultaneously.

[Sample program]

```
ST.W    r3, 200 [r2]
LD.W    100 [r1] , r4
```

[Pipeline]

	<1>	<2>	<3>	<4>	<5>	<6>	<7>
ST.W	IF	RF	EX	DF	WB		
LD.W		IF	RF	–	EX	DF	WB

A structure hazard is detected.

[Explanation] To prevent a malfunction caused by simultaneous access to the same hardware component by the subsequent instruction, execution of the subsequent instruction is held to shift the execution timing.
During the execution of the store instruction (ST.W) in the sample program, the built-in data RAM and data cache are accessed in the WB stage. During the execution of the subsequent load instruction (LD.W), the built-in data RAM and data cache are accessed in the DF stage. If these instructions are executed as is, they will use the same bus in the same cycle. To shift the timing, the hazard function holds the LD.W instruction for one cycle.

9.3.2 Structure Hazard (2)

[Related processing] A structure hazard (2) occurs if the value of the third operand is not forwarded from the previous instruction during the execution of MAC3, MACT3, SHLD3, or SHRD3.

[Pipeline]

	<1>	<2>	<3>	<4>	<5>	<6>	<7>
Target instruction	IF	RF	RF	EX	DF	WB	
Next instruction		IF	–	RF	EX	DF	WB

[Explanation] In the first cycle of the RF stage (**<2>**) for the target instruction, the first and second operands are read, but the third operand is not read. Therefore, if register forwarding is not to be performed for the third operand, the RF stage occurs again and the third operand is read. To perform this operation, the next instruction is held. When the EX stage for the previous instruction is executed, the RF stage for the next instruction is executed (**<4>**).

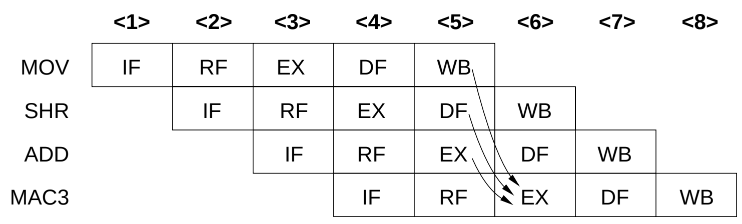
9.3.3 Register Forwarding

[Related processing] Register forwarding occurs if the subsequent instruction uses the operation result before the WB stage in which the register files are written.

[Sample program]

```
MOV    r8, r5
SHR    #1, r4
ADD    r2, r3
MAC3   r5, r4, r3
```

[Pipeline]



	<1>	<2>	<3>	<4>	<5>	<6>	<7>	<8>
MOV	IF	RF	EX	DF	WB			
SHR		IF	RF	EX	DF	WB		
ADD			IF	RF	EX	DF	WB	
MAC3				IF	RF	EX	DF	WB

[Explanation] The operation result to be used by the subsequent instruction is transferred from the EX, DF, and WB stages to the EX stage for the subsequent instruction (forwarding or bypass function). In the sample program, the operation result (r5) of the MOV instruction is forwarded to MAC3 (WB for MOV to EX for MAC3), the operation result (r4) of the SHR instruction is forwarded to MAC3 (DF for SHR to EX for MAC3), and the operation result (r3) of the ADD instruction is forwarded to MAC3 (EX for ADD to EX for MAC3). This function allows the subsequent instruction to start execution without waiting for the end of the WB stage for the previous instruction.

9.3.4 Instruction Code Hazard

[Related processing] An instruction code hazard occurs if a branch is made to a 32-bit instruction across a word boundary (branch due to a branch/jump instruction or interrupt).

[Pipeline]

	<1>	<2>	<3>	<4>	<5>	<6>	<7>
Target instruction	IF	IF	RF	EX	DF	WB	
Next instruction			IF	RF	EX	DF	WB

[Explanation] The CPU fetches an instruction from the instruction cache or RAM word by word. Therefore, for a 32-bit instruction across a word boundary which is executed immediately after a branch, only the first half of code can be fetched in the first cycle of the IF stage. The second cycle of the IF stage is activated to fetch the latter half of the code, and the IF stage for the next instruction is awaited until the code for the previous instruction has been fetched.

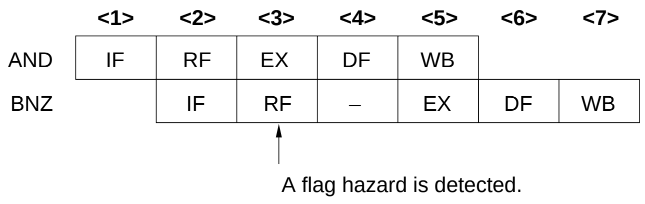
9.3.5 Flag Hazard

[Related processing] A flag hazard occurs during the execution of a conditional branch or SETF instruction following an instruction which generates the flags in the DF or subsequent stage (DF stage for a logical operation instruction; WB stage for a multiply instruction).

[Sample program]

```
AND    #1, r3
BNZ    skip
```

[Pipeline]



[Explanation] Execution of the subsequent instruction is held until flag forwarding because the flags are not generated in the EX stage.

APPENDIX A   INSTRUCTION SUMMARY

A.1 TYPES OF INSTRUCTIONS

A.1.1 Instructions Shared with V810TM

Load/store	LD.B	Load Byte
	LD.H	Load Halfword
	LD.W	Load Word
	ST.B	Store Byte
	ST.H	Store Halfword
	ST.W	Store Word
Arithmetic operation on integers	MOV	Move data
	MOVHI	Move with addition of High-order Immediate data
	ADD	Add
	ADDI	Add Immediate data
	MOVEA	More with Addition
	SUB	Subtract
	MUL	Multiply (signed)
	MULU	Multiply Unsigned
	DIV	Divide (signed)
	DIVU	Divide Unsigned
	CMP	Compare
	SETF	Ser Flag condition
Logical operation	OR	OR (disjunction)
	ORI	OR of Immediate data and register data
	AND	AND (conjunction)
	ANDI	AND of Immediate data and register data
	XOR	Exclusive OR
	XORI	Exclusive OR of Immediate and register data
	NOT	NOT (ones compliment)
	SHL	Shift Logical to the Left
	SHR	Shift Logical to the Right
	SAR	Shift Arithmetic to the Right

Input/output	IN.B	Input Byte from port
	IN.H	Input Halfword from port
	IN.W	Input Word from port
	OUT.B	Output Byte to port
	OUT.H	Output Halfword to port
	OUT.W	Output Word to port
Program control	JMP	Jump unconditional (via register)
	JR	Jump Relative to PC, unconditional
	JAL	Jump and Link
	BGT	Branch on Greater than signed
	BGE	Branch on Greater than or Equal signed
	BLT	Branch on Less than signed
	BLE	Branch on Less than or Equal signed
	BH	Branch on Higher
	BNH	Branch on Not Higher
	BL	Branch on Lower
	BNL	Branch on Not Lower
	BE	Branch on Equal
	BNE	Branch on Not Equal
	BV	Branch on Overflow
	BNV	Branch on No Overflow
	BN	Branch on Negative
	BP	Branch on Positive
	BC	Branch on Carry
	BNC	Branch on No Carry
	BZ	Branch on Zero
	BNZ	Branch on Not Zero
	BR	Branch Always
	NOP	Not always
Special	LDSR	Load to System Register
	STSR	Store contents of System Register
	TRAP	Software Trap
	RETI	Return from Trap or Interrupt
	CAXI	Compare and Exchange Interlocked
	HALT	Halt

A.1.2 Instructions Unique to V810

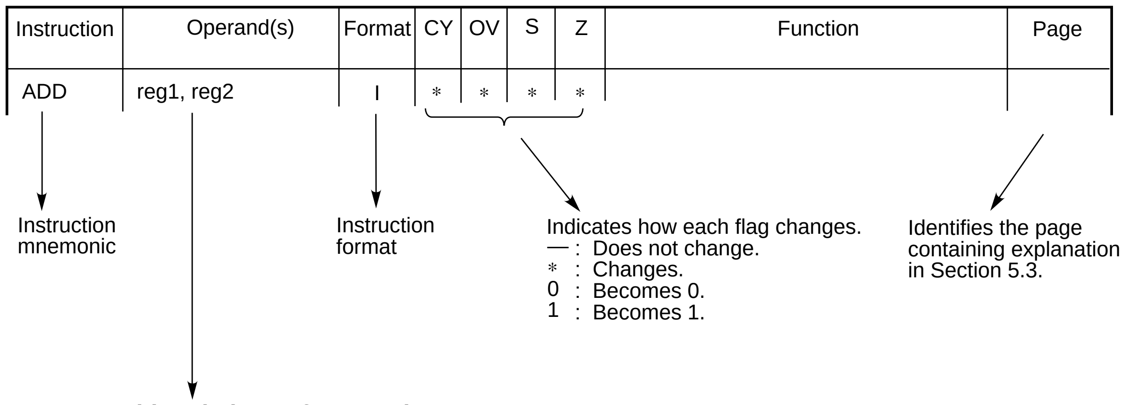
Operation on internal memory	BILD	Block Instruction Load to built-in instruction RAM
	BDLD	Block Data Load to built-in data RAM
	BIST	Block Instruction Store from built-in instruction RAM
	BDST	Block Data Store from built-in data RAM
V830 control	EI	Enable maskable Interrupt
	DI	Disable maskable Interrupt
	STBY	Standby
	BRKRET	Break Return from fatal exception
Instructions for multimedia features	MUL3	Multiply on 3 operands
	MAC3	Multiply and Accumulate on 3 operands
	MULI	Multiply on Immediate and register data
	MACI	Multiply and Accumulate on immediate and register data
	MULT3	Multiply with Truncation on 3 operands
	MACT3	Multiply and Accumulate with Truncation on 3 operands
	SATADD3	Saturatable Addition on 3 operands
	SATSUB3	Saturatable Subtraction on 3 operands
	MIN3	Minimum on 3 operands
	MAX3	Maximum on 3 operands
	SHLD3	Shift to the Left of Double word on 3 operands
	SHRD3	Shift to the Right of Double word on 3 operands
	ABGT	Advanced Branch on Greater than signed
	ABGE	Advanced Branch on Greater than or Equal signed
	ABLT	Advanced Branch on Less than signed
	ABLE	Advanced Branch on Less than or Equal signed
	ABH	Advanced Branch on Higher
	ABNH	Advanced Branch on Not Higher
	ABL	Advanced Branch on Lower
	ABNL	Advanced Branch on Not Lower
	ABE	Advanced Branch on Equal
	ABNE	Advanced Branch on Not Equal
	ABV	Advanced Branch on Overflow
	ABNV	Advanced Branch on No Overflow
	ABN	Advanced Branch on Negative
	ABP	Advanced Branch on Positive

Instructions for multimedia features	ABC	Advanced Branch on Carry
	ABNC	Advanced Branch on No Carry
	ABZ	Advanced Branch on Zero
	ABNZ	Advanced Branch on Not Zero
	ABR	Advanced Branch Always

A.2 INSTRUCTIONS (LISTED ALPHABETICALLY)

The instructions are listed below in alphabetic order of their mnemonics.

Explanation of list format



Abbreviations of operands

Abbreviation	Meaning
reg1	General-purpose register (used as a source register)
reg2	General-purpose register (used mainly as a destination register, but in some instructions, used as a source register)
reg3	General-purpose register (used mainly as a destination register, but in some instructions, used as a source register)
immx	x bits of immediate data
dispx	x-bit displacement
regID	System register number
vector adr	Trap handler address corresponding to trap vector

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
ABC	disp9	III	—	—	—	—	High-speed conditional branch (if Carry) relative to PC.	45
ABE	disp9	III	—	—	—	—	High-speed conditional branch (if Equal) relative to PC.	
ABGE	disp9	III	—	—	—	—	High-speed conditional branch (if Greater than or Equal) relative to PC.	
ABGT	disp9	III	—	—	—	—	High-speed conditional branch (if Greater than) relative to PC.	
ABH	disp9	III	—	—	—	—	High-speed conditional branch (if Higher) relative to PC.	
ABL	disp9	III	—	—	—	—	High-speed conditional branch (if Lower) relative to PC.	
ABLE	disp9	III	—	—	—	—	High-speed conditional branch (if Less than or Equal) relative to PC.	
ABLT	disp9	III	—	—	—	—	High-speed conditional branch (if Less than) relative to PC.	
ABN	disp9	III	—	—	—	—	High-speed conditional branch (if Negative) relative to PC.	
ABNC	disp9	III	—	—	—	—	High-speed conditional branch (if Not Carry) relative to PC.	
ABNE	disp9	III	—	—	—	—	High-speed conditional branch (if Not Equal) relative to PC.	
ABNH	disp9	III	—	—	—	—	High-speed conditional branch (if Not Higher) relative to PC.	
ABNL	disp9	III	—	—	—	—	High-speed conditional branch (if Not Lower) relative to PC.	
ABNV	disp9	III	—	—	—	—	High-speed conditional branch (if Not Overflow) relative to PC.	
ABNZ	disp9	III	—	—	—	—	High-speed conditional branch (if Not Zero) relative to PC.	
ABP	disp9	III	—	—	—	—	High-speed conditional branch (if Positive) relative to PC.	
ABR	disp9	III	—	—	—	—	High-speed unconditional branch (Always) relative to PC.	
ABV	disp9	III	—	—	—	—	High-speed conditional branch (if Overflow) relative to PC.	
ABZ	disp9	III	—	—	—	—	High-speed conditional branch (if Zero) relative to PC.	
ADD	reg1, reg2	I	*	*	*	*	Addition. reg1 is added to reg2 and the sum is written into reg2.	47
	imm5, reg2	II	*	*	*	*	Addition. imm5, sign-extended to a word, is added to reg2 and the sum is written into reg2.	

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
ADDI	imm16, reg1, reg2	V	*	*	*	*	Addition. imm16, sign-extended to a word, is added to reg1, and the sum is written into reg2.	48
AND	reg1, reg2	I	—	0	*	*	AND. reg2 and reg1 are ANDed and the result is written into reg2.	49
ANDI	imm16, reg1, reg2	V	—	0	0	*	AND. reg1 is ANDed with imm16, zero-extended to a word, and result is written into reg2.	50
BC	disp9	III	—	—	—	—	Conditional branch (if Carry) relative to PC.	51
BDLD	[reg1], [reg2]	VII	—	—	—	—	Block transfer. 4 words of data are transferred from external memory to built-in data RAM.	53
BDST	[reg2], [reg1]	VII	—	—	—	—	Block transfer. 4 words of data are transferred from built-in data RAM to external memory.	54
BE	disp9	III	—	—	—	—	Conditional branch (if Equal) relative to PC.	51
BGE	disp9	III	—	—	—	—	Conditional branch (if Greater than or Equal) relative to PC.	
BGT	disp9	III	—	—	—	—	Conditional branch (if Greater than) relative to PC.	
BH	disp9	III	—	—	—	—	Conditional branch (if Higher) relative to PC.	
BILD	[reg1], [reg2]	VII	—	—	—	—	Block transfer. 4 words of data are transferred from external memory to built-in instruction RAM.	55
BIST	[reg2], [reg1]	VII	—	—	—	—	Block transfer. 4 words of data are transferred from built-in instruction RAM to external memory.	56
BL	disp9	III	—	—	—	—	Conditional branch (if Lower) relative to PC.	51
BLE	disp9	III	—	—	—	—	Conditional branch (if Less than or Equal) relative to PC.	
BLT	disp9	III	—	—	—	—	Conditional branch (if Less than) relative to PC.	
BN	disp9	III	—	—	—	—	Conditional branch (if Negative) relative to PC.	
BNC	disp9	III	—	—	—	—	Conditional branch (if Not Carry) relative to PC.	
BNE	disp9	III	—	—	—	—	Conditional branch (if Not Equal) relative to PC.	
BNH	disp9	III	—	—	—	—	Conditional branch (if Not Higher) relative to PC.	

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
BNL	disp9	III	—	—	—	—	Conditional branch (if Not Lower) relative to PC.	51
BNV	disp9	III	—	—	—	—	Conditional branch (if Not Overflow) relative to PC.	
BNZ	disp9	III	—	—	—	—	Conditional branch (if Not Zero) relative to PC.	
BP	disp9	III	—	—	—	—	Conditional branch (if Positive) relative to PC.	
BR	disp9	III	—	—	—	—	Unconditional branch (Always) relative to PC.	
BRKRET		IX	—	—	—	—	Return from fatal exception handling	57
BV	disp9	III	—	—	—	—	Conditional branch (if Overflow) relative to PC.	51
BZ	disp9	III	—	—	—	—	Conditional branch (if Zero) relative to PC.	
CAXI	disp16 [reg1], reg2	VI	*	*	*	*	Inter-processor synchronization in multi-processor system.	58
CMP	reg1, reg2	I	*	*	*	*	Comparison. reg2 is compared with reg1 sign-extended to a word and the condition flag is set according to the result. The comparison involves subtracting reg1 from reg2.	60
	imm5,reg2	II	*	*	*	*	Comparison. reg2 is compared with imm5 sign-extended to a word and the condition flag is set according to the result. The comparison involves subtracting imm5, sign-extended to a word, from reg2.	
DI		II	—	—	—	—	Disable interrupt. Maskable interrupts are disabled. DI instruction cannot disable nonmaskable interrupts.	61
DIV	reg1, reg2	I	—	*	*	*	Division of signed operands. reg2 is divided by reg1 (signed operands). The quotient is stored in reg2 and the remainder in r30. The division is performed so that the sign of the remainder will match that of the dividend.	62
DIVU	reg1, reg2	I	—	0	*	*	Division of unsigned operands. reg2 is divided by reg1 (unsigned operands). The quotient is stored in reg2 and the remainder in r30. The division is performed so that the sign of the remainder will match that of the dividend.	63
EI		II	—	—	—	—	Enable interrupt. Maskable interrupts are enabled. The EI instruction cannot enable nonmaskable interrupts.	64
HALT		IX	—	—	—	—	Processor halt. The processor is placed in sleep mode.	65

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
IN.B	disp16 [reg1], reg2	VI	—	—	—	—	Port input. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit port address. A byte of data is read from the resulting port address, zero-extended to a word, then stored in reg2.	66
IN.H	disp16 [reg1], reg2	VI	—	—	—	—	Port input. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit port address. A halfword of data is read from the produced port address, zero-extended to a word, and stored in reg2. Bit 0 of the unsigned 32-bit port address is masked to 0.	
IN.W	disp16 [reg1], reg2	VI	—	—	—	—	Port input. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit port address. A word of data is read from the resulting port address, then written into reg2. Bits 0 and 1 of the unsigned 32-bit port address are masked to 0.	
JAL	disp26	IV	—	—	—	—	Jump and link. The sum of the current PC and 4 is written into r31. disp26, sign-extended to a word, is added to the PC and the sum is set to the PC for control transfer. Bit 0 of disp26 is masked.	68
JMP	[reg1]	I	—	—	—	—	Indirect unconditional branch via register. Control is passed to the address designated by reg1. Bit 0 of the address is masked to 0.	69
JR	disp26	IV	—	—	—	—	Unconditional branch. disp26, sign-extended to a word, is added to the current PC and control is passed to the address specified by that sum. Bit 0 of disp26 is masked to 0.	70
LD.B	disp16 [reg1], reg2	VI	—	—	—	—	Byte load. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit address. A byte of data is read from the produced address, sign-extended to a word, then written into reg2.	71
LD.H	disp16 [reg1], reg2	VI	—	—	—	—	Halfword load. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit address. A halfword of data is read from the produced address, sign-extended to a word, then written into reg2. Bit 0 of the unsigned 32-bit address is masked to 0.	
LD.W	disp16 [reg1], reg2	VI	—	—	—	—	Word load. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit address. A word of data is read from the produced address, then written into reg2. Bits 0 and 1 of the unsigned 32-bit address are masked to 0.	

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
LDSR	reg2, regID	II	*	*	*	*	Load into system register. The contents of reg2 are set in the system register identified by the system register number (regID).	73
MAC3	reg1, reg2, reg3	VIII	—	—	—	—	Saturatable operation on signed 32-bit operands. reg1 and reg2 are multiplied together as signed integers and the product is added to reg3. [If no overflow has occurred:] The result is stored in reg3. [If an overflow has occurred:] The SAT flag is set. If the result is positive, the positive maximum is written into reg3; if the result is negative, the negative maximum is written into reg3.	74
MACI	imm16, reg1, reg2	V	—	—	—	—	Saturatable operation on signed 32-bit operands. reg1 and imm16, sign-extended to 32 bits, are multiplied together as signed integers and the product is added to reg2 as a signed integer. [If no overflow has occurred:] The result is written into reg2. [If an overflow has occurred:] The SAT flag is set. If the result is positive, the positive maximum is written into reg2; if the result is negative, the negative maximum is written into reg2.	75
MACT3	reg1, reg2, reg3	VIII	—	—	—	—	Saturatable operation on signed 32-bit operands. reg1 and reg2 are multiplied together as signed integers and the high-order 32 bits of the product are added to reg3 as signed integers. [If no overflow has occurred:] The result is written into reg3. [If an overflow has occurred:] The SAT flag is set. If the result is positive, the positive maximum is written into reg3; if the result is negative, the negative maximum is written into reg3.	76
MAX3	reg1, reg2, reg3	VIII	—	—	—	—	Maximum. reg2 and reg1 are compared as signed integers. The larger value is written into reg3.	77
MIN3	reg1, reg2, reg3	VIII	—	—	—	—	Minimum. reg2 and reg1 are compared as signed integers. The smaller value is written into reg3.	78

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
MOV	reg1, reg2	I	—	—	—	—	Data transfer. reg1 is copied to reg2 for data transfer.	79
	imm5, reg2	II	—	—	—	—	Data transfer. imm5, sign-extended to a word, is copied into reg2 for data transfer.	
MOVEA	imm16, reg1, reg2	V	—	—	—	—	Addition. The high-order 16 bits (imm16), sign-extended to a word, are added to reg1 and the sum is written into reg2.	80
MOVHI	imm16, reg1, reg2	V	—	—	—	—	Addition. A word consisting of the high-order 16 bits (imm16) and low-order 16 bits (0) is added to reg1 and the sum is written into reg2.	81
MUL	reg1, reg2	I	—	*	*	*	Multiplication of signed operands. reg2 and reg1 are multiplied together as signed values. The high-order 32 bits of the product (double word) are written into r30 and low-order 32 bits are written into reg2.	82
MUL3	reg1, reg2, reg3	VIII	—	—	—	—	Multiplication of signed 32-bit operands. reg2 and reg1 are multiplied together as signed integers. The high-order 32 bits of the product are written into reg3.	83
MULI	imm16, reg1, reg2	V	—	—	—	—	Saturatable multiplication of signed 32-bit operands. reg1 and imm16, sign-extended to 32 bits, are multiplied together as signed integers. [If no overflow has occurred:] The result is written into reg2. [If an overflow has occurred:] The SAT flag is set. If the result is positive, the positive maximum is written into reg2; if the result is negative, the negative maximum is written into reg2.	84
MULT3	reg1, reg2, reg3	VIII	—	—	—	—	Saturatable multiplication of signed 32-bit operands. reg1 and reg2 are multiplied together as signed integers. The high-order 32 bits of the product are written into reg3.	85
MULU	reg1, reg2	I	—	*	*	*	Multiplication of unsigned operands. reg1 and reg2 are multiplied together as unsigned values. The high-order 32 bits of the product (double word) are written into r30 and the low-order 32 bits are written into reg2.	86
NOP		III	—	—	—	—	No operation.	51
NOT	reg1, reg2	I	—	0	*	*	NOT. The NOT (ones complement) of reg1 is taken and written into reg2.	87
OR	reg1, reg2	I	—	0	*	*	OR. The OR of reg2 and reg1 is taken and written into reg2.	88

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
ORI	imm16, reg1, reg2	V	—	0	*	*	OR. The OR of reg1 and imm16, zero-extended to a word, is taken and written into reg2.	89
OUT.B	reg2, disp16[reg1]	VI	—	—	—	—	Port output. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit port address. The low-order one byte of the data in reg2 is output to the resulting port address.	90
OUT.H	reg2, disp16[reg1]	VI	—	—	—	—	Port output. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit port address. The low-order two bytes of the data in reg2 are output to the resulting port address. Bit 0 of the unsigned 32-bit port address is masked to 0.	
OUT.W	reg2, disp16[reg1]	VI	—	—	—	—	Port output. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit port address. The word of data in reg2 is output to the produced port address. Bits 0 and 1 of the unsigned 32-bit port address are masked to 0.	
RETI		IX	*	*	*	*	Return from trap/interrupt handling routine. The return PC and PSW are read from the system registers so that program execution will return from the trap or interrupt handling routine.	91
SAR	reg1, reg2	I	*	0	*	*	Arithmetic right shift. reg2 is arithmetically shifted to the right by the displacement specified by the low-order five bits of reg1 (MSB value is copied to the MSB in sequence). The result is written into reg2.	92
	imm5, reg2	II	*	0	*	*	Arithmetic right shift. reg2 is arithmetically shifted to the right by the displacement specified by imm5, zero-extended to a word. The result is written into reg2.	
SATADD3	reg1, reg2, reg3	VIII	*	*	*	*	Saturatable addition. reg1 and reg2 are added together as signed integers. [If no overflow has occurred:] The result is written into reg3. [If an overflow has occurred:] The SAT flag is set. If the result is positive, the positive maximum is written into reg3; if the result is negative, the negative maximum is written into reg3.	93

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
SATSUB3	reg1, reg2, reg3	VIII	*	*	*	*	Saturatable subtraction. reg1 is subtracted from reg2 as signed integers. [If no overflow has occurred:] The result is written into reg3. [If an overflow has occurred:] The SAT flag is set. If the result is positive, the positive maximum is written into reg3; if the result is negative, the negative maximum is written into reg3.	94
SETF	imm5, reg2	II	—	—	—	—	Set flag condition. reg2 is set to 1 if the condition specified by the low-order four bits of imm5 matches the condition flag; otherwise it is set to 0.	95
SHL	reg1, reg2	I	*	0	*	*	Logical left shift. reg2 is logically shifted to the left (0 is put on the LSB) by the displacement specified by the low-order five bits of reg1. The result is written into reg2.	97
	imm5, reg2	II	*	0	*	*	Logical left shift. reg2 is logically shifted to the left by the displacement specified by imm5, zero-extended to a word. The result is written into reg2.	
SHLD3	reg1, reg2, reg3	VIII	—	—	—	—	Left shift of concatenation. The 64 bits consisting of reg3 (high order) and reg2 (low order) are logically shifted to the left by the displacement specified by the low-order five bits of reg1. The high-order 32 bits of the result are written into reg3.	98
SHR	reg1, reg2	I	*	0	*	*	Logical right shift. reg2 is logically shifted to the right by the displacement specified by the low-order five bits of reg1 (0 is put on the MSB). The result is written into reg2.	99
	imm5, reg2	II	*	0	*	*	Logical right shift. reg2 is logically shifted to the right by the displacement specified by imm5, zero-extended to a word. The result is written into reg2.	
SHRD3	reg1, reg2, reg3	VIII	—	—	—	—	Right shift of concatenation. The 64 bits consisting of reg3 (high order) and reg2 (low order) are logically shifted to the right by the displacement specified by the low-order five bits of reg1. The low-order 32 bits of the result are written into reg3.	100

Instruction	Operand(s)	Format	CY	OV	S	Z	Function	Page
ST.B	reg2, disp16[reg1]	VI	—	—	—	—	Byte store. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit address. The low-order one byte of data in reg2 is stored at the resulting address.	101
ST.H	reg2, disp16[reg1]	VI	—	—	—	—	Halfword store. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit address. The low-order two bytes of the data in reg2 are stored at the resulting address. Bit 0 of the unsigned 32-bit address is masked to 0.	
ST.W	reg2, disp16[reg1]	VI	—	—	—	—	Word store. disp16, sign-extended to a word, is added to reg1 to produce an unsigned 32-bit address. The word of data in reg2 is stored at the resulting address. Bits 0 and 1 of the unsigned 32-bit address are masked to 0.	
STBY		IX	—	—	—	—	Processor stop. The processor is placed in stop mode.	102
STSR	regID, reg2	II	—	—	—	—	System register store. The contents of the system register identified by the system register number (regID) are set in reg2.	103
SUB	reg1, reg2	I	*	*	*	*	Subtraction. reg1 is subtracted from reg2. The difference is written into reg2.	104
TRAP	vector	II	—	—	—	—	Software trap. The return PC and PSW are saved in the system registers: PSW.EP = 1 → Save in FEPC, FEPSW PSW.EP = 0 → Save in EIPC, EIPSW The exception code is set in the ECR: PSW.EP = 1 → Set in FECC PSW.EP = 0 → Set in EICC PSW flags are set: PSW.EP = 1 → Set NP and ID PSW.EP = 0 → Set EP and ID Program execution jumps to the trap handler address corresponding to the trap vector (0-31) specified by vector and begins exception handling.	105
XOR	reg1, reg2	I	—	0	*	*	Exclusive OR. The exclusive OR of reg2 and reg1 is taken and written into reg2.	107
XORI	imm16, reg1, reg2	V	—	0	*	*	Exclusive OR. The exclusive OR of reg1 and imm16, zero-extended to a word, is taken and written into reg2.	108

APPENDIX B OPERATION CODE MAP

Operation code map

Bits 15-10	Instruction syntax		Format	Sub-operation code
000000	MOV	reg1, reg2	I	
000001	ADD	reg1, reg2	I	
000010	SUB	reg1, reg2	I	
000011	CMP	reg1, reg2	I	
000100	SHL	reg1, reg2	I	
000101	SHR	reg1, reg2	I	
000110	JMP	[reg1]	I	
000111	SAR	reg1, reg2	I	
001000	MUL	reg1, reg2	I	
001001	DIV	reg1, reg2	I	
001010	MULU	reg1, reg2	I	
001011	DIVU	reg1, reg2	I	
001100	OR	reg1, reg2	I	
001101	AND	reg1, reg2	I	
001110	XOR	reg1, reg2	I	
001111	NOT	reg1, reg2	I	
010000	MOV	imm5, reg2	II	
010001	ADD	imm5, reg2	II	
010010	SETF	imm5, reg2	II	
010011	CMP	imm5, reg2	II	
010100	SHL	imm5, reg2	II	
010101	SHR	imm5, reg2	II	
010110	EI		II	
010111	SAR	imm5, reg2	II	
011000	TRAP	vector	II	
011001	RETI		IX	0
011001	BRKRET		IX	1
011010	HALT		IX	0
011010	STBY		IX	1
011100	LDSR	reg2, regID	II	
011101	STSR	regID, reg2	II	
011110	DI		II	

Bits 15-10	Instruction syntax		Format	Sub-operation code
100XXX	Bcond		III	0
100XXX	ABcond		III	1
101000	MOVEA	imm16, reg1, reg2	V	
101001	ADDI	imm16, reg1, reg2	V	
101010	JR	disp26	IV	
101011	JAL	disp26	IV	
101100	ORI	imm16, reg1, reg2	V	
101101	ANDI	imm16, reg1, reg2	V	
101110	XORI	imm16, reg1, reg2	V	
101111	MOVHI	imm16, reg1, reg2	V	
110000	LD.B	disp16[reg1], reg2	VI	
110001	LD.H	disp16[reg1], reg2	VI	
110010	MULI	imm16, reg1, reg2	V	
110011	LD.W	disp16[reg1], reg2	VI	
110100	ST.B	reg2, disp16[reg1]	VI	
110101	ST.H	reg2, disp16[reg1]	VI	
110110	MACI	imm16, reg1, reg2	V	
110111	ST.W	reg2, disp16[reg1]	VI	
111000	IN.B	disp16[reg1], reg2	VI	
111001	IN.H	disp16[reg1], reg2	VI	
111010	CAXI	disp16[reg1], reg2	VI	
111011	IN.W	disp16[reg1], reg2	VI	
111100	OUT.B	reg2, disp16[reg1]	VI	
111101	OUT.H	reg2, disp16[reg1]	VI	
111110	Special		VII/VIII	
111111	OUT.W	reg2, disp16[reg1]	VI	

Operation code field

Bits 15-13 \ Bits 12-10	000	001	010	011	100	101	110	111
000	MOV	ADD	SUB	CMP	SHL	SHR	JMP	SAR
001	MUL	DIV	MULU	DIVU	OR	AND	XOR	NOT
010	MOV	ADD	SETF	CMP	SHL	SHR	EI	SAR
011	TRAP	RETI BRKRET	HALT STBY		LDSR	STSR	DI	
100	Bcond/ABcond							
101	MOVEA	ADDI	JR	JAL	ORI	ANDI	XORI	MOVHI
110	LD.B	LD.H	MULI	LD.W	ST.B	ST.H	MACI	ST.W
111	IN.B	IN.H	CAXI	IN.W	OUT.B	OUT.H	Special	OUT.W

Conditional branch (Bcond/ABcond) condition code field

Bit 12 \ Bits 11-9	000	001	010	011	100	101	110	111
0	BV	BC/BL	BZ/BE	BNH	BN	BR	BLT	BLE
1	BNV	BNC/BNL	BNZ/BNE	BH	BP	NOP	BGE	BGT

Special operation code field

Bits 31-29 \ Bits 28-26	000	001	010	011	100	101	110	111
000								
001								
010	SATADD3	SATSUB3	MIN3	MAX3				
011	SHLD3	SHRD3			MACT3	MAC3	MULT3	MUL3
100	BILD	BDLD	BIST	BDST				
101								
110								
111								

[MEMO]

APPENDIX C INDEX

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America NEC Electronics Inc. Corporate Communications Dept. Fax: 1-800-729-9288 1-408-588-6130	**Hong Kong, Philippines, Oceania** NEC Electronics Hong Kong Ltd. Fax: +852-2886-9022/9044	**Asian Nations except Philippines** NEC Electronics Singapore Pte. Ltd. Fax: +65-250-3583
Europe NEC Electronics (Europe) GmbH Technical Documentation Dept. Fax: +49-211-6503-274	**Korea** NEC Electronics Hong Kong Ltd. Seoul Branch Fax: 02-528-4411	**Japan** NEC Semiconductor Technical Hotline Fax: 044-435-9608
South America NEC do Brasil S.A. Fax: +55-11-6462-6829	**Taiwan** NEC Electronics Taiwan Ltd. Fax: 02-2719-5951	

I would like to report the following error/make the following suggestion:

Document title: ______

Document number: ______ Page number: ______

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❏	❏	❏	❏
Technical Accuracy	❏	❏	❏	❏
Organization	❏	❏	❏	❏