NEC

# User's Manual

# V850E1

## 32-Bit Microprocessor Core

## Architecture

Document No. U14559EJ3V1UM00 (3rd edition)
Date Published February 2004 N CP(K)

© NEC Electronics Corporation 1999
Printed in Japan

**[MEMO]**

## NOTES FOR CMOS DEVICES

### ① VOLTAGE APPLICATION WAVEFORM AT INPUT PIN

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between $V_{IL}$ (MAX) and $V_{IH}$ (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between $V_{IL}$ (MAX) and $V_{IH}$ (MIN).

### ② HANDLING OF UNUSED INPUT PINS

Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to $V_{DD}$ or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

### ③ PRECAUTION AGAINST ESD

A 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 when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up 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 benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

### ④ STATUS BEFORE INITIALIZATION

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

These commodities, technology or software, must be exported in accordance with the export administration regulations of the exporting country.
Diversion contrary to the law of that country is prohibited.

- **The information in this document is current as of February, 2004. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02. 11-1

# Regional Information

Some information contained in this document may vary from country to country. Before using any NEC Electronics product in your application, please contact the NEC Electronics 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.

**[GLOBAL SUPPORT]**
**http://www.necel.com/en/support/support.html**

**NEC Electronics America, Inc. (U.S.)**
Santa Clara, California
Tel: 408-588-6000
800-366-9782

**NEC Electronics (Europe) GmbH**
Duesseldorf, Germany
Tel: 0211-65030

- **Sucursal en España**
  Madrid, Spain
  Tel: 091-504 27 87
- **Succursale Française**
  Vélizy-Villacoublay, France
  Tel: 01-30-67 58 00
- **Filiale Italiana**
  Milano, Italy
  Tel: 02-66 75 41
- **Branch The Netherlands**
  Eindhoven, The Netherlands
  Tel: 040-244 58 45
- **Tyskland Filial**
  Taeby, Sweden
  Tel: 08-63 80 820
- **United Kingdom Branch**
  Milton Keynes, UK
  Tel: 01908-691-133

**NEC Electronics Hong Kong Ltd.**
Hong Kong
Tel: 2886-9318

**NEC Electronics Hong Kong Ltd.**
Seoul Branch
Seoul, Korea
Tel: 02-558-3737

**NEC Electronics Shanghai Ltd.**
Shanghai, P.R. China
Tel: 021-5888-5400

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

**NEC Electronics Singapore Pte. Ltd.**
Novena Square, Singapore
Tel: 6253-8311

J04.1

# PREFACE

**Target Readers**

This manual is intended for users who wish to understand the functions of the V850E1 CPU core for designing application systems using the V850E1 CPU core.

**Purpose**

This manual is intended to give users an understanding of the architecture of the V850E1 CPU core described in the **Organization** below.

**Organization**

This manual contains the following information.

- Register set
- Data types
- Instruction format and instruction set
- Interrupts and exceptions
- Pipeline

**How to Use This Manual**

It is assumed that the reader of this manual has general knowledge in the fields of electrical engineering, logic circuits, and microcontrollers.

To learn about the hardware functions,

→ Read **Hardware User's Manual** of each product.

To learn about the functions of a specific instruction in detail,

→ Read **CHAPTER 5 INSTRUCTIONS**.

The mark ★ shows major revised points.

**Product Types**

This manual explains the products divided into types.
Before reading this manual, check the corresponding product type.

| Product Type | Product Name |
|---|---|
| Type A | NU85E CPU core |
| Type B | NU85ET CPU core |
| Type C | NB85E, NB85ET CPU core |
| Type D | V850E/IA1, V850E/IA2, V850E/MA1, V850E/SV2 |
| Type E | V850E/IA3, V850E/IA4, V850E/MA3 |
| Type F | V850E/MA2, V850E/ME2 |

**Conventions**

| | |
|---|---|
| Data significance: | Higher digits on the left and lower digits on the right |
| Active low representation: | ×××B (B is appended to pin or signal name) |
| **Note**: | Footnote for item marked with **Note** in the text |
| **Caution**: | Information requiring particular attention |
| **Remark**: | Supplementary information |
| Numerical representation: | Binary ... ×××× or ××××B |
| | Decimal ... ×××× |
| | Hexadecimal ... ××××H |

Prefix indicating the power of 2 (address space, memory capacity):

| | |
|---|---|
| K (Kilo): | $2^{10} = 1,024$ |
| M (Mega): | $2^{20} = 1,024^2$ |
| G (Giga): | $2^{30} = 1,024^3$ |

## CONTENTS


**CHAPTER 1 GENERAL ..... 12**

**1.1 Features ..... 13**
**1.2 Internal Configuration ..... 14**

**CHAPTER 2 REGISTER SET ..... 15**

**2.1 Program Registers ..... 16**
**2.2 System Registers ..... 18**
2.2.1 Interrupt status saving registers (EIPC, EIPSW) ..... 19
2.2.2 NMI status saving registers (FEPC, FEPSW) ..... 20
2.2.3 Exception cause register (ECR) ..... 20
2.2.4 Program status word (PSW) ..... 21
2.2.5 CALLT caller status saving registers (CTPC, CTPSW) ..... 23
2.2.6 Exception/debug trap status saving registers (DBPC, DBPSW) ..... 24
2.2.7 CALLT base pointer (CTBP) ..... 25
2.2.8 Debug interface register (DIR) ..... 26
2.2.9 Breakpoint control registers 0 and 1 (BPC0, BPC1) ..... 29
2.2.10 Program ID register (ASID) ..... 30
2.2.11 Breakpoint address setting registers 0 and 1 (BPAV0, BPAV1) ..... 31
2.2.12 Breakpoint address mask registers 0 and 1 (BPAM0, BPAM1) ..... 31
2.2.13 Breakpoint data setting registers 0 and 1 (BPDV0, BPDV1) ..... 32
2.2.14 Breakpoint data mask registers 0 and 1 (BPDM0, BPDM1) ..... 32

**CHAPTER 3 DATA TYPES ..... 33**

**3.1 Data Format ..... 33**
**3.2 Data Representation ..... 35**
3.2.1 Integer ..... 35
3.2.2 Unsigned integer ..... 35
3.2.3 Bit ..... 35
**3.3 Data Alignment ..... 36**

**CHAPTER 4 ADDRESS SPACE ..... 37**

**4.1 Memory Map ..... 38**
**4.2 Addressing Mode ..... 39**
4.2.1 Instruction address ..... 39
4.2.2 Operand address ..... 41

**CHAPTER 5 INSTRUCTIONS ..... 43**

**5.1 Instruction Format ..... 43**
**5.2 Outline of Instructions ..... 47**
**5.3 Instruction Set ..... 51**
ADD ..... 53
ADDI ..... 54
AND ..... 55
ANDI ..... 56

Bcond ..... 57
BSH ..... 59
BSW ..... 60
CALLT ..... 61
CLR1 ..... 62
CMOV ..... 63
CMP ..... 64
CTRET ..... 65
DBRET ..... 66
DBTRAP ..... 67
DI ..... 68
DISPOSE ..... 69
DIV ..... 71
DIVH ..... 72
DIVHU ..... 74
DIVU ..... 75
EI ..... 76
HALT ..... 77
HSW ..... 78
JARL ..... 79
JMP ..... 80
JR ..... 81
LD.B ..... 82
LD.BU ..... 83
LD.H ..... 84
LD.HU ..... 86
LD.W ..... 88
LDSR ..... 90
MOV ..... 91
MOVEA ..... 92
MOVHI ..... 93
MUL ..... 94
MULH ..... 96
MULHI ..... 97
MULU ..... 98
NOP ..... 100
NOT ..... 101
NOT1 ..... 102
OR ..... 103
ORI ..... 104
PREPARE ..... 105
RETI ..... 107
SAR ..... 109
SASF ..... 110
SATADD ..... 111
SATSUB ..... 112
SATSUBI ..... 113
SATSUBR ..... 114

SET1 ..... 115
SETF ..... 116
SHL ..... 118
SHR ..... 119
SLD.B ..... 120
SLD.BU ..... 121
SLD.H ..... 122
SLD.HU ..... 124
SLD.W ..... 126
SST.B ..... 128
SST.H ..... 129
SST.W ..... 131
ST.B ..... 133
ST.H ..... 134
ST.W ..... 136
STSR ..... 138
SUB ..... 139
SUBR ..... 140
SWITCH ..... 141
SXB ..... 142
SXH ..... 143
TRAP ..... 144
TST ..... 145
TST1 ..... 146
XOR ..... 147
XORI ..... 148
ZXB ..... 149
ZXH ..... 150
**5.4 Number of Instruction Execution Clock Cycles ..... 151**

**CHAPTER 6 INTERRUPTS AND EXCEPTIONS ..... 155**

**6.1 Interrupt Servicing ..... 156**
6.1.1 Maskable interrupts ..... 156
6.1.2 Non-maskable interrupts ..... 158
**6.2 Exception Processing ..... 159**
6.2.1 Software exceptions ..... 159
6.2.2 Exception trap ..... 160
6.2.3 Debug trap ..... 161
**6.3 Restoring from Interrupt/Exception Processing ..... 162**
6.3.1 Restoring from interrupt and software exception ..... 162
6.3.2 Restoring from exception trap and debug trap ..... 163

**CHAPTER 7 RESET ..... 164**

**7.1 Register Status After Reset ..... 164**
**7.2 Starting Up ..... 165**

**CHAPTER 8 PIPELINE ..... 166**

**8.1 Features ..... 167**

8.1.1 Non-blocking load/store ..... 168
8.1.2 2-clock branch ..... 169
8.1.3 Efficient pipeline processing ..... 170
**8.2 Pipeline Flow During Execution of Instructions ..... 171**
8.2.1 Load instructions ..... 171
8.2.2 Store instructions ..... 172
8.2.3 Multiply instructions ..... 172
8.2.4 Arithmetic operation instructions ..... 173
8.2.5 Saturated operation instructions ..... 174
8.2.6 Logical operation instructions ..... 174
8.2.7 Branch instructions ..... 174
8.2.8 Bit manipulation instructions ..... 176
8.2.9 Special instructions ..... 176
8.2.10 Debug function instructions ..... 181
**8.3 Pipeline Disorder ..... 182**
8.3.1 Alignment hazard ..... 182
8.3.2 Referencing execution result of load instruction ..... 183
8.3.3 Referencing execution result of multiply instruction ..... 184
8.3.4 Referencing execution result of LDSR instruction for EIPC and FEPC ..... 185
8.3.5 Cautions when creating programs ..... 185
**8.4 Additional Items Related to Pipeline ..... 186**
8.4.1 Harvard architecture ..... 186
8.4.2 Short path ..... 187

★ **CHAPTER 9 SHIFTING TO DEBUG MODE ..... 189**
**9.1 How to Shift to Debug Mode ..... 189**
**9.2 Cautions ..... 195**

★ **APPENDIX A NOTES ..... 197**
**A.1 Restriction on Conflict Between sld Instruction and Interrupt request ..... 197**
A.1.1 Description ..... 197
A.1.2 Countermeasure ..... 197

**APPENDIX B INSTRUCTION LIST ..... 198**

**APPENDIX C INSTRUCTION OPCODE MAP ..... 212**

**APPENDIX D DIFFERENCES WITH ARCHITECTURE OF V850 CPU ..... 217**

**APPENDIX E INSTRUCTIONS ADDED FOR V850E1 CPU COMPARED WITH V850 CPU ..... 219**

**APPENDIX F INDEX ..... 221**

★ **APPENDIX G REVISION HISTORY ..... 224**
**G.1 Major Revisions in This Edition ..... 224**
**G.2 History of Revisions up to This Edition ..... 225**

# CHAPTER 1 GENERAL

Real-time control systems are used in a wide range of applications, including:

- office equipment such as HDDs (Hard Disk Drives), PPCs (Plain Paper Copiers), printers, and facsimiles,
- automobile electronics such as engine control systems and ABSs (Antilock Braking Systems), and
- factory automation equipment such as NC (Numerical Control) machine tools and various controllers.

The great majority of these systems conventionally employ 8-bit or 16-bit microcontrollers. However, the performance level of these microcontrollers has become inadequate in recent years as control operations have risen in complexity, leading to the development of increasingly complicated instruction sets and hardware design. As a result, the need has arisen for a new generation of microcontrollers operable at much higher frequencies to achieve an acceptable level of performance under today's more demanding requirements.

The V850 Series of microcontrollers was developed to satisfy this need. This series uses RISC architecture that can provide maximum performance with simpler hardware, allowing users to obtain a performance approximately 15 times higher than that of the existing 78K/III Series and 78K/IV Series of CISC single-chip microcontrollers at a lower total cost.

In addition to the basic instructions of conventional RISC CPUs, the V850 Series is provided with special instructions such as saturation, bit manipulation, and multiply/divide (executed by a hardware multiplier) instructions, which are especially suited to digital servo control systems. Moreover, instruction formats are designed for maximum compiler coding efficiency, allowing the reduction of object code sizes.

The V850E1 CPU is a 32-bit RISC CPU core for ASIC, newly developed as the CPU core central to system LSI in the current age of system-on-a-chip. This core includes not only the control functions of the V850 CPU, the CPU core incorporated in the V850 Series, but also supports data processing through its enhanced external bus interface performance, and the addition of features such as C language switch statement processing, table lookup branching, stack frame creation/deletion, data conversion, and other high-level language supporting instructions.

In addition, because the instruction codes are upwardly compatible with the V850 CPU at the object code level, the software resources of systems that incorporate the V850 CPU can be used unchanged.

## 1.1 Features

(1) High-performance 32-bit architecture for embedded control
- Number of instructions: 83
- 32-bit general-purpose registers: 32
- Load/store instructions in long/short format
- 3-operand instruction
- 5-stage pipeline of 1 clock cycle per stage
- Hardware interlock on register/flag hazards
- Memory space Program space: 64 MB linear
  Data space: 4 GB linear

(2) Special instructions
- Saturation operation instructions
- Bit manipulation instructions
- Multiply instructions (On-chip hardware multiplier executing multiplication in 1 clock)
  16 bits × 16 bits → 32 bits
  32 bits × 32 bits → 32 bits or 64 bits

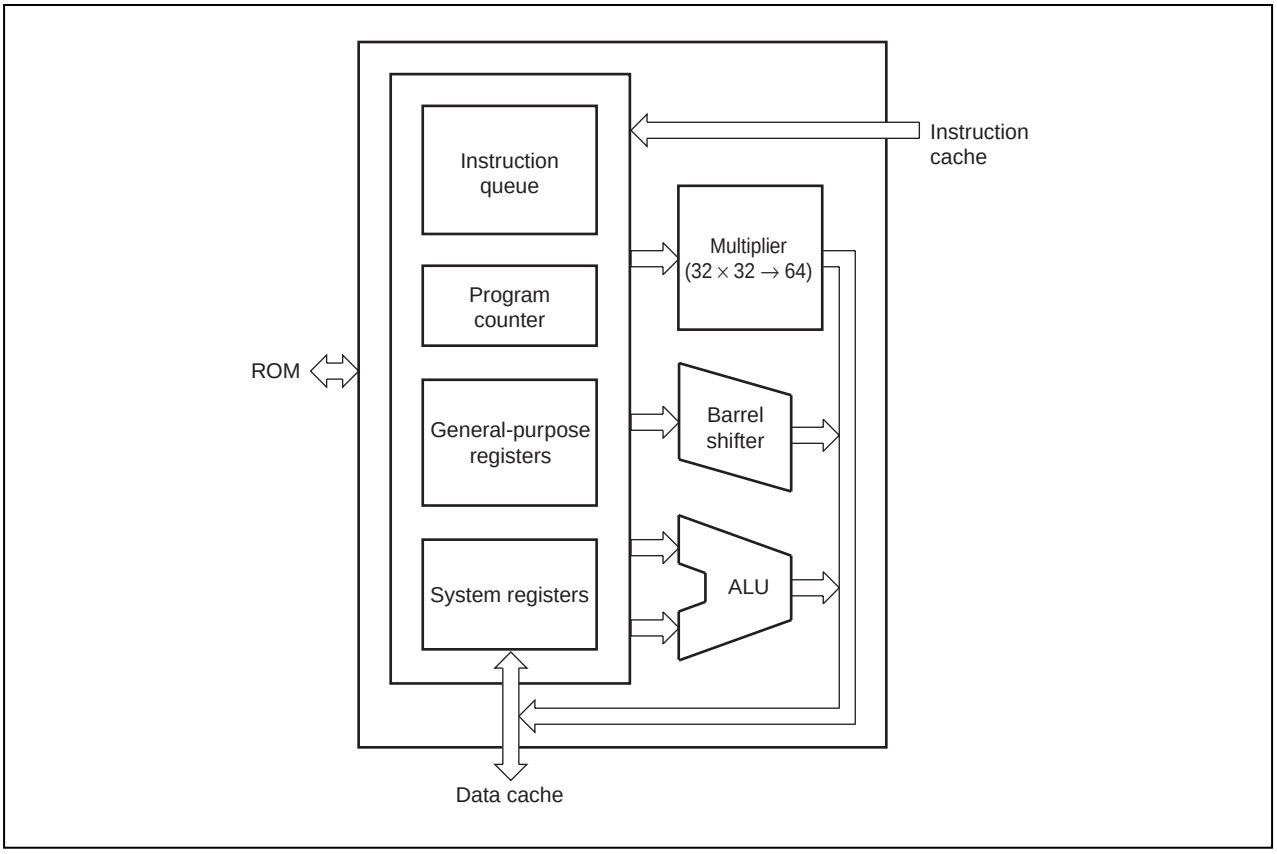
## 1.2 Internal Configuration

The V850E1 CPU executes almost all instructions such as address calculation, arithmetic and logical operation, and data transfer in one clock by using a 5-stage pipeline.

It contains dedicated hardware such as a multiplier (32 × 32 bits) and a barrel shifter (32 bits/clock) to execute complicated instructions at high speeds.

Figure 1-1 shows the internal block diagram.

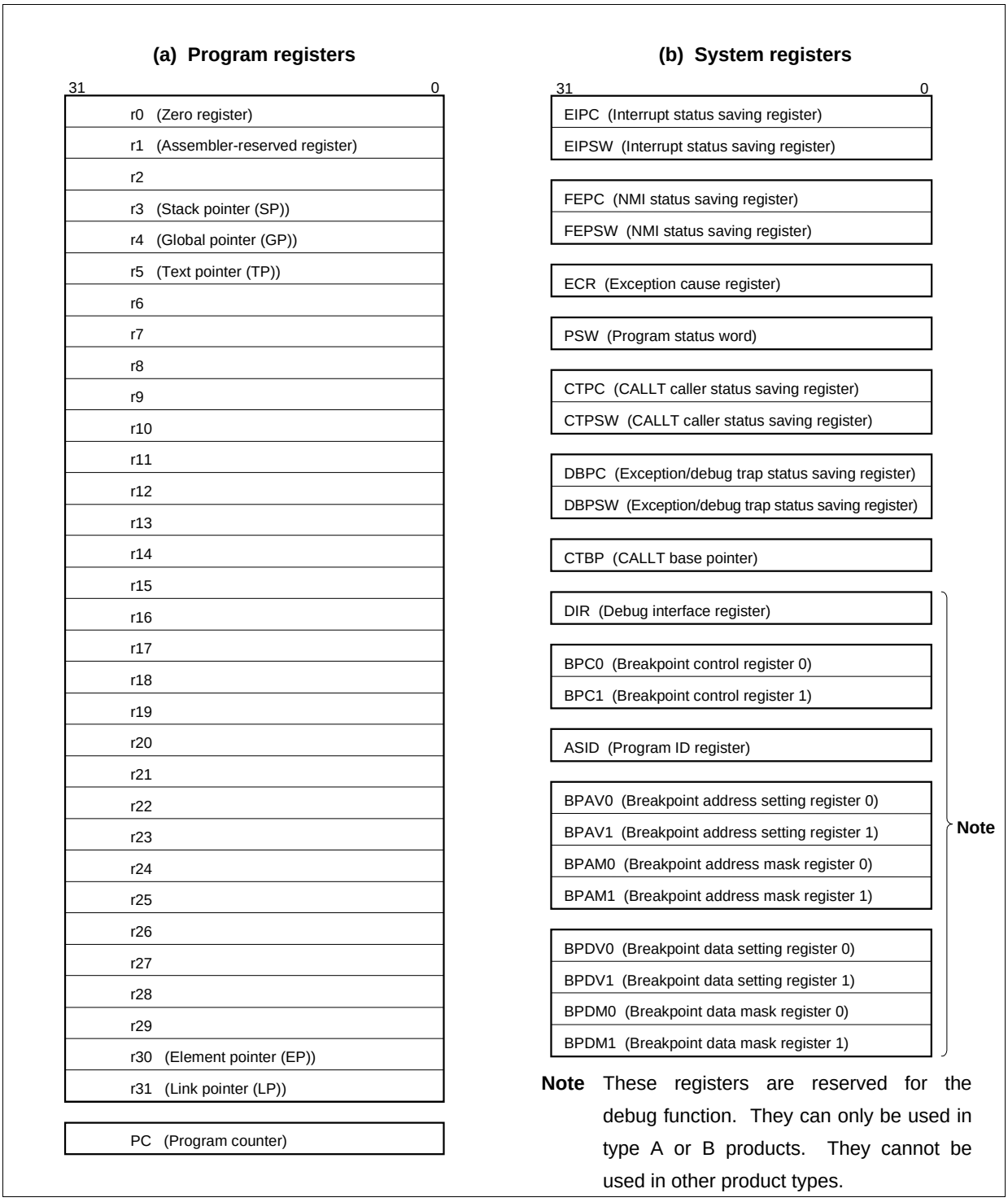
**Figure 1-1. Internal Block Diagram of V850E1 CPU**



# CHAPTER 2 REGISTER SET

The registers can be classified into two types: program registers that can be used for general programming, and system registers that can control the execution environment. All the registers are 32 bits wide.

**Figure 2-1. Registers**




★

**Note** These registers are reserved for the debug function. They can only be used in type A or B products. They cannot be used in other product types.

## 2.1 Program Registers

The program registers include general-purpose registers (r0 to r31) and a program counter (PC).

**Table 2-1. Program Registers**

| Program Registers | Name | Function | Description |
|---|---|---|---|
| General-purpose registers | r0 | Zero register | Always holds 0. |
| | r1 | Assembler-reserved register | Used as working register for address generation. |
| | r2 | Address/data variable register (when the real-time OS to be used is not using r2) | |
| | r3 | Stack pointer (SP) | Used for stack frame generation when function is called. |
| | r4 | Global pointer (GP) | Used to access global variable in data area. |
| | r5 | Text pointer (TP) | Used as register for pointing to start address of text area (area where program code is placed) |
| | r6 to r29 | Address/data variable registers | |
| | r30 | Element pointer (EP) | Used as base pointer for address generation when memory is accessed. |
| | r31 | Link pointer (LP) | Used when compiler calls function. |
| Program counter | PC | Holds instruction address during program execution. | |

**Remark** For detailed descriptions of r1, r3 to r5, and r31 used by an assembler or C compiler, refer to the **CA850 (C Compiler Package) Assembly Language User's Manual**.

★ **(1) General-purpose registers (r0 to r31)**

Thirty-two general-purpose registers, r0 to r31, are provided. All these registers can be used for data variables or address variables.

However, care must be exercised as follows in using the r0 to r5, r30, and r31 registers.

**(a) r0, r30**

r0 and r30 are implicitly used by instructions.

r0 is a register that always holds 0, and is used for operations using 0 and offset 0 addressing. r30 is used as a base pointer when accessing memory using the SLD and SST instructions.

**(b) r1, r3 to r5, r31**

r1, r3 to r5, and r31 are implicitly used by the assembler and C compiler.

Before using these registers, therefore, their contents must be saved so that they are not lost. The contents must be restored to the registers after the registers have been used.

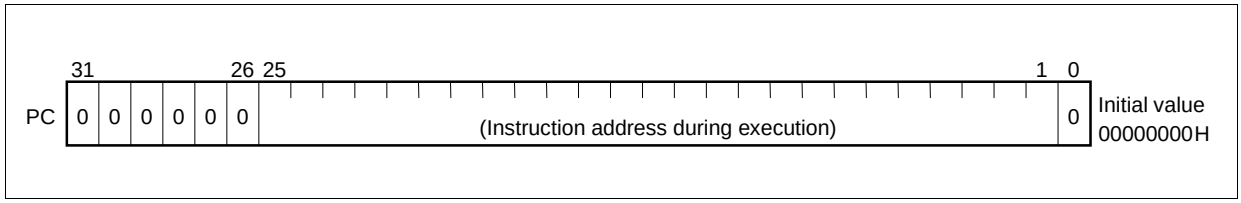
**(c) r2**

r2 is sometimes used by a real-time OS. When the real-time OS to be used is not using r2, r2 can be used as an address variable register or a data variable register.

**(2) Program counter (PC)**

This register holds an instruction address during program execution. The lower 26 bits of this register are valid, and bits 31 to 26 are reserved for future function expansion (fixed to 0). If a carry occurs from bit 25 to bit 26, it is ignored. Bit 0 is always fixed to 0 so that execution cannot branch to an odd address.

**Figure 2-2. Program Counter (PC)**



## 2.2 System Registers

The system registers control the CPU status and hold information on interrupts.

System registers can be read or written by specifying the relevant system register number from the following list using a system register load/store instruction (LDSR or STSR instruction).

**Table 2-2. System Register Numbers**

| | Register No. | Register Name | Operand Specifiability | |
|---|---|---|---|---|
| | | | LDSR Instruction | STSR Instruction |
| | 0 | Interrupt status saving register (EIPC) | ○ | ○ |
| | 1 | Interrupt status saving register (EIPSW) | ○ | ○ |
| | 2 | NMI status saving register (FEPC) | ○ | ○ |
| | 3 | NMI status saving register (FEPSW) | ○ | ○ |
| | 4 | Exception cause register (ECR) | × | ○ |
| | 5 | Program status word (PSW) | ○ | ○ |
| | 6 to 15 | (Numbers reserved for future function expansion (operation cannot be guaranteed if accessed)) | × | × |
| | 16 | CALLT caller status saving register (CTPC) | ○ | ○ |
| | 17 | CALLT caller status saving register (CTPSW) | ○ | ○ |
| ★ | 18 | Exception/debug trap status saving register (DBPC) | ○ | ○Note 1 |
| ★ | 19 | Exception/debug trap status saving register (DBPSW) | ○ | ○Note 1 |
| | 20 | CALLT base pointer (CTBP) | ○ | ○ |
| ★ | 21 | Debug interface register (DIR) | ○Note 1 | ○ |
| | 22 | Breakpoint control registers 0 and 1 (BPC0, BPC1)Note 2 | ○Note 1 | ○Note 1 |
| | 23 | Program ID register (ASID) | ○ | ○ |
| | 24 | Breakpoint address setting registers 0 and 1 (BPAV0, BPAV1)Note 2 | ○Note 1 | ○Note 1 |
| | 25 | Breakpoint address mask registers 0 and 1 (BPAM0, BPAM1)Note 2 | ○Note 1 | ○Note 1 |
| | 26 | Breakpoint data setting registers 0 and 1 (BPDV0, BPDV1)Note 2 | ○Note 1 | ○Note 1 |
| | 27 | Breakpoint data mask registers 0 and 1 (BPDM0, BPDM1)Note 2 | ○Note 1 | ○Note 1 |
| | 28 to 31 | (Numbers reserved for future function expansion (operation cannot be guaranteed if accessed)) | × | × |

★ **Notes 1.** These registers can be accessed only in the debug mode of type A and B products. Accessing these registers in other product types is prohibited. If they are accessed, the operation is not guaranteed.

**2.** The actual register to be accessed is specified by the DIR.CS bit.

**Caution When returning using the RETI instruction after setting bit 0 of EIPC, FEPC, or CTPC to 1 using the LDSR instruction and servicing an interrupt, the value of bit 0 is ignored (because bit 0 of the PC is fixed to 0). Therefore, be sure to set an even number (bit 0 = 0) when setting a value to EIPC, FEPC, or CTPC.**

**Remark** ○: Accessible

×: Inaccessible

### 2.2.1 Interrupt status saving registers (EIPC, EIPSW)

Two interrupt status saving registers are provided: EIPC and EIPSW.

If a software exception or maskable interrupt occurs, the contents of the program counter (PC) are saved to EIPC, and the contents of the program status word (PSW) are saved to EIPSW (if a non-maskable interrupt (NMI) occurs, the contents are saved to the NMI status saving registers (FEPC, FEPSW)).

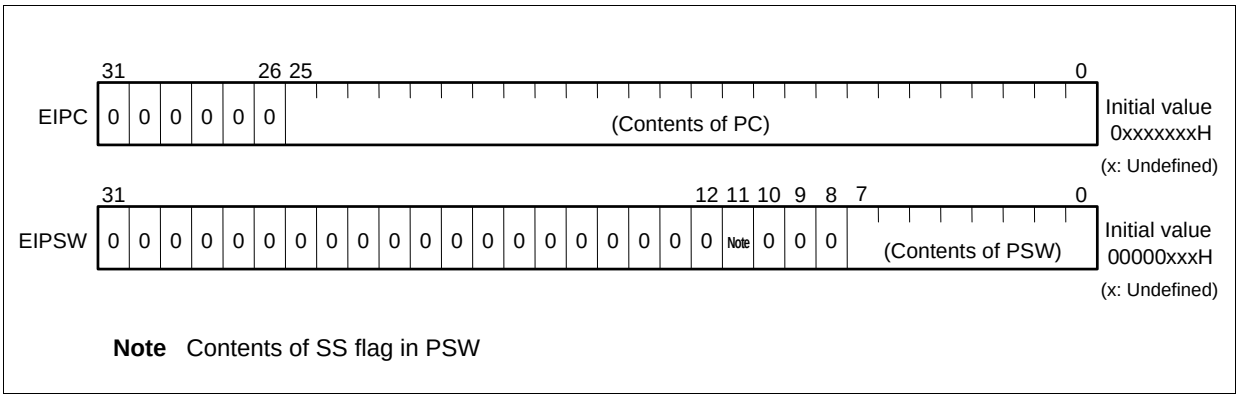
Except for some instructions, the address of the instruction next to the one being executed when the software exception or maskable interrupt occurs is saved to EIPC (see **Table 6-1 Interrupt/Exception Codes**).

The current value of the PSW is saved to EIPSW.

Because only one pair of interrupt status saving registers is provided, the contents of these registers must be saved by program when multiple interrupt servicing is enabled.

Bits 31 to 26 of EIPC and bits 31 to 12 and 10 to 8 of EIPSW are reserved for future function expansion (fixed to 0).

**Figure 2-3. Interrupt Status Saving Registers (EIPC, EIPSW)**




**Note** Contents of SS flag in PSW

### 2.2.2 NMI status saving registers (FEPC, FEPSW)

Two NMI status saving registers are provided: FEPC and FEPSW.

If a non-maskable interrupt (NMI) occurs, the contents of the program counter (PC) are saved to FEPC, and the contents of the program status word (PSW) are saved to FEPSW.

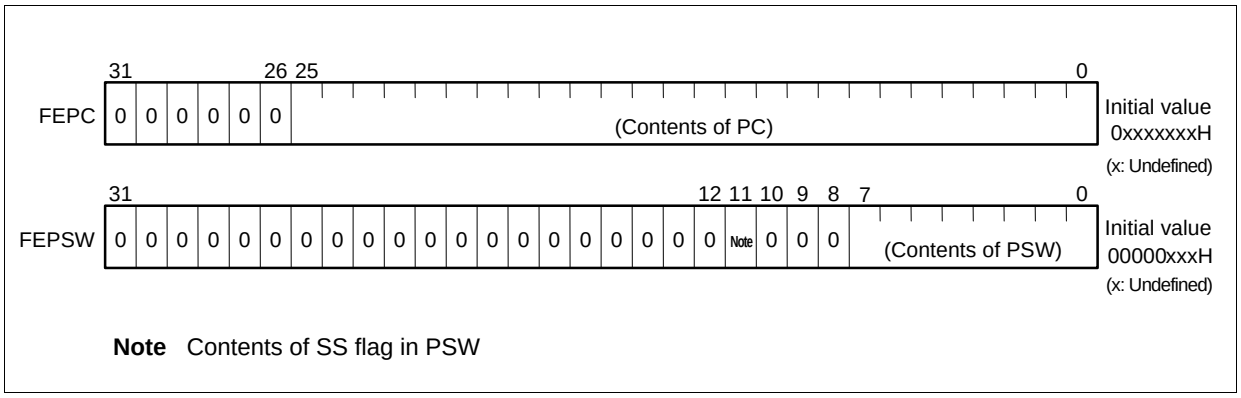
Except for some instructions, the address of the instruction next to the one being executed when the NMI occurs is saved to FEPC (see **Table 6-1 Interrupt/Exception Codes**).

The current value of the PSW is saved to FEPSW.

Because only one pair of NMI status saving registers is provided, the contents of these registers must be saved by program when multiple interrupt servicing is enabled.

Bits 31 to 26 of FEPC and bits 31 to 12 and 10 to 8 of FEPSW are reserved for future function expansion (fixed to 0).

**Figure 2-4. NMI Status Saving Registers (FEPC, FEPSW)**

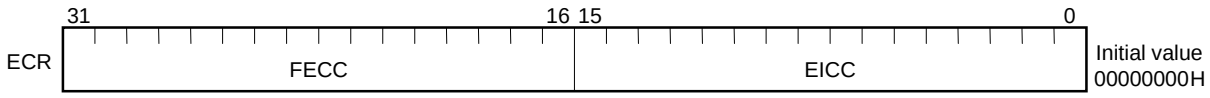



**Note** Contents of SS flag in PSW

### 2.2.3 Exception cause register (ECR)

The exception cause register (ECR) holds the cause information when an exception or interrupt occurs. The ECR holds an exception code which identifies each interrupt source (see **Table 6-1 Interrupt/Exception Codes**). This is a read-only register, and therefore no data can be written to it by using the LDSR instruction.

**Figure 2-5. Exception Cause Register (ECR)**




| Bit Position | Bit Name | Function |
|---|---|---|
| 31 to 16 | FECC | Exception code of non-maskable interrupt (NMI) |
| 15 to 0 | EICC | Exception code of exception or maskable interrupt |

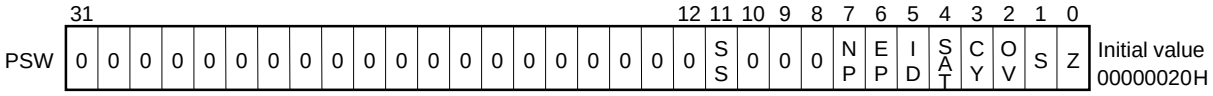
### 2.2.4 Program status word (PSW)

The program status word (PSW) is a collection of flags that indicate the status of the program (result of instruction execution) and the status of the CPU.

If the contents of the bits in this register are modified by the LDSR instruction, the PSW will assume the new value immediately after the LDSR instruction has been executed. Setting the ID flag to 1, however, will disable interrupt requests even while the LDSR instruction is being executed.

Bits 31 to 12 and 10 to 8 are reserved for future function expansion (fixed to 0).

**Figure 2-6. Program Status Word (PSW) (1/2)**




| Bit Position | Flag Name | Function |
|---|---|---|
| 11 | SS[Note] | Operates with single-step execution when this flag is set to 1 (debug trap occurs each time instruction is executed).<br>This flag is cleared to 0 when branching to the interrupt servicing routine.<br>When the SE bit of the DIR register is 0, this flag is not set (fixed to 0). |
| 7 | NP | Indicates that non-maskable interrupt (NMI) servicing is in progress. This flag is set to 1 when an NMI request is acknowledged, and multiple interrupt servicing is disabled.<br>0: NMI servicing is not in progress<br>1: NMI servicing is in progress |
| 6 | EP | Indicates that exception processing is in progress. This flag is set to 1 when an exception occurs. Even when this bit is set, interrupt requests can be acknowledged.<br>0: Exception processing is not in progress<br>1: Exception processing is in progress |
| 5 | ID | Indicates whether a maskable interrupt request can be acknowledged.<br>0: Interrupts enabled (EI)<br>1: Interrupts disabled (DI) |
| 4 | SAT[Note] | Indicates that an overflow has occurred in a saturated operation and the result is saturated. This is a cumulative flag. When the result is saturated, the flag is set to 1 and is not cleared to 0 even if the next result is not saturated. To clear this flag to 0, use the LDSR instruction. This flag is neither set to 1 nor cleared to 0 by execution of an arithmetic operation instruction.<br>0: Not saturated<br>1: Saturated |
| 3 | CY | Indicates whether a carry or borrow occurred as a result of the operation.<br>0: Carry or borrow did not occur<br>1: Carry or borrow occurred |
| 2 | OV[Note] | Indicates whether overflow occurred as a result of the operation.<br>0: Overflow did not occur<br>1: Overflow occurred |

★ **Note** Can only be used in type A or B products. Cannot be used in other product types.

**Figure 2-6. Program Status Word (PSW) (2/2)**

| Bit Position | Flag Name | Function |
|---|---|---|
| 1 | S[Note] | Indicates whether the result of the operation is negative.<br>0: Result is positive or zero<br>1: Result is negative |
| 0 | Z | Indicates whether the result of the operation is zero.<br>0: Result is not zero<br>1: Result is zero |

**Note** In the case of saturate instructions, the SAT, S, and OV flags will be set according to the result of the operation as shown in the table below. Note that the SAT flag is set to 1 only when the OV flag has been set to 1 during a saturated operation.

| Status of Operation Result | Status of Flag | | | Operation Result of Saturation Processing |
|---|---|---|---|---|
| | SAT | OV | S | |
| Maximum positive value is exceeded | 1 | 1 | 0 | 7FFFFFFFH |
| Maximum negative value is exceeded | 1 | 1 | 1 | 80000000H |
| Positive (Not exceeding maximum value) | Holds the value before operation | 0 | 0 | Operation result |
| Negative (Not exceeding maximum value) | | | 1 | |

### 2.2.5 CALLT caller status saving registers (CTPC, CTPSW)

Two CALLT caller status saving registers are provided: CTPC and CTPSW.

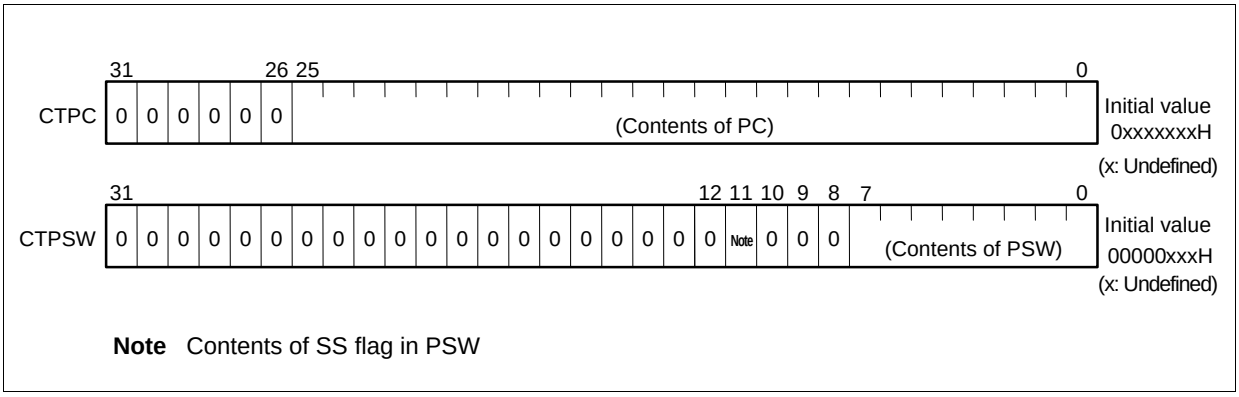
If a CALLT instruction is executed, the contents of the program counter (PC) are saved to CTPC, and the contents of the program status word (PSW) are saved to CTPSW.

The contents saved to CTPC are the address of the instruction next to the CALLT instruction.

The current value of the PSW is saved to CTPSW.

Bits 31 to 26 of CTPC and bits 31 to 12 and 10 to 8 of CTPSW are reserved for future function expansion (fixed to 0).

**Figure 2-7. CALLT Caller Status Saving Registers (CTPC, CTPSW)**




**Note** Contents of SS flag in PSW

### 2.2.6 Exception/debug trap status saving registers (DBPC, DBPSW)

Two exception/debug trap status saving registers are provided: DBPC and DBPSW.

★ When an exception trap, debug trap[Note], or debug break occurs or during a single-step operation, the contents of the program counter (PC) are saved to DBPC, and the contents of the program status word (PSW) are saved to DBPSW.

The contents to be saved to DBPC are as follows.

★ **Table 2-3. Contents to Be Saved to DBPC**

| Cause for Saving | | Contents Saved to DBPC |
|---|---|---|
| Occurrence of exception trap | | Address of the instruction next to the instruction that caused an exception trap |
| Occurrence of debug trap | | Address of the instruction next to the instruction that caused a debug trap |
| Occurrence of debug break | Execution trap | Address of the instruction that caused a break |
| | Misalign access exception | |
| | Alignment error exception | |
| | Access trap | Address of the instruction next to the instruction that caused a break |
| Single-step operation execution | | Address of the instruction to be executed next (instruction executed when restoring from the debug monitor routine) |

**Remark** For details of causes for saving, refer to **CHAPTER 9 SHIFTING TO DEBUG MODE**.

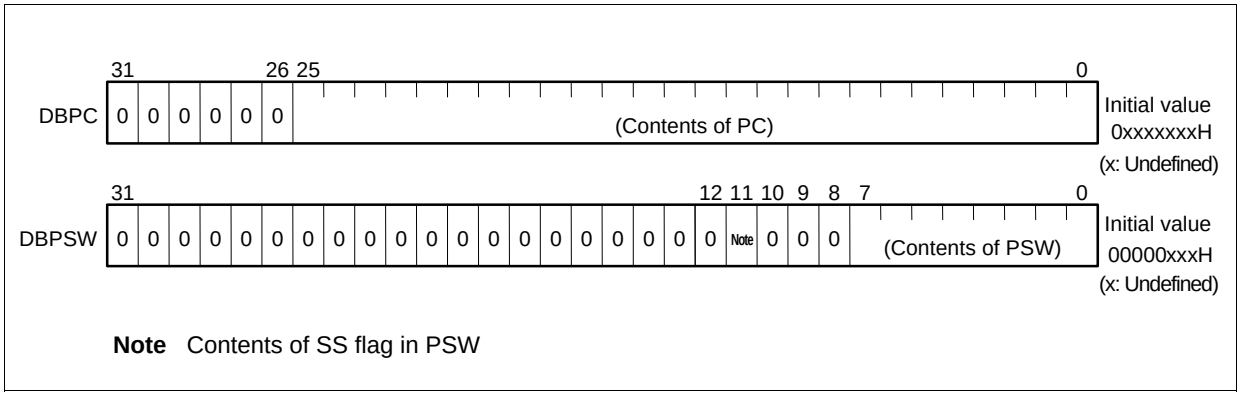
The current value of the PSW is saved to DBPSW.

★ Reading from this register is enabled only in debug mode (DIR.DM bit = 1) (writing to this register is always enabled). If this register is read in user mode (DM bit = 0), an undefined value is read.

Bits 31 to 26 of DBPC and bits 31 to 12 and 10 to 8 of DBPSW are reserved for future function expansion (fixed to 0).

★ **Note** Type C products do not support a debug trap.

**Figure 2-8. Exception/Debug Trap Status Saving Registers (DBPC, DBPSW)**



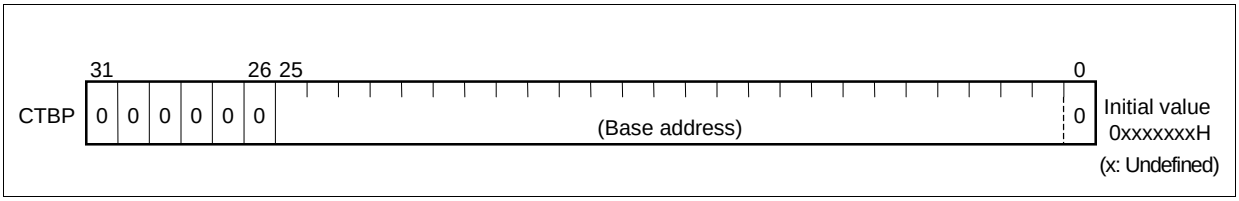

**Note** Contents of SS flag in PSW

### 2.2.7 CALLT base pointer (CTBP)

The CALLT base pointer (CTBP) is used to specify a table address and to generate a target address (bit 0 is fixed to 0).

Bits 31 to 26 are reserved for future function expansion (fixed to 0).

**Figure 2-9. CALLT Base Pointer (CTBP)**



### 2.2.8 Debug interface register (DIR)

The debug interface register (DIR) controls the debug function and indicates the debug function status.

The values of the bits in this register can be changed by using the LDSR instruction. Changed values become valid immediately after the execution of this instruction is complete.

This register can only be written in the debug mode (DM bit = 1) (except for bits 3 and 1) but can always be read.

Bits 14 to 8, 6 to 4, 2, and 1 are undefined in the user mode (DM bit = 0).

Bits 31 to 15 and 7 are reserved for future function expansion (fixed to 0).

★ **Caution Use of the debug interface register (DIR) is possible only in type A and B products, not in other product types.**

★ **Figure 2-10. Debug Interface Register (DIR) (1/3)**

| | 31 | | | | | | | | | | | | | | | | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DIR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SQ | RE | CS | CE | MA | AE | SE | 0 | IN | T1 | T0 | CM | MT | AT | DM | Initial value 00000040H |

| Bit Position | Bit Name | Function |
|---|---|---|
| ★ 14 | SQ Notes 1, 2 | Sets sequential break mode (sets a break if a break occurs for channel 0 and channel 1 in that order).<br>0: Normal break mode<br>1: Sequential break mode |
| ★ 13 | RE Notes 1, 2 | Sets range break mode (sets a break only when a break occurs for channels 0 and 1 simultaneously).<br>0: Normal break mode<br>1: Range break mode |
| 12 | CS Note 2 | Sets break register bank.<br>0: Select bank 0 register (channel 0 control register)<br>1: Select bank 1 register (channel 1 control register) |
| 11 | CE | Enables/disables COMBO interrupt.<br>0: COMBO interrupt disabled<br>1: COMBO interrupt enabled |
| 10 | MA | Enables/disables misalign access exception detection.<br>0: Misalign access exception detection disabled<br>1: Misalign access exception detection enabled |
| 9 | AE | Enables/disables alignment error exception detection.<br>0: Alignment error exception detection disabled<br>1: Alignment error exception detection enabled |

★ **Notes 1.** Always set either the SQ or RE bit to 1 or clear both bits to 0. If both bits are set to 1, the operation cannot be guaranteed.

★ **2.** While the IN bit is set to 1, writing to the SQ, RE, and CS bits is disabled. When the IN bit is set to 1, each bit is automatically cleared to 0.

**Figure 2-10. Debug Interface Register (DIR) (2/3)**

| Bit Position | Bit Name | Function |
|---|---|---|
| 8 | SE | Enables/disables writing to SS flag of PSW.<br>0: Writing to SS flag disabled (SS flag is fixed to 0)<br>1: Writing to SS flag enabled |
| 6 | IN[Note 1] | Set to 1 by debug function reset.<br>Be sure to clear this bit to 0 after reset (while this bit is set to 1, writing to SQ, RE, and CS bits is disabled, and T1 and T0 bits do not operate). |
| 5 | T1[Notes 1, 2] | Set to 1 by channel 1 break generation.<br>Cleared to 0 by setting 0[Note 4]. |
| 4 | T0[Notes 1, 2] | Set to 1 by channel 0 break generation.<br>Cleared to 0 by setting 0[Note 4]. |
| 3 | CM[Note 3] | Set to 1 by shift to COMBO interrupt routine or debug monitor routine 2.<br>Writing to this bit is disabled. |
| 2 | MT[Note 1] | Set to 1 by detection of misalign access exception.<br>Cleared to 0 by setting 0[Note 4]. |
| 1 | AT[Note 1] | Set to 1 by detection of alignment error exception.<br>Cleared to 0 by setting 0[Note 4]. |
| 0 | DM[Note 3] | Set to 1 when debug mode is entered. Cleared to 0 when user mode is entered.<br>Writing to this bit is disabled. |

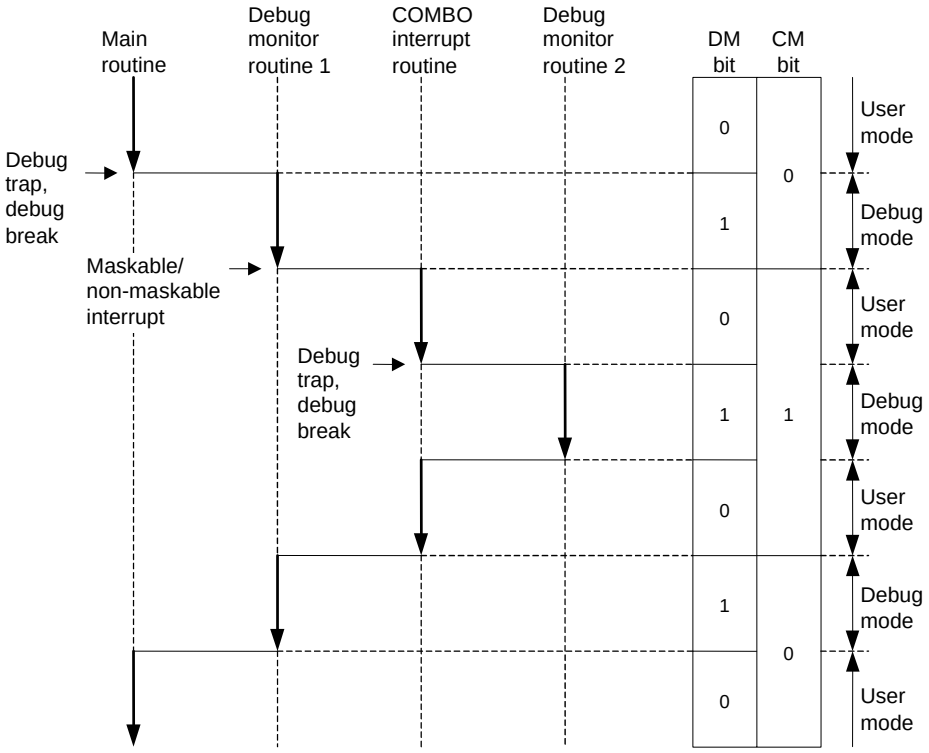
**Remark** The explanations of the **Notes** are given on the next page.

**Figure 2-10. Debug Interface Register (DIR) (3/3)**

**Notes 1.** The IN, T1, T0, MT, and AT bits are not automatically cleared to 0 after being set to 1 (they are cleared to 0 only by the LDSR instruction).

**2.** While the IN bit is set to 1, the T1 and T0 bits do not operate (even if a break occurs, these bits are not set to 1), and are automatically cleared to 0.

**3.** The DM and CM bits change as follows.




**Notes 4.** The T1, T0, MT, and AT bits cannot be arbitrarily set to 1 by a user program.

### 2.2.9 Breakpoint control registers 0 and 1 (BPC0, BPC1)

Breakpoint control registers 0 and 1 (BPC0, BPC1) indicate the control and status of the debug function.

One or other of these registers is enabled by the setting of the DIR.CS bit.

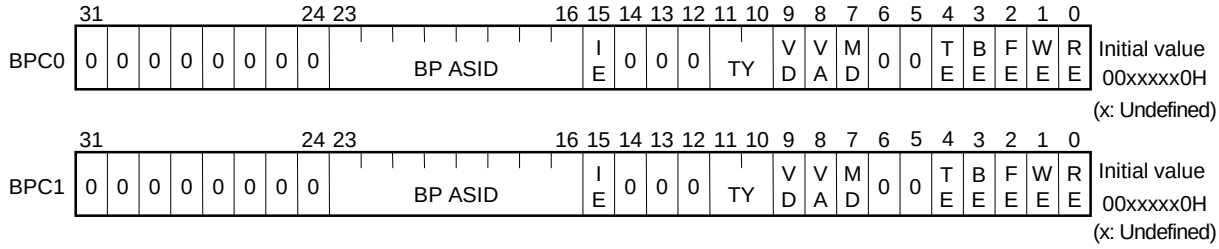
The values of the bits in these registers can be changed by using the LDSR instruction. Changed values become valid immediately after execution of this instruction. (If the FE bit is set to 1, the timing at which the changed values become valid is delayed, but the changes are definitely reflected after the DBRET instruction is executed.)

These registers can only be set in the debug mode (DIR.DM bit = 1). In the user mode (DM bit = 0), bit 0 = 0, and bits 23 to 15, 11 to 7, and 4 to 1 are undefined.

Bits 31 to 24, 14 to 12, 6, and 5 are reserved for future function expansion (fixed to 0).

★ **Caution Use of breakpoint control registers 0 and 1 (BPC0, BPC1) is possible only in type A and B products, not in other product types.**

**Figure 2-11. Breakpoint Control Registers 0 and 1 (BPC0, BPC1) (1/2)**




| Bit Position | Bit Name | Function |
|---|---|---|
| 23 to 16 | BP ASID | Sets the program ID that generates a break (valid only when IE bit = 1). |
| 15 | IE | Sets the comparison of the BP ASID bit and the program ID set in the ASID register.<br>0: Not compared<br>1: Compared |
| 11, 10 | TY | Sets the type of access for which a break is detected.<br>0,0: Access by all data types<br>0,1: Byte access (including bit manipulation)<br>1,0: Halfword access<br>1,1: Word access<br>Note that the contents set in this register are ignored in the case of an execution trap. |
| 9 | VD | Sets the match condition of the data comparator.<br>0: Break on a match<br>1: Break on a mismatch |
| 8 | VA | Sets the match condition of the address comparator.<br>0: Break on a match<br>1: Break on a mismatch |
| ★ 7 | MD | Sets the operation of the data comparator.<br>0: Break on match of data and condition.<br>1: Whether data matches (data comparator) is ignored regardless of the setting of the VD bit or BPDVx and BPDMx registers |

**Figure 2-11. Breakpoint Control Registers 0 and 1 (BPC0, BPC1) (2/2)**

| Bit Position | Bit Name | Function |
|---|---|---|
| 4 | TE[Note 1] | Enables/disables trigger output.<br>0: Trigger output disabled<br>1: Trigger output enabled (output corresponding trigger before break occurs in channel 0 or 1). |
| 3 | BE[Note 1] | Sets whether or not a break in channel 0 or 1 is reported to the CPU.<br>0: Not reported.<br>1: Reported (break). |
| ★ 2 | FE | Enables/disables break/trigger due to instruction execution address match.<br>0: Break/trigger disabled<br>1: Break/trigger enabled[Note 2] |
| ★ 1 | WE | Enables/disables break/trigger on data write.<br>0: Break/trigger disabled<br>1: Break/trigger enabled[Note 3] |
| ★ 0 | RE | Enables/disables break/trigger on data read.<br>0: Break/trigger disabled<br>1: Break/trigger enabled[Note 3] |

★ **Notes 1.** The TE and BE bits can be set only in type B products. In other product types, the TE and BE bits are fixed to 0 (however, even when the BE bit is fixed to 0, it reports a break to the CPU).

★ **2.** If the FE bit is set to 1, always clear the WE and RE bits to 0.

★ **3.** If the WE and RE bits are set to 1, always clear the FE bit to 0.

### 2.2.10 Program ID register (ASID)

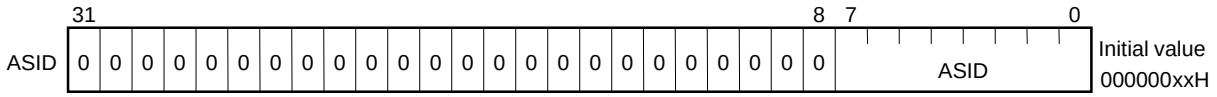
This register sets the ID of the program currently under execution.

★ The program ID is used when a shift to the debug mode is necessary only in cases such as when a specific program is being executed to download different programs to the RAM of the same address area. While the BPCn.IE bit is set to 1, the system does not shift to the debug mode if the program IDs set to the BPCn.BP ASID bit and the ASID register do not match; even if the break conditions match (n = 0, 1).

Bits 31 to 8 are reserved for future function expansion (fixed to 0).

★ **Caution Use of the program ID register (ASID) is possible only in the type A and B products, not in other product types.**

**Figure 2-12. Program ID Register (ASID)**




| Bit Position | Flag Name | Function |
|---|---|---|
| 7 to 0 | ASID | ID of program currently under execution |

### 2.2.11 Breakpoint address setting registers 0 and 1 (BPAV0, BPAV1)

These registers set the breakpoint addresses to be used by the address comparator.

One or other of these registers is enabled by the setting of the DIR.CS bit.

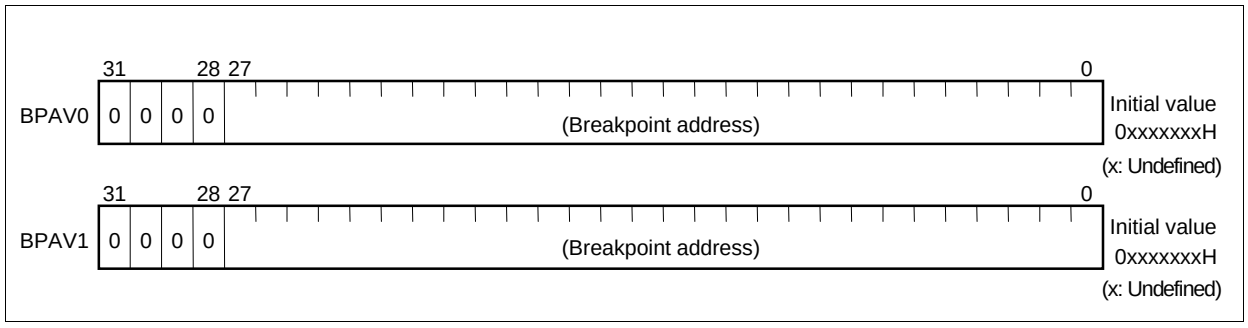
★ Writing to/reading from these registers is enabled only in the debug mode (DIR.DM bit = 1). If read in the user mode (DM bit = 0), an undefined value is read.

When these registers are not used, be sure to set each bit to 1.

Bits 31 to 28 are reserved for future function expansion (fixed to 0).

★ **Caution Use of breakpoint address setting registers 0 and 1 (BPAV0, BPAV1) is possible only in the type A and B products, not in other type products.**

**Figure 2-13. Breakpoint Address Setting Registers 0 and 1 (BPAV0, BPAV1)**




### 2.2.12 Breakpoint address mask registers 0 and 1 (BPAM0, BPAM1)

These registers set the bit mask for address comparison (masked by 1).

One or other of these registers is enabled by the setting of the DIR.CS bit.

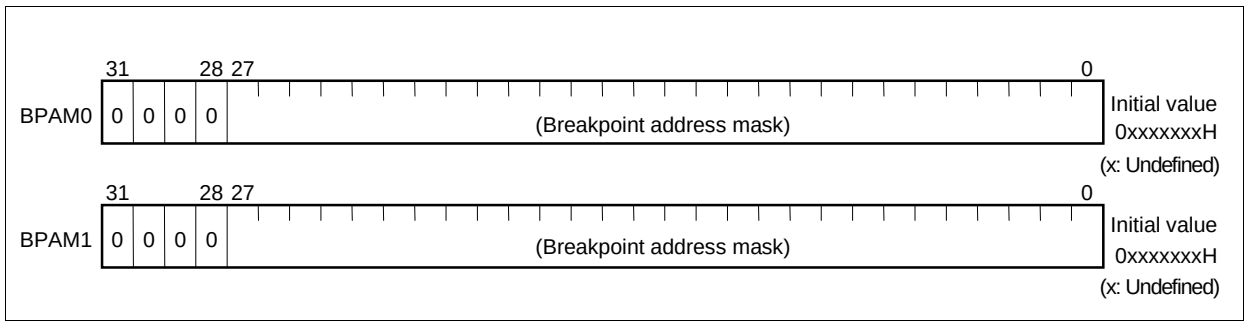
★ Writing to/reading from these registers is enabled only in the debug mode (DIR.DM bit = 1). If read in the user mode (DM bit = 0), an undefined value is read.

When these registers are not used, be sure to set each bit to 1.

Bits 31 to 28 are reserved for future function expansion (fixed to 0).

★ **Caution Use of breakpoint address mask registers 0 and 1 (BPAM0, BPAM1) is possible only in the type A and B products, not in other product types.**

**Figure 2-14. Breakpoint Address Mask Registers 0 and 1 (BPAM0, BPAM1)**



### 2.2.13 Breakpoint data setting registers 0 and 1 (BPDV0, BPDV1)

These registers set the breakpoint data to be used by the data comparator.

One or other of these registers is enabled by the setting of the DIR.CS bit.

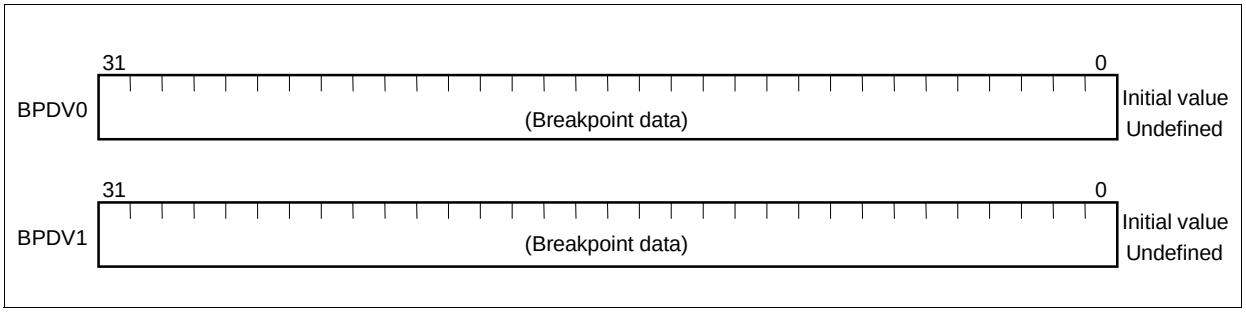
★ Writing to/reading from these registers is enabled only in the debug mode (DIR.DM bit = 1). If read in the user mode (DM bit = 0), an undefined value is read.

When these registers are not used, be sure to set each bit to 1.

★ **Caution Use of breakpoint data setting registers 0 and 1 (BPDV0, BPDV1) is possible only in the type A and B products, not in other product types.**

**Remark** Set the instruction code for 16-bit instructions aligned to the LSB. Set the instruction codes for 32-bit instructions in little endian format.

**Figure 2-15. Breakpoint Data Setting Registers 0 and 1 (BPDV0, BPDV1)**




### 2.2.14 Breakpoint data mask registers 0 and 1 (BPDM0, BPDM1)

These registers set the bit mask for data comparison (masked by 1).

One or other of these registers is enabled by the setting of the DIR.CS bit.

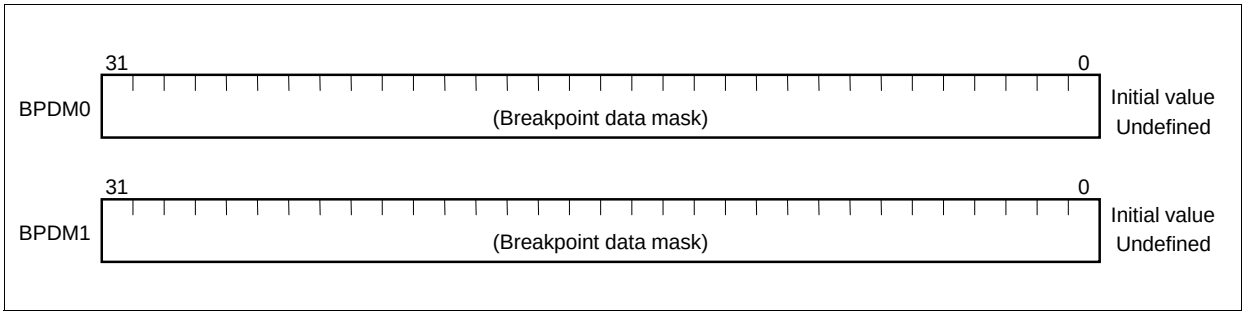
★ Writing to/reading from these registers is enabled only in the debug mode (DIR.DM bit = 1). If read in the user mode (DM bit = 0), an undefined value is read.

When these registers are not used, be sure to set each bit to 1.

★ When the data access type that detects breaks is set to the byte access (BPCn.TY bit = 0, 1), set bits 31 to 8 to 1, and if halfword access (TY bit = 0, 1), set bits 31 to 16 to 1 (n = 0, 1).

★ **Caution Use of breakpoint data mask registers 0 and 1 (BPDM0, BPDM1) is possible only in the type A and B products, not in other product types.**

**Figure 2-16. Breakpoint Data Mask Registers 0 and 1 (BPDM0, BPDM1)**



# CHAPTER 3 DATA TYPES

## 3.1 Data Format

The following data types are supported (see **3.2 Data Representation**).

- Integer (32, 16, 8 bits)
- Unsigned integer (32, 16, 8 bits)
- Bit

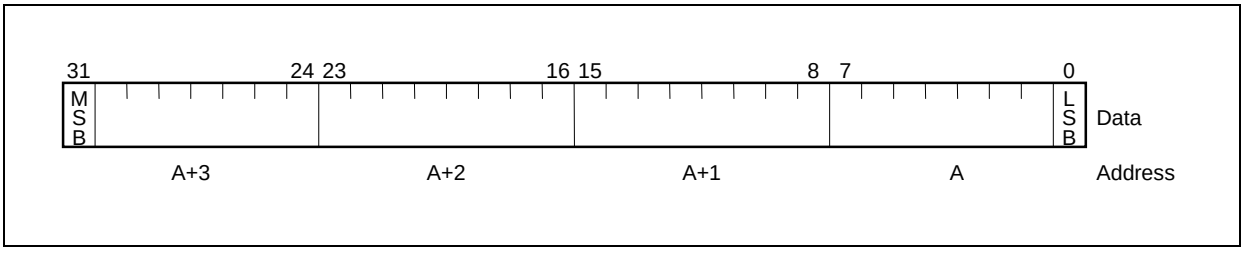
Three types of data lengths: word (32 bits), halfword (16 bits), and byte (8 bits) are supported. Byte 0 of any data is always the least significant byte (this is called little endian) and is shown at the rightmost position in figures throughout this manual.

The following paragraphs describe the data format where data of fixed length is in memory.

**(1) Word**

A word is 4-byte (32-bit) contiguous data that starts from any word boundary[Note]. Each bit is assigned a number from 0 to 31. The LSB (Least Significant Bit) is bit 0 and the MSB (Most Significant Bit) is bit 31. A word is specified by its address "A" (with the 2 lowest bits fixed to 0 when misalign access is disabled[Note]), and occupies 4 bytes, A, A+1, A+2, and A+3.

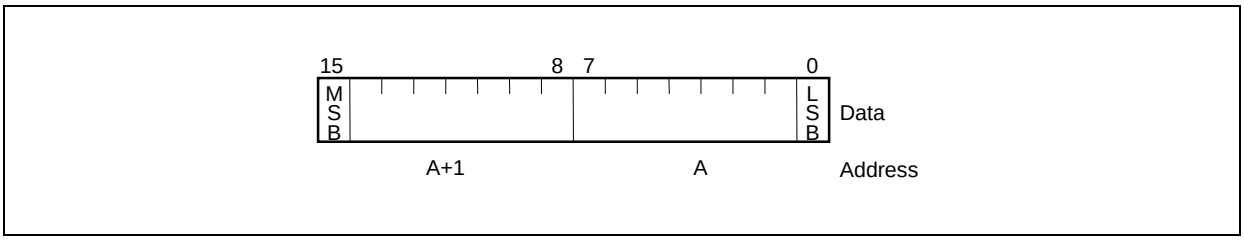
**Note** When misalign access is enabled, any byte boundary can be accessed whether access is in halfword or word units. See **3.3 Data Alignment**.



**(2) Halfword**

A halfword is 2-byte (16-bit) contiguous data that starts from any halfword boundary[Note]. Each bit is assigned a number from 0 to 15. The LSB is bit 0 and the MSB is bit 15. A halfword is specified by its address "A" (with the lowest bit fixed to 0[Note]), and occupies 2 bytes, A and A+1.

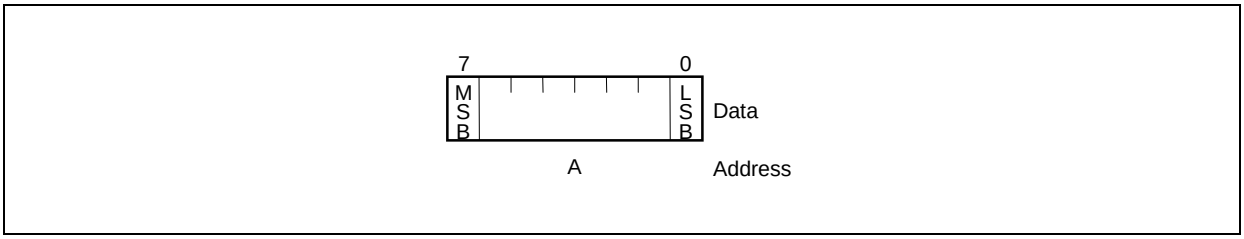
**Note** When misalign access is enabled, any byte boundary can be accessed whether access is in halfword or word units. See **3.3 Data Alignment**.




**(3) Byte**

A byte is 8-bit contiguous data that starts from any byte boundary[Note]. Each bit is assigned a number from 0 to 7. The LSB is bit 0 and the MSB is bit 7. A byte is specified by its address "A".

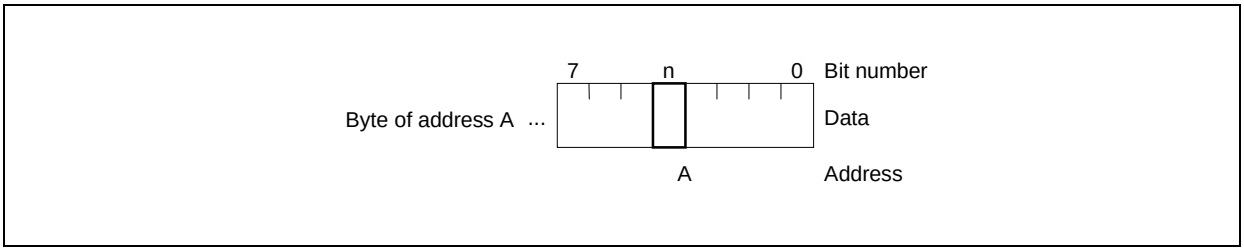
**Note** When misalign access is enabled, any byte boundary can be accessed whether access is in halfword or word units. See **3.3 Data Alignment**.




**(4) Bit**

A bit is 1-bit data at the nth bit position in 8-bit data that starts from any byte boundary[Note]. A bit is specified by its address "A" and bit number "n".

**Note** When misalign access is enabled, any byte boundary can be accessed whether access is in halfword or word units. See **3.3 Data Alignment**.



## 3.2 Data Representation

### 3.2.1 Integer

An integer is expressed as a binary number of 2's complement and is 32, 16, or 8 bits long. Regardless of its length, bit 0 of an integer is the least significant bit. The higher the bit number, the more significant the bit. Because 2's complement is used, the most significant bit is used as a sign bit.

The integer range of each data length is as follows.

- Word (32 bits): –2,147,483,648 to +2,147,483,647
- Halfword (16 bits): –32,768 to +32,767
- Byte (8 bits): –128 to +127

### 3.2.2 Unsigned integer

While an integer is data that can take either a positive or a negative value, an unsigned integer is an integer that is not negative. Like an integer, an unsigned integer is also expressed as 2's complement and is 32, 16, or 8 bits long. Regardless of its length, bit 0 of an unsigned integer is the least significant bit, and the higher the bit number, the more significant the bit. However, no sign bit is used.

The unsigned integer range of each data length is as follows.

- Word (32 bits): 0 to 4,294,967,295
- Halfword (16 bits): 0 to 65,535
- Byte (8 bits): 0 to 255

### 3.2.3 Bit

1-bit data that can take a value of 0 (cleared) or 1 (set) can be handled as bit data. Bit manipulation can be performed only on 1-byte data in the memory space in the following four ways.

- SET1
- CLR1
- NOT1
- TST1

## ★ 3.3 Data Alignment

Data must be aligned (boundary aligned) in accordance with the setting of misalign access enable/disable.

Misalign access indicates access to other than a halfword boundary (LSB of the address is 0) when the target data is in halfword format, and access to other than a word boundary (lower two bits of the address are 0) when the target data is in word format.

**Remark** The V850E1 CPU enables/disables misalign access in accordance with the IFIMAEN pin input level.

**(1) When misalign access is enabled**

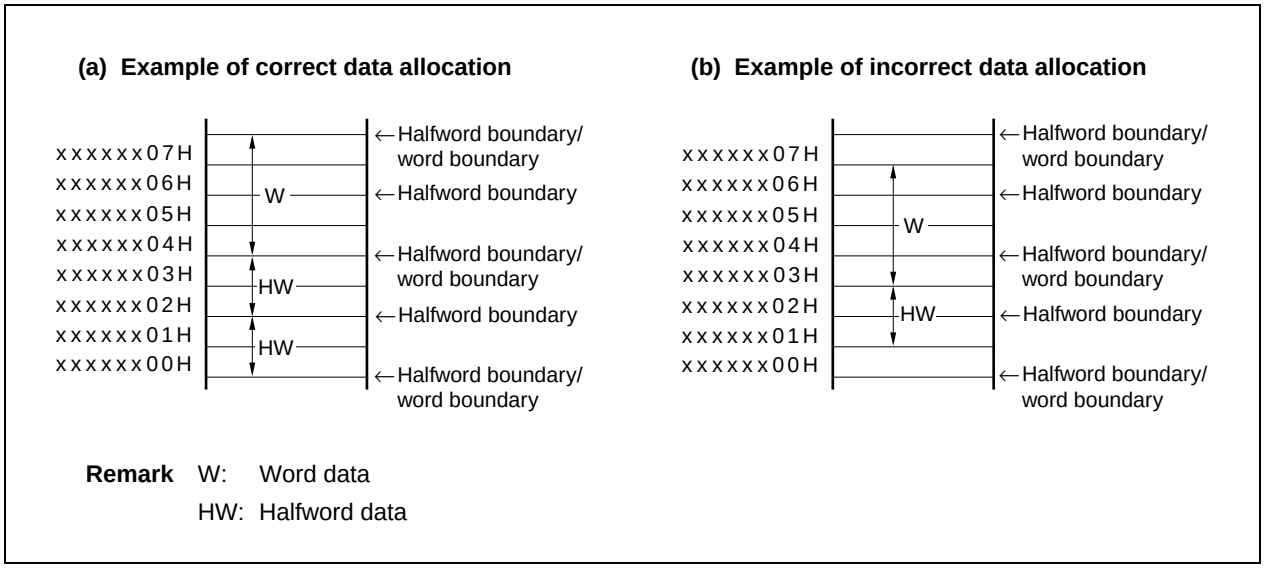
Regardless of the data format (byte, halfword, word), data can be allocated to all addresses.

However, when halfword or word data is used, at least one bus cycle occurs and the bus efficiency is degraded if data is not aligned.

**(2) When misalign access is disabled**

The lower bit(s) of the address (LSB if halfword data is used, lower two bits if word data is used) are masked by 0 and accessed. Therefore, if the target data is not aligned correctly, data may be lost or be rounded off. Therefore, allocate the halfword data to be processed from a halfword boundary, and the word data to be processed from a word boundary.

**Figure 3-1. Example of Data Allocation When Misalign Access Is Disabled**




**Remark** W: Word data
HW: Halfword data

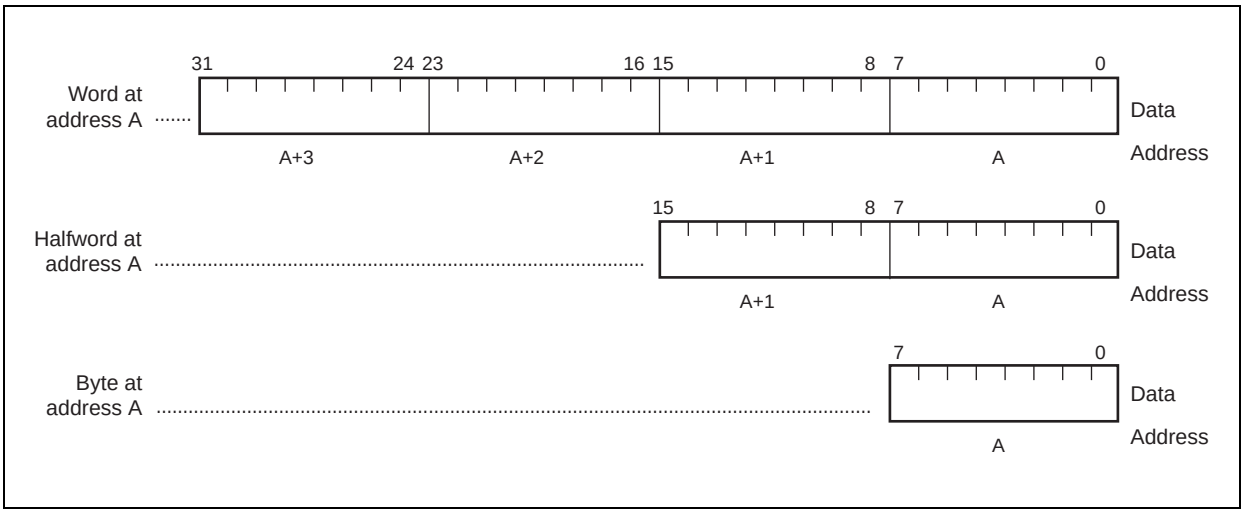
# CHAPTER 4 ADDRESS SPACE

The V850E1 CPU supports a 4 GB linear address space. Both memory and I/O are mapped to this address space (memory-mapped I/O). The V850E1 CPU (NB85E) outputs 32-bit addresses to the memory and I/O. The maximum address is $2^{32}$–1.

Byte data allocated to each address is defined with bit 0 as the LSB and bit 7 as the MSB. With regards to multiple-byte data, the byte with the lowest address value is defined to be the LSB and the byte with the highest address value is defined to be the MSB (little endian).

Data consisting of 2 bytes is called a halfword, and 4-byte data is called a word.

In this user's manual, data consisting of 2 or more bytes is illustrated as shown below, with the lower address shown on the right and the higher address on the left.



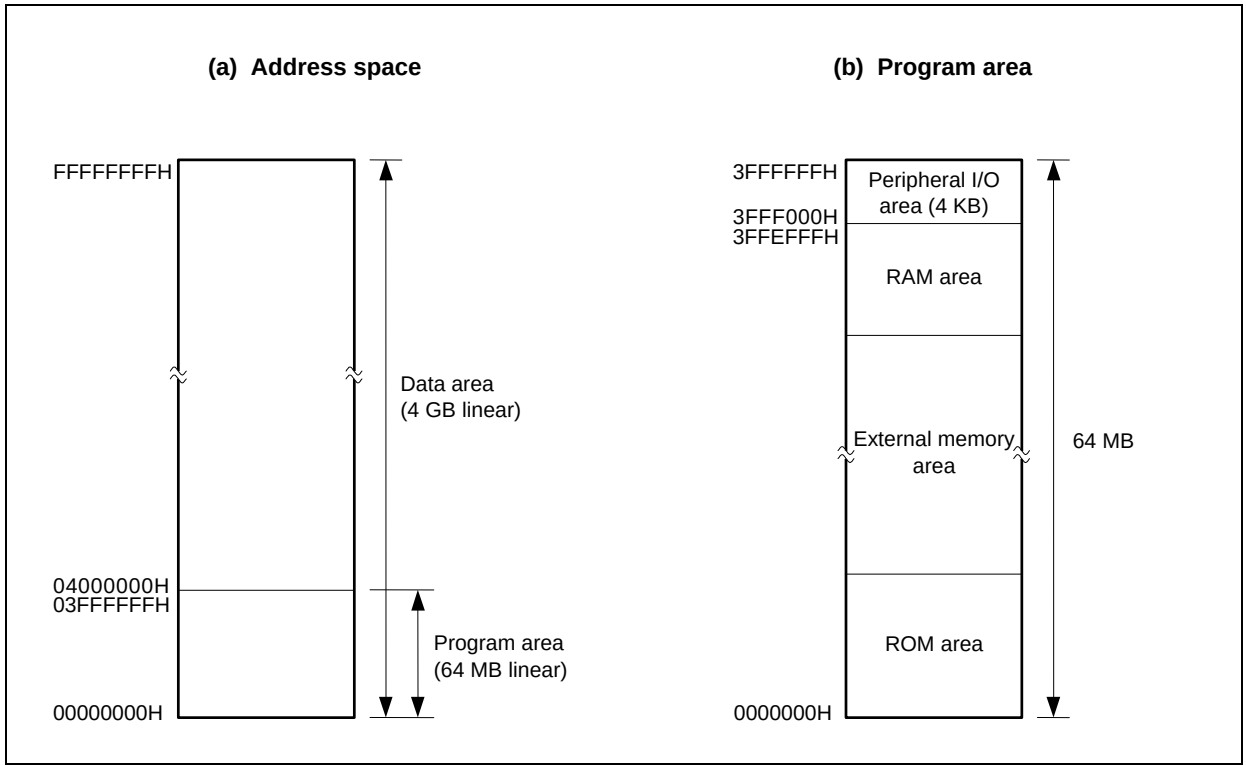
## 4.1 Memory Map

The V850E1 CPU employs a 32-bit architecture and supports a linear address space (data area) of up to 4 GB for operand addressing (data access).

It supports a linear address space (program area) of up to 64 MB for instruction addressing.

Figure 4-1 shows the memory map.

**Figure 4-1. Memory Map**



## 4.2 Addressing Mode

The CPU generates two types of addresses: instruction addresses used for instruction fetch and branch operations; and operand addresses used for data access.

### 4.2.1 Instruction address

An instruction address is determined by the contents of the program counter (PC), and is automatically incremented (+2) according to the number of bytes of an instruction to be fetched each time an instruction is executed. When a branch instruction is executed, the branch destination address is loaded into the PC using one of the following two addressing modes.

**(1) Relative addressing (PC relative)**

The signed 9- or 22-bit data of an instruction code (displacement: disp×) is added to the value of the program counter (PC). At this time, the displacement is treated as 2's complement data with bits 8 and 21 serving as sign bits (S).

This addressing is used for the JARL disp22, reg2, JR disp22, and Bcond disp9 instructions.

**Figure 4-2. Relative Addressing (1/2)**

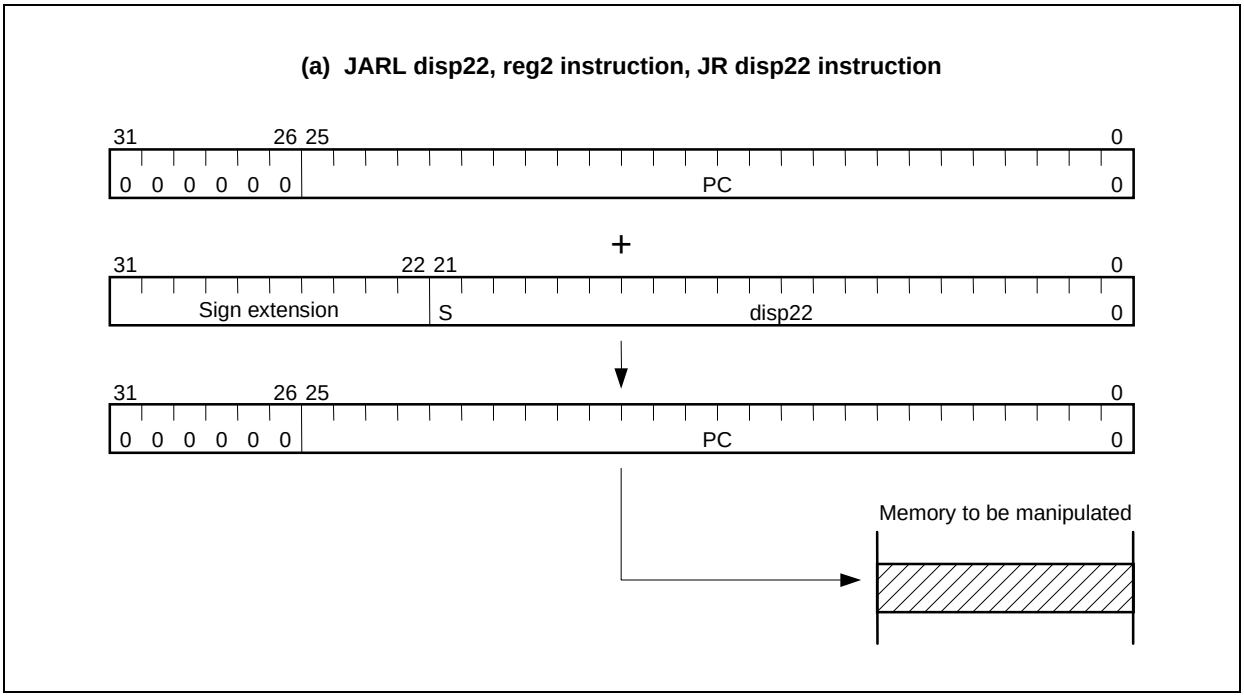
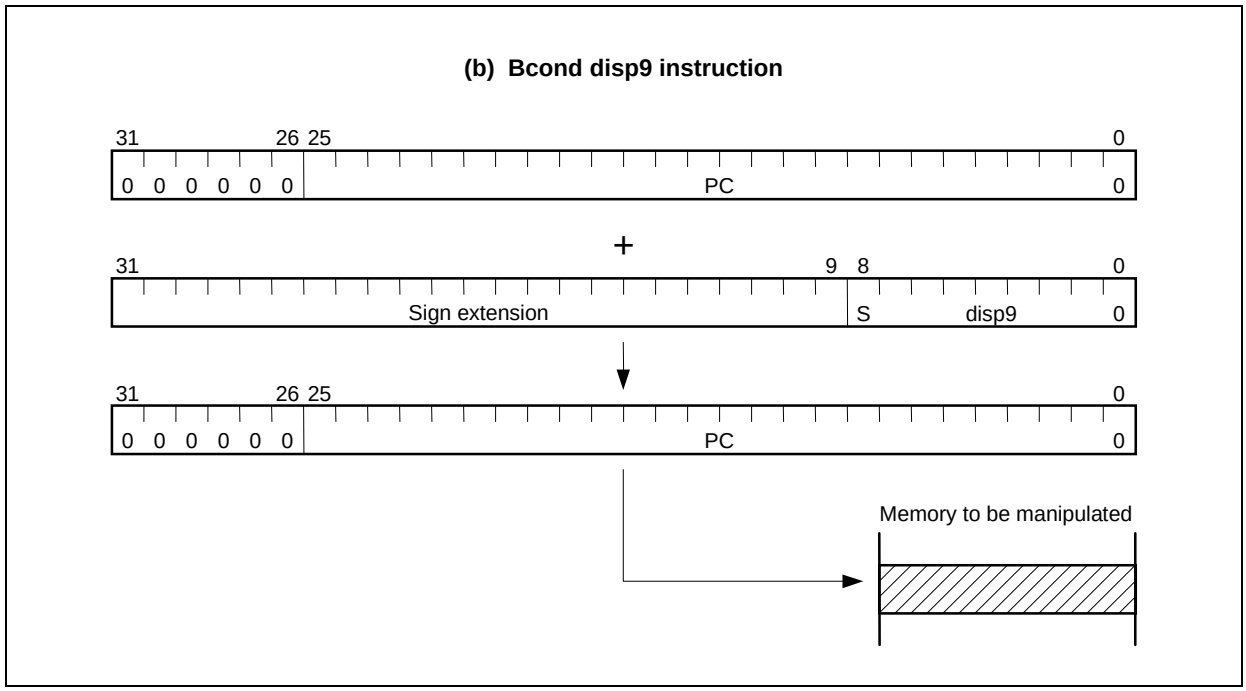


**Figure 4-2. Relative Addressing (2/2)**

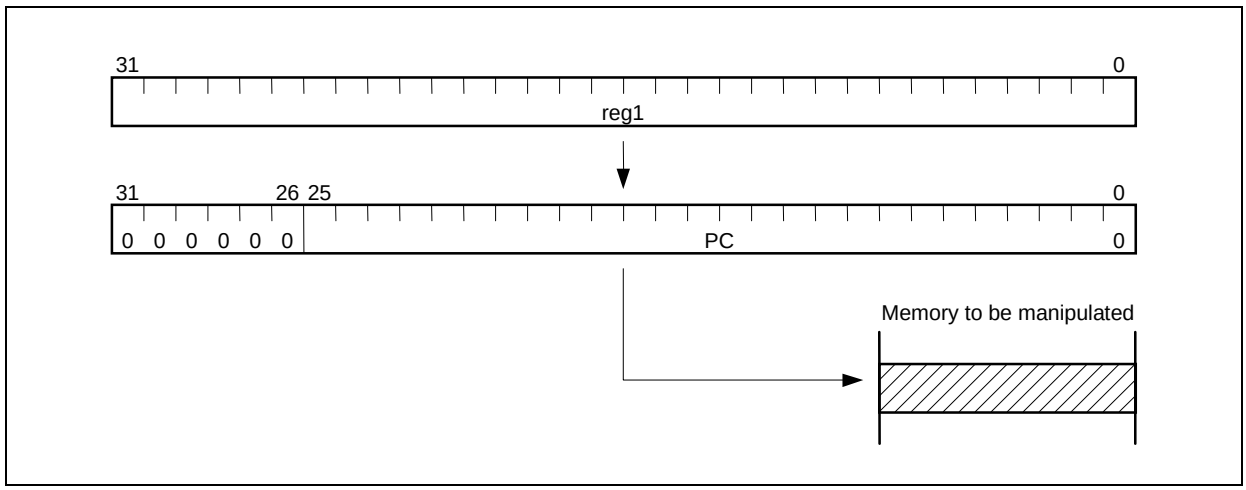



**(2) Register addressing (register indirect)**

The contents of a general-purpose register (reg1) specified by an instruction are transferred to the program counter (PC).

This addressing is used for the JMP [reg1] instruction.

**Figure 4-3. Register Addressing (JMP [reg1] Instruction)**



### 4.2.2 Operand address

When an instruction is executed, the register or memory area to be accessed is specified in one of the following four addressing modes.

**(1) Register addressing**

The general-purpose register or system register specified in the general-purpose register specification field is accessed as operand.

This addressing mode applies to instructions using the operand format reg1, reg2, reg3, or regID.

**(2) Immediate addressing**

The 5-bit or 16-bit data for manipulation is contained in the instruction code.

This addressing mode applies to instructions using the operand format imm5, imm16, vector, or cccc.

**Remark** vector: Operand that is 5-bit immediate data for specifying a trap vector (00H to 1FH), and is used in the TRAP instruction.

cccc: Operand consisting of 4-bit data used in the CMOV, SASF, and SETF instructions to specify a condition code. Assigned as part of the instruction code as 5-bit immediate data by appending 1-bit 0 above the highest bit.

**(3) Based addressing**

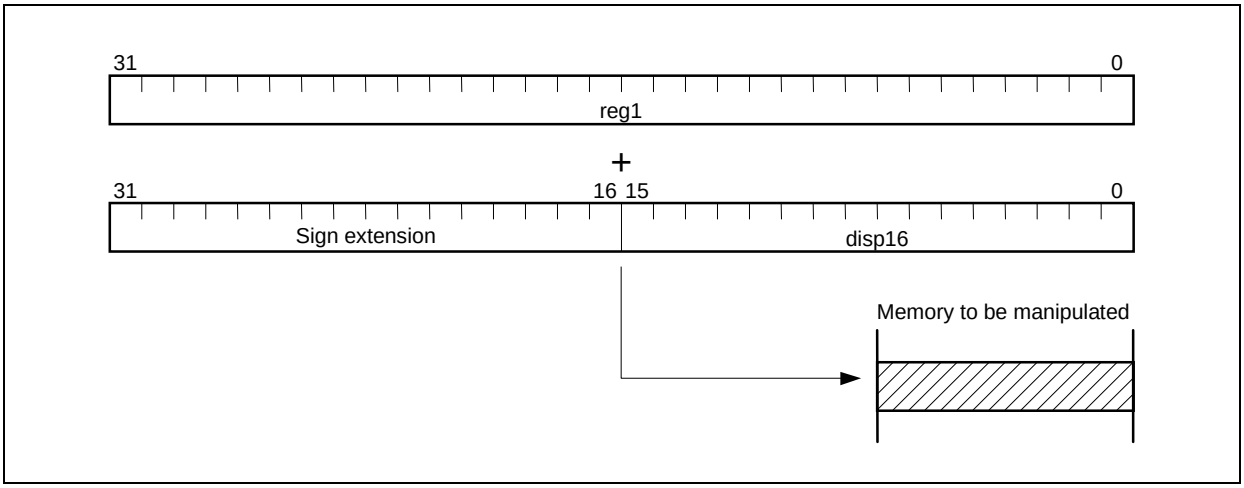
The following two types of based addressing are supported.

**(a) Type 1**

The address of the data memory location to be accessed is determined by adding the value in the specified general-purpose register (reg1) to the 16-bit displacement value (disp16) contained in the instruction code.

This addressing mode applies to instructions using the operand format disp16 [reg1].

**Figure 4-4. Based Addressing (Type 1)**

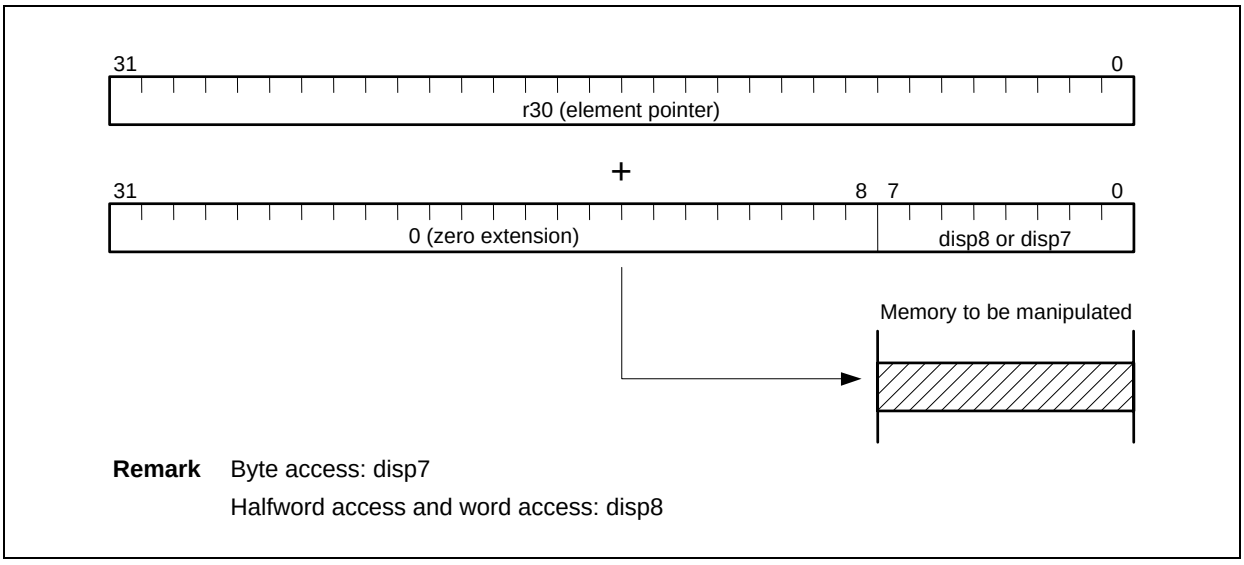


**(b) Type 2**

The address of the data memory location to be accessed is determined by adding the value in the element pointer (r30) to the 7- or 8-bit displacement value (disp7, disp8).

This addressing mode applies to SLD and SST instructions.

**Figure 4-5. Based Addressing (Type 2)**



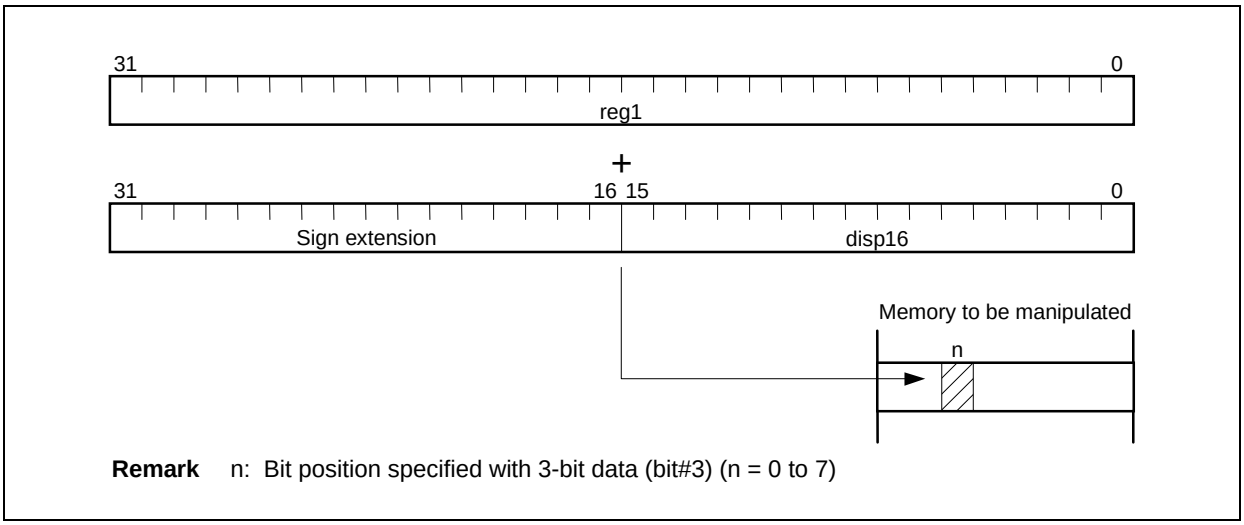

**Remark** Byte access: disp7
Halfword access and word access: disp8

**(4) Bit addressing**

This addressing is used to access 1 bit (specified with bit#3 of 3-bit data) among 1 byte of the memory space to be manipulated by using an operand address which is the sum of the contents of a general-purpose register (reg1) and a 16-bit displacement (disp16) sign-extended to a word length.

This addressing mode applies only to bit manipulation instructions.

**Figure 4-6. Bit Addressing**




**Remark** n: Bit position specified with 3-bit data (bit#3) (n = 0 to 7)

# CHAPTER 5 INSTRUCTIONS

## 5.1 Instruction Format

There are two types of instruction formats: 16-bit and 32-bit. The 16-bit format instructions include binary operation, control, and conditional branch instructions, and the 32-bit format instructions include load/store, jump, and instructions that handle 16-bit immediate data.

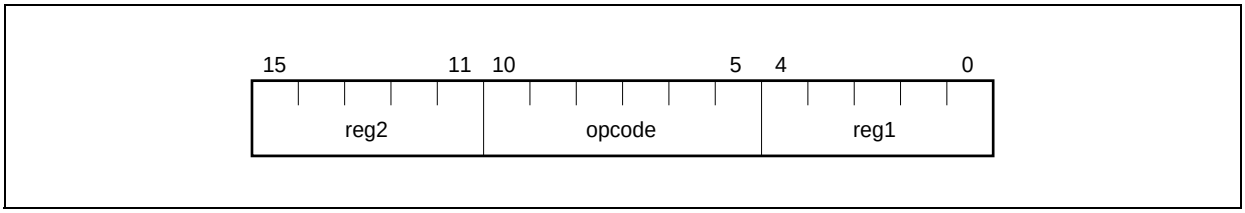
An instruction is actually stored in memory as follows.

- Lower bytes of instruction (including bit 0) → lower address
- Higher bytes of instruction (including bit 15 or bit 31) → higher address

**Caution Some instructions have an unused field (RFU). This field is reserved for future expansion and must be fixed to 0.**

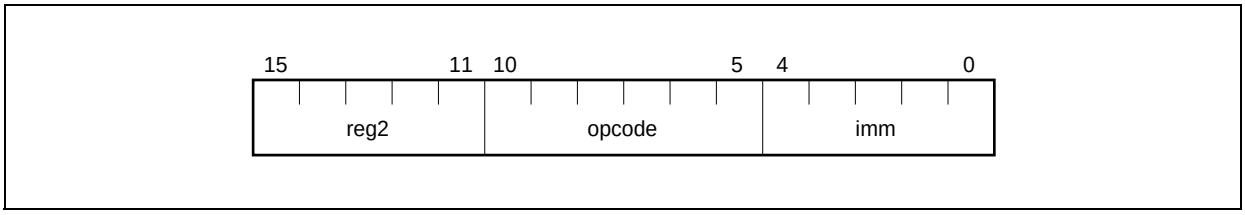
**(1) reg-reg instruction (Format I)**

A 16-bit instruction format having a 6-bit opcode field and two general-purpose register specification fields.



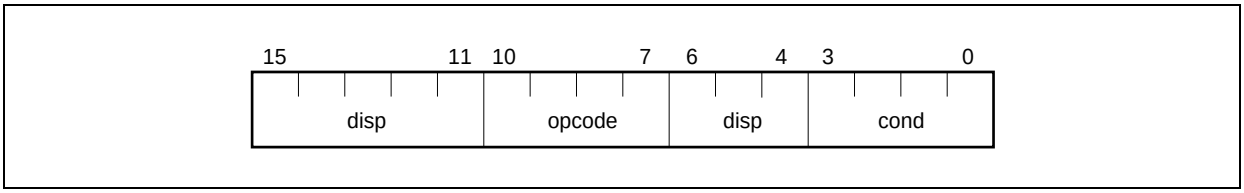

**(2) imm-reg instruction (Format II)**

A 16-bit instruction format having a 6-bit opcode field, 5-bit immediate field, and a general-purpose register specification field.



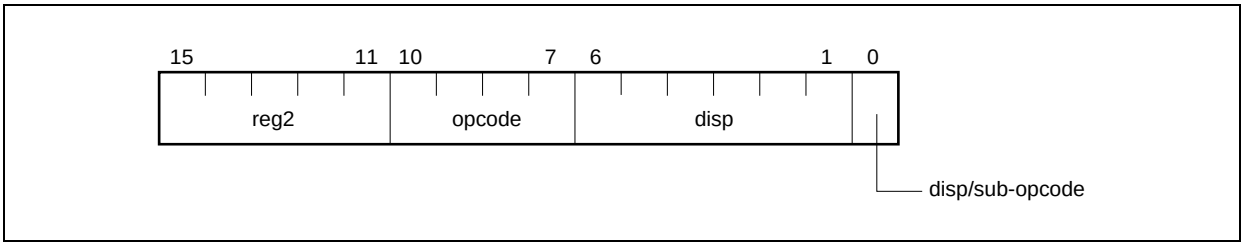
**(3) Conditional branch instruction (Format III)**

A 16-bit instruction format having a 4-bit opcode field, 4-bit condition code field, and an 8-bit displacement field.

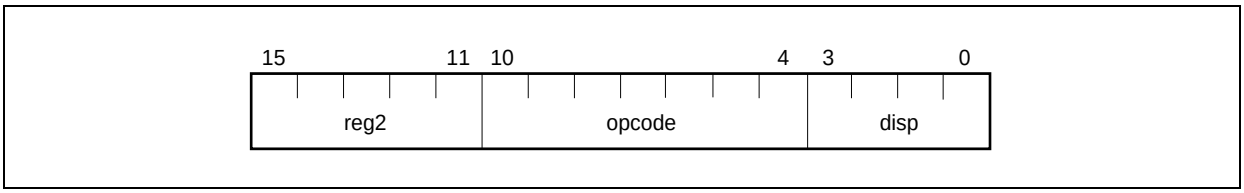



**(4) 16-bit load/store instruction (Format IV)**

A 16-bit instruction format having a 4-bit opcode field, a general-purpose register specification field, and a 7-bit displacement field (or 6-bit displacement field + 1-bit sub-opcode field).

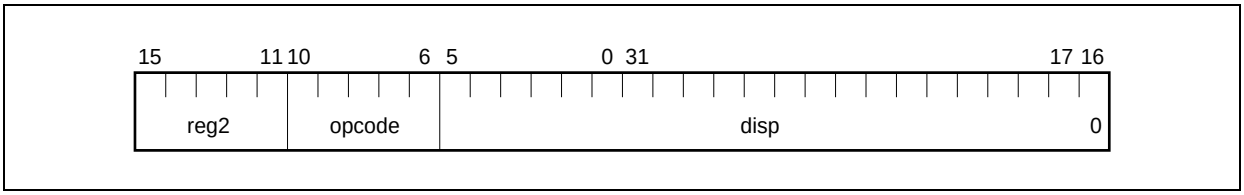



A 16-bit instruction format having a 7-bit opcode field, a general-purpose register specification field, and a 4-bit displacement field.



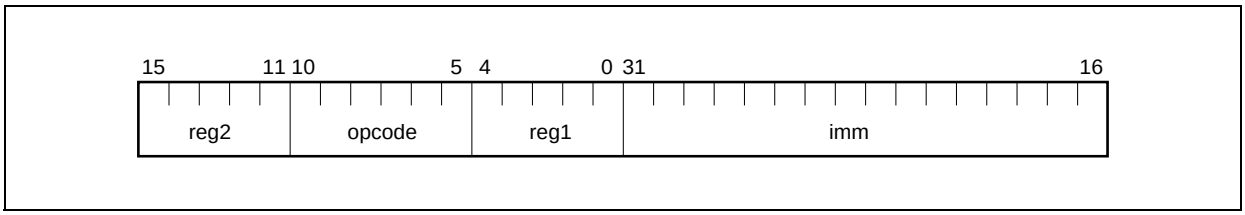

**(5) Jump instruction (Format V)**

A 32-bit instruction format having a 5-bit opcode field, a general-purpose register specification field, and a 22-bit displacement field.



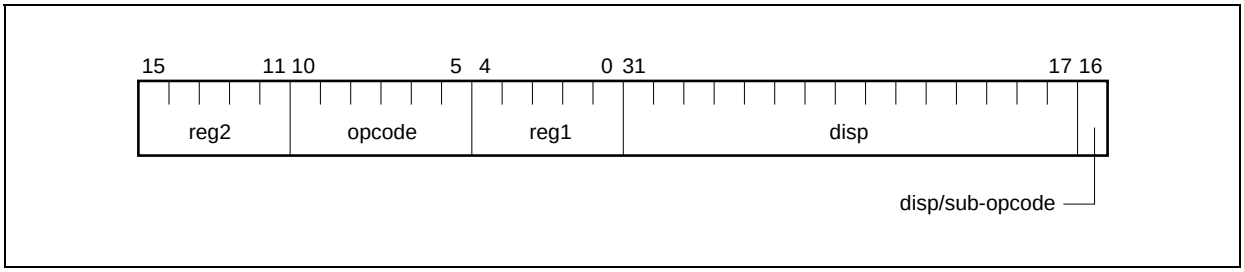
**(6) 3-operand instruction (Format VI)**

A 32-bit instruction format having a 6-bit opcode field, two general-purpose register specification fields, and a 16-bit immediate field.



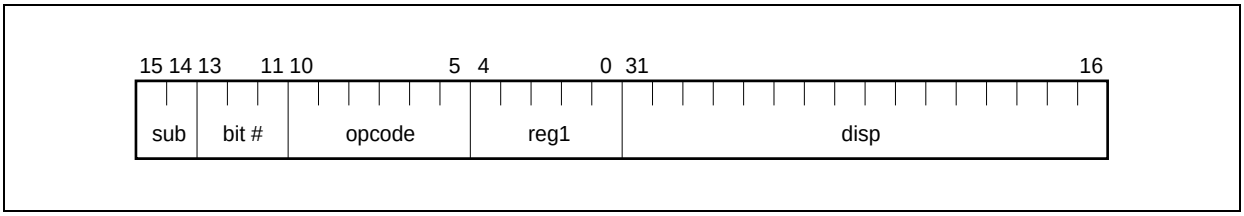

**(7) 32-bit load/store instruction (Format VII)**

A 32-bit instruction format having a 6-bit opcode field, two general-purpose register specification fields, and a 16-bit displacement field (or 15-bit displacement field + 1-bit sub-opcode field).



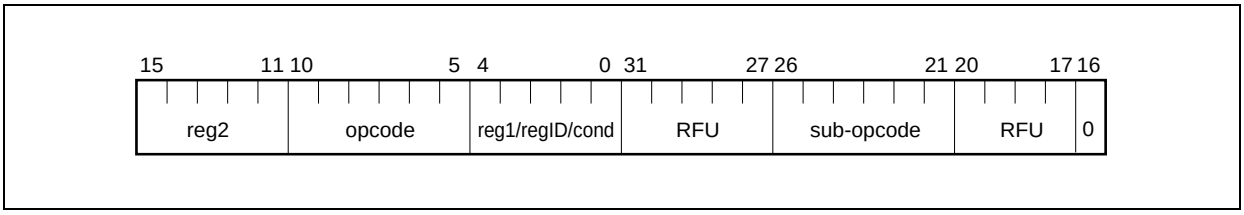

**(8) Bit manipulation instruction (Format VIII)**

A 32-bit instruction format having a 6-bit opcode field, 2-bit sub-opcode field, 3-bit bit specification field, a general-purpose register specification field, and a 16-bit displacement field.



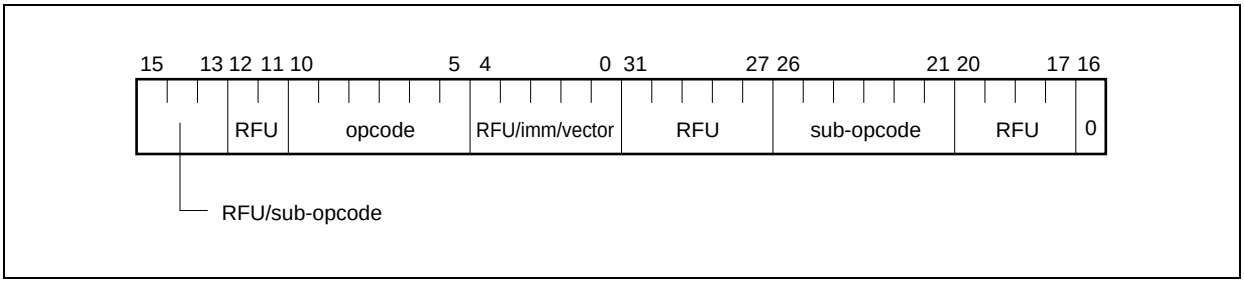

**(9) Extended instruction format 1 (Format IX)**

A 32-bit instruction format having a 6-bit opcode field, 6-bit sub-opcode field, and two general-purpose register specification fields (one field may be register number field (regID) or condition code field (cond)).



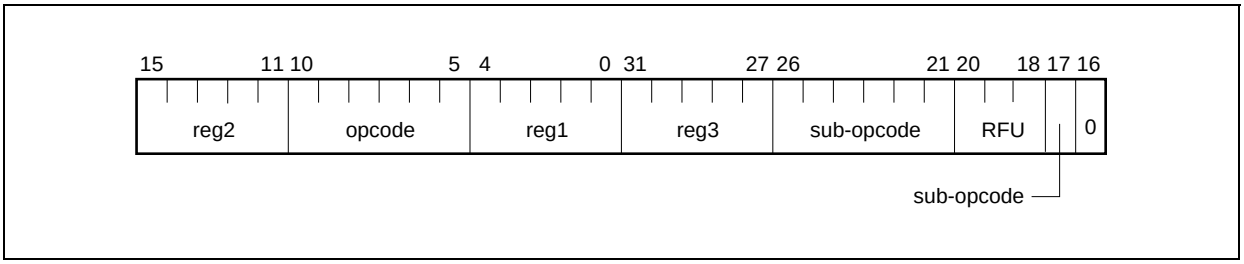
**(10) Extended instruction format 2 (Format X)**

A 32-bit instruction format having a 6-bit opcode field and 6-bit sub-opcode field.



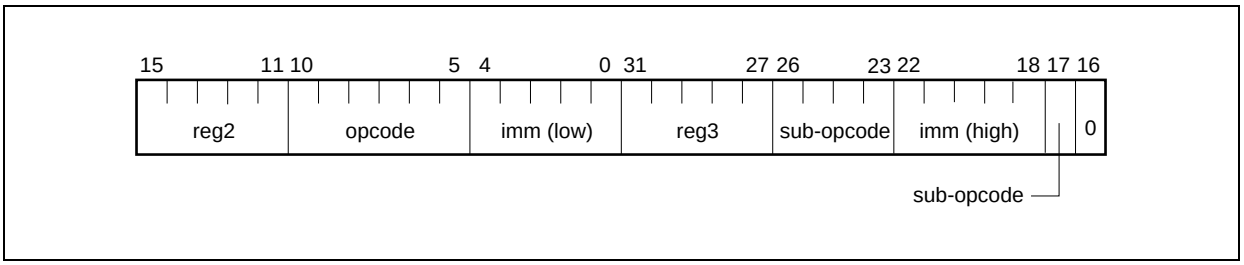

**(11) Extended instruction format 3 (Format XI)**

A 32-bit instruction format having a 6-bit opcode field, 6-bit and 1-bit sub-opcode field, and three general-purpose register specification fields.



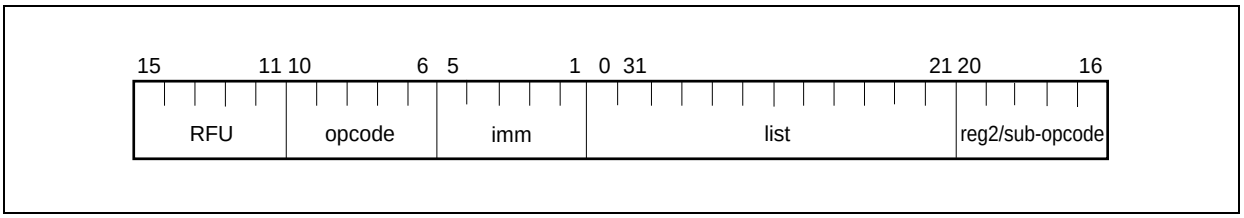

**(12) Extended instruction format 4 (Format XII)**

A 32-bit instruction format having a 6-bit opcode field, 4-bit and 1-bit sub-opcode field, 10-bit immediate field, and two general-purpose register specification fields.




**(13) Stack manipulation instruction 1 (Format XIII)**

A 32-bit instruction format having a 5-bit opcode field, 5-bit immediate field, 12-bit register list field, and one general-purpose register specification field (or 5-bit sub-opcode field).



## 5.2 Outline of Instructions

### (1) Load instructions

Transfer data from memory to a register. The following instructions (mnemonics) are provided.

#### (a) LD instructions

- LD.B: Load byte
- LD.BU: Load byte unsigned
- LD.H: Load halfword
- LD.HU: Load halfword unsigned
- LD.W: Load word

#### (b) SLD instructions

- SLD.B: Short format load byte
- SLD.BU: Short format load byte unsigned
- SLD.H: Short format load halfword
- SLD.HU: Short format load halfword unsigned
- SLD.W: Short format load word

### (2) Store instructions

Transfer data from register to a memory. The following instructions (mnemonics) are provided.

#### (a) ST instructions

- ST.B: Store byte
- ST.H: Store halfword
- ST.W: Store word

#### (b) SST instructions

- SST.B: Short format store byte
- SST.H: Short format store halfword
- SST.W: Short format store word

### (3) Multiply instructions

Execute multiply processing in 1 to 2 clocks with on-chip hardware multiplier. The following instructions (mnemonics) are provided.

- MUL: Multiply word
- MULH: Multiply halfword
- MULHI: Multiply halfword immediate
- MULU: Multiply word unsigned

**(4) Arithmetic operation instructions**

Add, subtract, divide, transfer, or compare data between registers. The following instructions (mnemonics) are provided.

- ADD: Add
- ADDI: Add immediate
- CMOV: Conditional move
- CMP: Compare
- DIV: Divide word
- DIVH: Divide halfword
- DIVHU: Divide halfword unsigned
- DIVU: Divide word unsigned
- MOV: Move
- MOVEA: Move effective address
- MOVHI: Move high halfword
- SASF: Shift and set flag condition
- SETF: Set flag condition
- SUB: Subtract
- SUBR: Subtract reverse

**(5) Saturated operation instructions**

Execute saturation addition and subtraction. If the result of the operation exceeds the maximum positive value (7FFFFFFFH), 7FFFFFFFH is returned. If the result of the operation exceeds the maximum negative value (80000000H), 80000000H is returned. The following instructions (mnemonics) are provided.

- SATADD: Saturated add
- SATSUB: Saturated subtract
- SATSUBI: Saturated subtract immediate
- SATSUBR: Saturated subtract reverse

**(6) Logical operation instructions**

These instructions include logical operation and shift instructions. The shift instructions include arithmetic shift and logical shift instructions. Operands can be shifted by two or more bit positions in one clock cycle by the on-chip barrel shifter. The following instructions (mnemonics) are provided.

- AND: AND
- ANDI: AND immediate
- BSH: Byte swap halfword
- BSW: Byte swap word
- HSW: Halfword swap word
- NOT: NOT
- OR: OR
- ORI: OR immediate
- SAR: Shift arithmetic right
- SHL: Shift logical left
- SHR: Shift logical right
- SXB: Sign extend byte
- SXH: Sign extend halfword

- TST: Test
- XOR: Exclusive OR
- XORI: Exclusive OR immediate
- ZXB: Zero extend byte
- ZXH: Zero extend halfword

**(7) Branch instructions**

These instructions include unconditional branch instructions (JARL, JMP, JR) and a conditional branch instruction (Bcond) that alters the control depending on the status of flags. Program control can be transferred to the address specified by the branch instruction. The following instructions (mnemonics) are provided.

- Bcond (BC, BE, BGE, BGT, BH, BL, BLE, BLT, BN, BNC, BNE, BNH, BNL, BNV, BNZ, BP, BR, BSA, BV, BZ): Branch on condition code
- JARL: Jump and register link
- JMP: Jump register
- JR: Jump relative

**(8) Bit manipulation instructions**

Execute a logical operation to bit data in memory. Only the specified bit is affected. The following instructions (mnemonics) are provided.

- CLR1: Clear bit
- NOT1: Not bit
- SET1: Set bit
- TST1: Test bit

**(9) Special instructions**

These instructions are instructions not included in the categories of instructions described above. The following instructions (mnemonics) are provided.

- CALLT: Call with table look up
- CTRET: Return from CALLT
- DI: Disable interrupt
- DISPOSE: Function dispose
- EI: Enable interrupt
- HALT: Halt
- LDSR: Load system register
- NOP: No operation
- PREPARE: Function prepare
- RETI: Return from trap or interrupt
- STSR: Store system register
- SWITCH: Jump with table look up
- TRAP: Trap

**(10) Debug function instructions**

These instructions are instructions reserved for the debug function. The following instructions (mnemonics) are provided.

- DBRET: Return from debug trap
- DBTRAP: Debug trap

★ **Caution Type C products do not support debug function instructions.**

## 5.3 Instruction Set

In this section, the mnemonic of each instruction is described divided into the following items.

- Instruction format: Indicates the description and operand of the instruction (for symbols, see **Table 5-1**).
- Operation: Indicates the function of the instruction (for symbols, see **Table 5-2**).
- Format: Indicates the instruction format (see **5.1 Instruction Format**).
- Opcode: Indicates the bit field of the instruction opcode (for symbols, see **Table 5-3**).
- Flag: Indicates the operation of the flag that is altered after executing the instruction.
  0 indicates clear (reset), 1 indicates set, and – indicates no change.
- Explanation: Explains the operation of the instruction.
- Remark: Explains the supplementary information of the instruction.
- Caution: Indicates the cautions.

**Table 5-1. Instruction Format Conventions**

| Symbol | Meaning |
|---|---|
| reg1 | General-purpose register (used as source register) |
| reg2 | General-purpose register (mainly used as destination register. Some are also used as source registers.) |
| reg3 | General-purpose register (mainly used as remainder of division results or higher 32 bits of multiply results) |
| bit#3 | 3-bit data for specifying bit number |
| imm× | ×-bit immediate data |
| disp× | ×-bit displacement data |
| regID | System register number |
| vector | 5-bit data for trap vector (00H to1FH) specification |
| cccc | 4-bit data for condition code specification |
| sp | Stack pointer (r3) |
| ep | Element pointer (r30) |
| list 12 | Lists of registers |

**Table 5-2. Operation Conventions (1/2)**

| Symbol | Meaning |
|---|---|
| ← | Assignment |
| GR [ ] | General-purpose register |
| SR [ ] | System register |
| zero-extend (n) | Zero-extends n to word |
| sign-extend (n) | Sign-extends n to word |
| load-memory (a, b) | Reads data of size b from address a |
| store-memory (a, b, c) | Writes data b of size c to address a |
| load-memory-bit (a, b) | Reads bit b from address a |
| store-memory-bit (a, b, c) | Writes c to bit b of address a |

**Table 5-2. Operation Conventions (2/2)**

| Symbol | Meaning |
|---|---|
| saturated (n) | Performs saturation processing of n.<br>If n ≥ 7FFFFFFFH as result of calculation, n = 7FFFFFFFH.<br>If n ≥ 80000000H as result of calculation, n = 80000000H. |
| result | Reflects result on flag |
| Byte | Byte (8 bits) |
| Halfword | Halfword (16 bits) |
| Word | Word (32 bits) |
| + | Add |
| – | Subtract |
| \|\| | Bit concatenation |
| × | Multiply |
| ÷ | Divide |
| % | Remainder of division results |
| AND | And |
| OR | Or |
| XOR | Exclusive Or |
| NOT | Logical negate |
| logically shift left by | Logical left shift |
| logically shift right by | Logical right shift |
| arithmetically shift right by | Arithmetic right shift |

**Table 5-3. Opcode Conventions**

| Symbol | Meaning |
|---|---|
| R | 1-bit data of code specifying reg1 or regID |
| r | 1-bit data of code specifying reg2 |
| w | 1-bit data of code specifying reg3 |
| d | 1-bit data of displacement |
| I | 1-bit data of immediate (indicates higher bits of immediate) |
| i | 1-bit data of immediate |
| cccc | 4-bit data for condition code specification |
| CCCC | 4-bit data for condition code specification of Bcond instruction |
| bbb | 3-bit data for bit number specification |
| L | 1-bit data of code specifying general-purpose register in register list |

**<Arithmetic operation instruction>**

**Add register/immediate**

# ADD

**Add**

**Instruction format**
(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

**Opcode**

(1)

| 15 | 0 |
|---|---|
| rrrrr001110RRRRR | |

(2)

| 15 | 0 |
|---|---|
| rrrrr010010iiiii | |

**Flag**

| | |
|---|---|
| CY | 1 if a carry occurs from MSB; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise 0. |
| SAT | – |

**Explanation**
(1) Adds the word data of general-purpose register reg1 to the word data of general-purpose register reg2, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.
(2) Adds 5-bit immediate data, sign-extended to word length, to the word data of general-purpose register reg2, and stores the result in general-purpose register reg2.

**<Arithmetic operation instruction>**

**Add immediate**

# ADDI

**Add Immediate**

**Instruction format** ADDI imm16, reg1, reg2

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

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110000RRRRR | iiiiiiiiiiiiiiii |

**Flag**

| | |
|---|---|
| CY | 1 if a carry occurs from MSB; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise 0. |
| SAT | – |

**Explanation** Adds 16-bit immediate data, sign-extended to word length, to the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Logical operation instruction>**

# AND





**Instruction format** AND reg1, reg2

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

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr001010RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise 0. |
| SAT | – |

**Explanation** ANDs the word data of general-purpose register reg2 with the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Logical operation instruction>**

# ANDI

**AND immediate**

**And Immediate**

**Instruction format** ANDI imm16, reg1, reg2

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

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110110RRRRR | iiiiiiiiiiiiiiii |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 0 |
| Z | 1 if the result of an operation is 0; otherwise 0. |
| SAT | – |

**Explanation** ANDs the word data of general-purpose register reg1 with the value of the 16-bit immediate data, zero-extended to word length, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Branch instruction>**

**Branch on condition code with 9-bit displacement**

# Bcond

**Branch on Condition Code**

**Instruction format** Bcond disp9

**Operation** if conditions are satisfied
then PC ← PC + sign-extend (disp9)

**Format** Format III

**Opcode**

| 15 | 0 |
|---|---|
| ddddd1011dddCCCC | |

dddddddd is the higher 8 bits of disp9.

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Tests each flag of the PSW specified by the instruction. Branches if a specified condition is satisfied; otherwise, executes the next instruction. The branch destination PC holds the sum of the current PC value and 9-bit displacement, which is 8-bit immediate shifted 1 bit and sign-extended to word length.

**Remark** Bit 0 of the 9-bit displacement is masked by 0. The current PC value used for calculation is the address of the first byte of this instruction. If the displacement value is 0, therefore, the branch destination is this instruction itself.

**Table 5-4. Bcond Instructions**

| Instruction | | Condition Code (CCCC) | Status of Flag | Branch Condition |
|---|---|---|---|---|
| Signed integer | BGE | 1110 | (S xor OV) = 0 | Greater than or equal signed |
| | BGT | 1111 | ( (S xor OV) or Z) = 0 | Greater than signed |
| | BLE | 0111 | ( (S xor OV) or Z) = 1 | Less than or equal signed |
| | BLT | 0110 | (S xor OV) = 1 | Less than signed |
| Unsigned integer | BH | 1011 | (CY or Z) = 0 | Higher (Greater than) |
| | BL | 0001 | CY = 1 | Lower (Less than) |
| | BNH | 0011 | (CY or Z) = 1 | Not higher (Less than or equal) |
| | BNL | 1001 | CY = 0 | Not lower (Greater than or equal) |
| Common | BE | 0010 | Z = 1 | Equal |
| | BNE | 1010 | Z = 0 | Not equal |
| Others | BC | 0001 | CY = 1 | Carry |
| | BN | 0100 | S = 1 | Negative |
| | BNC | 1001 | CY = 0 | No carry |
| | BNV | 1000 | OV = 0 | No overflow |
| | BNZ | 1010 | Z = 0 | Not zero |
| | BP | 1100 | S = 0 | Positive |
| | BR | 0101 | – | Always (unconditional) |
| | BSA | 1101 | SAT = 1 | Saturated |
| | BV | 0000 | OV = 1 | Overflow |
| | BZ | 0010 | Z = 1 | Zero |

**Caution** If executing a conditional branch instruction of a signed integer (BGE, BGT, BLE, or BLT) when the SAT flag is set to 1 as a result of executing a saturated operation instruction, the branch condition loses its meaning. In ordinary operations, if an overflow occurs, the S flag is inverted ($0 \rightarrow 1$ or $1 \rightarrow 0$). This is because the result is a negative value if it exceeds the maximum positive value and it is a positive value if it exceeds the maximum negative value. However, when a saturated operation instruction is executed, and if the result exceeds the maximum positive value, the result is saturated with a positive value; if the result exceeds the maximum negative value, the result is saturated with a negative value. Unlike the ordinary operation, therefore, the S flag is not inverted even if an overflow occurs. Hence, the S flag is affected differently when the instruction is a saturated operation, as opposed to an ordinary operation. A branch condition which is an XOR of the S and OV flags will therefore have no meaning.

**<Logical operation instruction>**

**Byte swap halfword**

# BSH

**Byte Swap Halfword**

**Instruction format** BSH reg2, reg3

**Operation** GR [reg3] ← GR [reg2] (23:16) || GR [reg2] (31:24) || GR [reg2] (7:0) || GR [reg2] (15:8)

**Format** Format XII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr11111100000 | wwwww01101000010 |

**Flag**

| | |
|---|---|
| CY | 1 if one or more bytes in the lower halfword of the operation result is 0; otherwise 0. |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the lower halfword data of the operation result is 0; otherwise, 0. |
| SAT | – |

**Explanation** Endian translation.

**<Logical operation instruction>**

**Byte swap word**

# BSW

**Byte Swap Word**

**Instruction format** BSW reg2, reg3

**Operation** GR [reg3] ← GR [reg2] (7:0) || GR [reg2] (15:8) || GR [reg2] (23:16) || GR [reg2] (31:24)

**Format** Format XII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr11111100000 | wwwww01101000000 |

**Flag**

| | |
|---|---|
| CY | 1 if one or more bytes in the word data of the operation result is 0; otherwise 0. |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the word data of the operation result is 0; otherwise, 0. |
| SAT | – |

**Explanation** Endian translation.

**<Special instruction>**

**Call with table look up**

# CALLT

**Call with Table Look Up**

**Instruction format** CALLT imm6

**Operation**
CTPC ← PC + 2 (return PC)
CTPSW ← PSW
adr ← CTBP + zero-extend (imm6 logically shift left by 1)
PC ← CTBP + zero-extend (Load-memory (adr, Halfword))

**Format** Format II

**Opcode**

| 15 | 0 |
|---|---|
| 0000001000iiiiii | |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Performs processing as follows.

<1> Transfers the restored PC and PSW contents to CTPC and CTPSW.
<2> Adds the CTBP value and the 6-bit immediate data logically shifted left by 1 bit and zero-extended to word length, to generate a 32-bit table entry address.
<3> Loads the halfword of the address generated in step <2> and zero-extends to word length.
<4> Adds the data of step <3> and the CTBP value to generate a 32-bit target address.
<5> Branches to the target address generated in step <4>.

**Caution** If an interrupt is generated during instruction execution, the execution of that instruction may stop after the end of the read/write cycle. Execution is resumed after returning from the interrupt.

**<Bit manipulation instruction>**

**Clear bit**

# CLR1

**Clear Bit**

**Instruction format**

(1) CLR1 bit#3, disp16 [reg1]
(2) CLR1 reg2, [reg1]

**Operation**

(1) adr ← GR [reg1] + sign-extend (disp16)
Z flag ← Not (Load-memory-bit (adr, bit#3))
Store-memory-bit (adr, bit#3, 0)
(2) adr ← GR [reg1]
Z flag ← Not (Load-memory-bit (adr, reg2))
Store-memory-bit (adr, reg2, 0)

**Format**

(1) Format VIII
(2) Format IX

**Opcode**

| | 15 0 | 31 16 |
|---|---|---|
| (1) | 10bbb111110RRRRR | ddddddddddddddddd |

| | 15 0 | 31 16 |
|---|---|---|
| (2) | rrrrr111111RRRRR | 0000000011100100 |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | 1 if bit specified by operands = 0, 0 if bit specified by operands = 1 |
| SAT | – |

**Explanation**

(1) Adds the data of general-purpose register reg1 to the 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Then reads the byte data referenced by the generated address, clears the bit specified by the 3-bit bit number, and writes back to the original address.
(2) Reads the data of general-purpose register reg1 to generate a 32-bit address. Then reads the byte data referenced by the generated address, clears the bit specified by the data of the lower 3 bits of reg2, and writes back to the original address.

**Remark**

The Z flag of the PSW indicates whether the specified bit was a 0 or 1 before this instruction was executed. It does not indicate the content of the specified bit after this instruction has been executed.

**<Arithmetic operation instruction>**

# CMOV

**Conditional move**

**Conditional Move**

**Instruction format**

(1) CMOV cccc, reg1, reg2, reg3
(2) CMOV cccc, imm5, reg2, reg3

**Operation**

(1) if conditions are satisfied
then GR [reg3] ← GR [reg1]
else GR [reg3] ← GR [reg2]
(2) if conditions are satisfied
then GR [reg3] ← sign-extend (imm5)
else GR [reg3] ← GR [reg2]

**Format**

(1) Format XI
(2) Format XII

**Opcode**

(1)

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | wwwww011001cccc0 |

(2)

| 15 0 | 31 16 |
|---|---|
| rrrrr111111iiiii | wwwww011000cccc0 |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation**

(1) The data of general-purpose register reg1 is transferred to general-purpose register reg3 if the condition specified by condition code “cccc” is satisfied; otherwise, the data of general-purpose register reg2 is transferred to general-purpose register reg3. One of the codes shown in **Table 5-5 Condition Codes** should be specified as the condition code “cccc”.
(2) The data of 5-bit immediate, sign-extended to word length, is transferred to general-purpose register reg3 if the condition specified by condition code “cccc” is satisfied; otherwise, the data of general-purpose register reg2 is transferred to general-purpose register reg3. One of the codes shown in **Table 5-5 Condition Codes** should be specified as the condition code “cccc”.

**Remark**

See SETF instruction.

**<Arithmetic operation instruction>**

**Compare register/immediate (5-bit)**

# CMP

**Compare**

**Instruction format** (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

**Opcode**

(1)

| 15 | 0 |
|---|---|
| rrrrr001111RRRRR | |

(2)

| 15 | 0 |
|---|---|
| rrrrr010011iiiii | |

**Flag**

| | |
|---|---|
| CY | 1 if a borrow to MSB occurs; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise 0. |
| S | 1 if the result of the operation is negative; otherwise, 0. |
| Z | 1 if the result of the operation is 0; otherwise, 0. |
| SAT | – |

**Explanation**

(1) Compares the word data of general-purpose register reg2 with the word data of general-purpose register reg1, and indicates the result by using the flags of the PSW. To compare, the contents of general-purpose register reg1 are subtracted from the word data of general-purpose register reg2. The data of general-purpose registers reg1 and reg2 is not affected.

(2) Compares the word data of general-purpose register reg2 with 5-bit immediate data, sign-extended to word length, and indicates the result by using the flags of the PSW. To compare, the contents of the sign-extended immediate data are subtracted from the word data of general-purpose register reg2. The data of general-purpose register reg2 is not affected.

**<Special instruction>**

# CTRET

**Return from CALLT**

**Return from CALLT**

**Instruction format** CTRET

**Operation**
PC ← CTPC
PSW ← CTPSW

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 0000011111100000 | 0000000101000100 |

**Flag**

| | |
|---|---|
| CY | Value read from CTPSW is restored. |
| OV | Value read from CTPSW is restored. |
| S | Value read from CTPSW is restored. |
| Z | Value read from CTPSW is restored. |
| SAT | Value read from CTPSW is restored. |

**Explanation** Fetches the restored PC and PSW from the appropriate system register and returns from the routine called by CALLT instruction. The operations of this instruction are as follows.

(1) The restored PC and PSW are read from CTPC and CTPSW.
(2) Once the PC and PSW are restored to the return values, control is transferred to the return address.

**<Debug function instruction>**

**Return from debug trap**

# DBRET

**Return from debug trap**

**Instruction format** DBRET

**Operation**
PC ← DBPC
PSW ← DBPSW

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 0000011111100000 | 0000000101000110 |

**Flag**
CY Value read from DBPSW is restored.
OV Value read from DBPSW is restored.
S Value read from DBPSW is restored.
Z Value read from DBPSW is restored.
SAT Value read from DBPSW is restored.

**Explanation** Fetches the restored PC and PSW from the appropriate system register and returns from debug mode.

**Caution**
(1) Because the DBRET instruction is for debugging, it is essentially used by debug tools. When a debug tool is using this instruction, therefore, use of it in the application may cause a malfunction.

★ (2) Type C products do not support the DBRET instruction.

**<Debug function instruction>**

**Debug trap**

# DBTRAP

**Debug trap**

**Instruction format** DBTRAP

**Operation**
DBPC ← PC + 2 (restored PC)
DBPSW ← PSW
PSW.NP ← 1
PSW.EP ← 1
PSW.ID ← 1
PC ← 00000060H

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 1111100001000000 | |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Saves the contents of the restored PC (address of the instruction following the DBTRAP instruction) and the PSW to DBPC and DBPSW, respectively, and sets the NP, EP, and ID flags of the PSW to 1.
Next, the handler address (00000060H) of the exception trap is set to the PC, and control shifts to the PC. PSW flags other than NP, EP, and ID flags are unaffected.
Note that the value saved to DBPC is the address of the instruction following the DBTRAP instruction.

**Caution**
(1) Because the DBTRAP instruction is for debugging, it is essentially used by debug tools. When a debug tool is using this instruction, therefore, use of it in the application may cause a malfunction.

★ (2) Type C products do not support the DBTRAP instruction.

**<Special instruction>**

**Disable interrupt**

# DI

**Disable Interrupt**

**Instruction format** DI

**Operation** PSW.ID ← 1 (Disables maskable interrupt)

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 0000011111100000 | 0000000101100000 |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |
| ID | 1 |

**Explanation** Sets the ID flag of the PSW to 1 to disable the acknowledgment of maskable interrupts during execution of this instruction.

**Remark** Interrupts are not sampled during execution of this instruction. The PSW flag actually becomes valid at the start of the next instruction. But because interrupts are not sampled during instruction execution, interrupts are immediately disabled. Non-maskable interrupts (NMI) are not affected by this instruction.

**<Special instruction>**

# DISPOSE

**Function dispose**

**Function Dispose**

**Instruction format** (1) DISPOSE imm5, list12
(2) DISPOSE imm5, list12, [reg1]

**Operation**

(1) sp ← sp + zero-extend (imm5 logically shift left by 2)
GR [reg in list12] ← Load-memory (sp, Word)
sp ← sp + 4
repeat 2 steps above until all regs in list12 are loaded

(2) sp ← sp + zero-extend (imm5 logically shift left by 2)
GR [reg in list12] ← Load-memory (sp, Word)
sp ← sp + 4
repeat 2 states above until all regs in list12 are loaded
PC ← GR [reg1]

**Format** Format XIII

**Opcode**

(1)

| 15 ... 0 | 31 ... 16 |
|---|---|
| 0000011001iiiiiL | LLLLLLLLLLL00000 |

(2)

| 15 ... 0 | 31 ... 16 |
|---|---|
| 0000011001iiiiiL | LLLLLLLLLLLRRRRR |

RRRRR must not be 00000.

LLLLLLLLLLL indicates the bit value corresponding to the register list (list12) (for example, “L” of bit 21 in an opcode indicates the value of bit 21 of list12). list12 is a 32-bit register list defined as follows.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 ... 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| r24 | r25 | r26 | r27 | r20 | r21 | r22 | r23 | r28 | r29 | r31 | – | r30 |

Bits 31 to 21 and bit 0 correspond to each bit of the general-purpose registers (r21 to r31). The register corresponding to the set bit (1) is specified as the manipulation target. For example, when r20 and r30 are specified, list12 values are as follows (the set values of bits 20 to 1 to which registers do not correspond can be 0 or 1 (don’t care)).

- If the values of all the bits to which registers do not correspond are set to 0: 08000001H
- If the values of all the bits to which registers do not correspond are set to 1: 081FFFFFH

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation**

(1) Adds the data of 5-bit immediate imm5, logically shifted left by 2 and zero-extended to word length, to sp. Then pops (loads data from the address specified by sp and adds 4 to sp) the general-purpose registers listed in list12. Bit 0 of the address is masked by 0.

(2) Adds the data of 5-bit immediate imm5, logically shifted left by 2 and zero-extended to word length, to sp. Then pops (loads data from the address specified by sp and adds 4 to sp) the general-purpose registers listed in list12, transfers control to the address specified by general-purpose register reg1. Bit 0 of the address is masked by 0.

**Remark**

The general-purpose registers in list12 are loaded in the downward direction (r31, r30, ... r20).

The 5-bit immediate imm5 is used to restore a stack frame for auto variables and temporary data.

The lower 2 bits of the address specified by sp are always masked by 0 even if misaligned access is enabled.

If an interrupt occurs before updating sp, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction (sp will retain its original value prior to the start of execution).

**Caution**

If an interrupt is generated during instruction execution, due to manipulation of the stack, the execution of that instruction may stop after the read/write cycle and register value rewriting are complete. Execution is resumed after returning from the interrupt.

**<Arithmetic operation instruction>**

# DIV

**Divide word**

**Divide Word**

**Instruction format** DIV reg1, reg2, reg3

**Operation**
GR [reg2] ← GR [reg2] ÷ GR [reg1]
GR [reg3] ← GR [reg2] % GR [reg1]

**Format** Format XI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | wwwww01011000000 |

**Flag**
CY –
OV 1 if overflow occurs; otherwise, 0.
S 1 if the result of an operation is negative; otherwise, 0.
Z 1 if the result of an operation is 0; otherwise, 0.
SAT –

**Explanation** Divides the word data of general-purpose register reg2 by the word data of general-purpose register reg1, and stores the quotient in general-purpose register reg2, and the remainder in general-purpose register reg3. If the data is divided by 0, overflow occurs, and the quotient is undefined. The data of general-purpose register reg1 is not affected.

**Remark** Overflow occurs when the maximum negative value (80000000H) is divided by –1 (in which case the quotient is 80000000H) and when data is divided by 0 (in which case the quotient is undefined).
If an interrupt occurs while this instruction is being executed, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction. Also, general-purpose registers reg1 and reg2 will retain their original values prior to the start of execution.
If the address of reg2 is the same as the address of reg3, the remainder is stored in reg2 (= reg3).

**<Arithmetic operation instruction>**

# DIVH

**Divide halfword**

**Divide Halfword**

**Instruction format** (1) DIVH reg1, reg2
(2) DIVH reg1, reg2, reg3

**Operation** (1) GR [reg2] ← GR [reg2] ÷ GR [reg1]
(2) GR [reg2] ← GR [reg2] ÷ GR [reg1]
GR [reg3] ← GR [reg2] % GR [reg1]

**Format** (1) Format I
(2) Format XI

**Opcode**

(1)

| 15 | 0 |
|---|---|
| rrrrr000010RRRRR | |

(2)

| 15 | 0 | 31 | 16 |
|---|---|---|---|
| rrrrr111111RRRRR | | wwwww01010000000 | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** (1) Divides the word data of general-purpose register reg2 by the lower halfword data of general-purpose register reg1, and stores the quotient in general-purpose register reg2. If the data is divided by 0, overflow occurs, and the quotient is undefined. The data of general-purpose register reg1 is not affected.
(2) Divides the word data of general-purpose register reg2 by the lower halfword data of general-purpose register reg1, and stores the quotient in general-purpose register reg2 and the remainder in general-purpose register reg3. If the data is divided by 0, overflow occurs, and the quotient is undefined. The data of general-purpose register reg1 is not affected.

**Remark** (1) The remainder is not stored. Overflow occurs when the maximum negative value (80000000H) is divided by –1 (in which case the quotient is 80000000H) and when data is divided by 0 (in which case the quotient is undefined). If an interrupt occurs while this instruction is being executed, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction. Also, general-purpose registers reg1 and reg2 will retain their original values prior to the start of execution.
Do not specify r0 as the destination register reg2.
The higher 16 bits of general-purpose register reg1 are ignored when division is executed.

(2) Overflow occurs when the maximum negative value (80000000H) is divided by –1 (in which case the quotient is 80000000H) and when data is divided by 0 (in which case the quotient is undefined).

If an interrupt occurs while this instruction is being executed, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction. Also, general-purpose registers reg1 and reg2 will retain their original values prior to the start of execution.

The higher 16 bits of general-purpose register reg1 are ignored when division is executed.

If the address of reg2 is the same as the address of reg3, the remainder is stored in reg2 (= reg3).

**<Arithmetic operation instruction>**

**Divide halfword unsigned**

# DIVHU

**Divide Halfword Unsigned**

**Instruction format** DIVHU reg1, reg2, reg3

**Operation**
GR [reg2] ← GR [reg2] ÷ GR [reg1]
GR [reg3] ← GR [reg2] % GR [reg1]

**Format** Format XI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | wwwww01010000010 |

**Flag**
CY –
OV 1 if overflow occurs; otherwise, 0.
S 1 if the MSB of the word data of the operation result is 1; otherwise, 0.
Z 1 if the result of an operation is 0; otherwise, 0.
SAT –

**Explanation** Divides the word data of general-purpose register reg2 by the lower halfword data of general-purpose register reg1, and stores the quotient in general-purpose register reg2, and the remainder in general-purpose register reg3. If the data is divided by 0, overflow occurs, and the quotient is undefined. The data of general-purpose register reg1 is not affected.

**Remark** Overflow occurs when data is divided by 0 (in which case the quotient is undefined).
If an interrupt occurs while this instruction is being executed, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction. Also, general-purpose registers reg1 and reg2 will retain their original values prior to the start of execution.
If the address of reg2 is the same as the address of reg3, the remainder is stored in reg2 (= reg3).

**<Arithmetic operation instruction>**

**Divide word unsigned**

# DIVU

**Divide Word Unsigned**

**Instruction format** DIVU reg1, reg2, reg3

**Operation**
GR [reg2] ← GR [reg2] ÷ GR [reg1]
GR [reg3] ← GR [reg2] % GR [reg1]

**Format** Format XI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | wwwww01011000010 |

**Flag**
CY –
OV 1 if overflow occurs; otherwise, 0.
S 1 if the MSB of the word data of the operation result is 1; otherwise, 0.
Z 1 if the result of an operation is 0; otherwise, 0.
SAT –

**Explanation** Divides the word data of general-purpose register reg2 by the word data of general-purpose register reg1, and stores the quotient in general-purpose register reg2, and the remainder in general-purpose register reg3. If the data is divided by 0, overflow occurs, and the quotient is undefined. The data of general-purpose register reg1 is not affected.

**Remark** Overflow occurs when data is divided by 0 (in which case the quotient is undefined).
If an interrupt occurs while this instruction is being executed, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction. Also, general-purpose registers reg1 and reg2 will retain their original values prior to the start of execution.
If the address of reg2 is the same as the address of reg3, the remainder is stored in reg2 (= reg3).

**<Special instruction>**

**Enable interrupt**

# EI

**Enable Interrupt**

**Instruction format** EI

**Operation** PSW.ID ← 0 (enables maskable interrupt)

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 1000011111100000 | 0000000101100000 |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |
| ID | 0 |

**Explanation** Clears the ID flag of the PSW to 0 and enables the acknowledgment of maskable interrupts beginning at the next instruction.

**Remark** Interrupts are not sampled during instruction execution.

**<Special instruction>**

# HALT

**Halt**

**Halt**

**Instruction format** HALT

**Operation** Halts

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 0000011111100000 | 0000000100100000 |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Stops the operating clock of the CPU and places the CPU in the HALT mode.

**Remark** The HALT mode is exited by any of the following three events.

- Reset input
- Non-maskable interrupt request (NMI input)
- Unmasked maskable interrupt request (when ID of PSW = 0)

If an interrupt is acknowledged in the HALT mode, the address of the following instruction is stored in EIPC or FEPC.

**<Logical operation instruction>**

**Halfword swap word**

# HSW

**Halfword Swap Word**

**Instruction format** HSW reg2, reg3

**Operation** GR [reg3] ← GR [reg2] (15:0) || GR [reg2] (31:16)

**Format** Format XII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr11111100000 | wwwww01101000100 |

**Flag**

| | |
|---|---|
| CY | 1 if one or more halfwords in the word data of the operation result is 0; otherwise 0. |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the word data of the operation result is 0; otherwise, 0. |
| SAT | – |

**Explanation** Endian translation.

**<Branch instruction>**

**Jump and register link**

# JARL

**Jump and Register Link**

**Instruction format** JARL disp22, reg2

**Operation**
GR [reg2] ← PC + 4
PC ← PC + sign-extend (disp22)

**Format** Format V

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr11110dddddd | ddddddddddddddd0 |

ddddddddddddddddddddd is the higher 21 bits of disp22.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Saves the current PC value plus 4 to general-purpose register reg2, adds the current PC value and 22-bit displacement, sign-extended to word length, and transfers control to the PC. Bit 0 of the 22-bit displacement is masked by 0.

**Remark** The current PC value used for calculation is the address of the first byte of this instruction. If the displacement value is 0, the branch destination is this instruction itself.
This instruction is equivalent to a call subroutine instruction, and saves the restored PC address to general-purpose register reg2. The JMP instruction, which is equivalent to a subroutine-return instruction, can be used to specify the general-purpose register containing the return address saved during the JARL subroutine-call instruction as reg1, to restore the program counter.

**<Branch instruction>**

**Jump register**

# JMP

**Jump Register**

**Instruction format** JMP [reg1]

**Operation** PC ← GR [reg1]

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 00000000011RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Transfers control to the address specified by general-purpose register reg1. Bit 0 of the address is masked by 0.

**Remark** When using this instruction as the subroutine-return instruction, specify the general-purpose register containing the return address saved during the JARL subroutine-call instruction, to restore the program counter. When using the JARL instruction, which is equivalent to the subroutine-call instruction, store the PC return address in general-purpose register reg2.

**<Branch instruction>**

**Jump relative**

# JR

**Jump Relative**

**Instruction format** JR disp22

**Operation** PC ← PC + sign-extend (disp22)

**Format** Format V

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 0000011110dddddd | ddddddddddddddd0 |

ddddddddddddddddddddd is the higher 21 bits of disp22.

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the 22-bit displacement, sign-extended to word length, to the current PC value and stores the value in the PC, and then transfers control to the PC. Bit 0 of the 22-bit displacement is masked by 0.

**Remark** The current PC value used for the calculation is the address of the first byte of this instruction itself. Therefore, if the displacement value is 0, the jump destination is this instruction.

**<Load instruction>**

**Load byte**

# LD.B

**Load**

**Instruction format** LD.B disp16 [reg1], reg2

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
GR [reg2] ← sign-extend (Load-memory (adr, Byte))

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111000RRRRR | dddddddddddddddd |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the data of general-purpose register reg1 to a 16-bit displacement sign-extended to word length to generate a 32-bit address. Byte data is read from the generated address, sign-extended to word length, and stored in general-purpose register reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Load instruction>**

**Load byte unsigned**

# LD.BU

**Load**

**Instruction format** LD.BU disp16 [reg1], reg2

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
GR [reg2] ← zero-extend (Load-memory (adr, Byte))

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr11110bRRRRR | ddddddddddddddd1 |

ddddddddddddddd is the higher 15 bits of disp16. b is the bit 0 of disp16.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the data of general-purpose register reg1 to a 16-bit displacement sign-extended to word length to generate a 32-bit address. Byte data is read from the generated address, zero-extended to word length, and stored in general-purpose register reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Load instruction>**

**Load halfword**

# LD.H

**Load**

**Instruction format** LD.H disp16 [reg1], reg2

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
GR [reg2] ← sign-extend (Load-memory (adr, Halfword))

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111001RRRRR | ddddddddddddddd0 |

ddddddddddddddd is the higher 15 bits of disp16.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the data of general-purpose register reg1 to a 16-bit displacement sign-extended to word length to generate a 32-bit address. Halfword data is read from the generated address, sign-extended to word length, and stored in general-purpose register reg2.

**Caution** The result of adding the data of general-purpose register reg1 and the 16-bit displacement sign-extended to word length can be of two types depending on the type of data to be accessed (halfword, word), and the misalign mode setting.

- Lower bits are masked to 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled in type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Load instruction>**

**Load halfword unsigned**

# LD.HU

**Load**

**Instruction format** LD.HU disp16 [reg1], reg2

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
GR [reg2] ← zero-extend (Load-memory (adr, Halfword))

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | ddddddddddddddd1 |

ddddddddddddddd is the higher 15 bits of disp16.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the data of general-purpose register reg1 to a 16-bit displacement sign-extended to word length to generate a 32-bit address. Halfword data is read from the generated address, zero-extended to word length, and stored in general-purpose register reg2.

**Caution** The result of adding the data of general-purpose register reg1 and the 16-bit displacement sign-extended to word length can be of two types depending on the type of data to be accessed (halfword, word), and the misalign mode setting.

- Lower bits are masked to 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled for the type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Load instruction>**

**Load word**

# LD.W

**Load**

**Instruction format** LD.W disp16 [reg1], reg2

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
GR [reg2] ← Load-memory (adr, Word)

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111001RRRRR | ddddddddddddddd1 |

ddddddddddddddd is the higher 15 bits of disp16.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the data of general-purpose register reg1 to a 16-bit displacement sign-extended to word length to generate a 32-bit address. Word data is read from the generated address, and stored in general-purpose register reg2.

**Caution** The result of adding the data of general-purpose register reg1 and the 16-bit displacement sign-extended to word length can be of two types depending on the type of data to be accessed (halfword, word), and the misalign mode setting.

- Lower bits are masked to 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled for the type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is processed. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Special instruction>**

# LDSR

**Load to system register**

**Load to System Register**

**Instruction format** LDSR reg2, regID

**Operation** SR [regID] ← GR [reg2]

**Format** Format IX

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | 0000000000100000 |

**Caution The source register in this instruction is represented by reg2 for convenience in describing its mnemonic . In the opcode, however, the reg1 field is used for the source register. Unlike other instructions, therefore, the register specified in the mnemonic description has a different meaning in the opcode.**

rrrrr**: regID specification**

RRRRR**: reg2 specification**

**Flag**

CY – (See **Remark** below.)
OV – (See **Remark** below.)
S – (See **Remark** below.)
Z – (See **Remark** below.)
SAT – (See **Remark** below.)

**Explanation** Loads the word data of general-purpose register reg2 to a system register specified by the system register number (regID). The data of general-purpose register reg2 is not affected.

**Remark** If the system register number (regID) is equal to 5 (PSW register), the values of the corresponding bits of the PSW are set according to the contents of reg2. Also, interrupts are not sampled when the PSW is being written with a new value. If the ID flag is enabled with this instruction, interrupt disabling begins at the start of execution, even though the ID flag does not become valid until the beginning of the next instruction.

**Caution** The system register number regID is a number which identifies a system register. Accessing system registers which are reserved or write-prohibited is prohibited and will lead to undefined results.

**<Arithmetic operation instruction>**

# MOV

**Move register/immediate (5-bit)/immediate (32-bit)**

**Move**

**Instruction format**

(1) MOV reg1, reg2
(2) MOV imm5, reg2
(3) MOV imm32, reg1

**Operation**

(1) GR [reg2] ← GR [reg1]
(2) GR [reg2] ← sign-extend (imm5)
(3) GR [reg1] ← imm32

**Format**

(1) Format I
(2) Format II
(3) Format VI

**Opcode**

(1)

| 15 | 0 |
|---|---|
| rrrrr000000RRRRR | |

(2)

| 15 | 0 |
|---|---|
| rrrrr010000iiiii | |

(3)

| 15 | 0 | 31 | 16 | 47 | 32 |
|---|---|---|---|---|---|
| 00000110001RRRRR | | iiiiiiiiiiiiiiii | | IIIIIIIIIIIIIIII | |

i (bits 31 to 16) refers to the lower 16 bits of 32-bit immediate data.
I (bits 47 to 32) refers to the higher 16 bits of 32-bit immediate data.

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation**

(1) Transfers the word data of general-purpose register reg1 to general-purpose register reg2. The data of general-purpose register reg1 is not affected.
(2) Transfers the value of a 5-bit immediate data, sign-extended to word length, to general-purpose register reg2.
Do not specify r0 as the destination register reg2.
(3) Transfers the value of a 32-bit immediate data to general-purpose register reg1.

**<Arithmetic operation instruction>**

# MOVEA

**Move effective address**

**Move Effective Address**

**Instruction format** MOVEA imm16, reg1, reg2

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

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110001RRRRR | iiiiiiiiiiiiiiii |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Adds the 16-bit immediate data, sign-extended to word length, to the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected. The flags are not affected by the addition.
Do not specify r0 as the destination register reg2.

**Remark** This instruction calculates a 32-bit address and stores the result without affecting the PSW flags.

**<Arithmetic operation instruction>**

# MOVHI

**Move high halfword**

**Move High Halfword**

**Instruction format** MOVHI imm16, reg1, reg2

**Operation** GR [reg2] ← GR [reg1] + (imm16 II $0^{16}$)

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110010RRRRR | iiiiiiiiiiiiiiii |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Adds a word data whose higher 16 bits are specified by the 16-bit immediate data and lower 16 bits are 0 to the word data of general-purpose register reg1 and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.
The flags are not affected by the addition.
Do not specify r0 as the destination register reg2.

**Remark** This instruction is used to generate the higher 16 bits of a 32-bit address.

**<Multiply instruction>**

**Multiply word by register/immediate (9-bit)**

# MUL

**Multiply Word**

**Instruction format** (1) MUL reg1, reg2, reg3
(2) MUL imm9, reg2, reg3

**Operation** (1) GR [reg3] || GR [reg2] ← GR [reg2] × GR [reg1]
(2) GR [reg3] || GR [reg2] ← GR [reg2] × sign-extend (imm9)

**Format** (1) Format XI
(2) Format XII

**Opcode**

| | 15 0 | 31 16 |
|---|---|---|
| (1) | rrrrr111111RRRRR | wwwww01000100000 |

| | 15 0 | 31 16 |
|---|---|---|
| (2) | rrrrr111111iiiii | wwwww01001IIII00 |

iiiii is the lower 5 bits of 9-bit immediate data.
IIII is the higher 4 bits of 9-bit immediate data.

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation**

★ (1) Multiplies the word data of general-purpose register reg2 by the word data of general-purpose register reg1, and stores the higher 32 bits of the result (64-bit data) in general-purpose register reg3 and the lower 32 bits in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

★ (2) Multiplies the word data of general-purpose register reg2 by a 9-bit immediate data, sign-extended to word length, and stores the higher 32 bits of the result (64-bit data) in general-purpose registers reg3 and the lower 32 bits in general-purpose register reg2.

**Remark** If the address of reg2 is the same as the address of reg3, the higher 32 bits of the result are stored in reg2 (= reg3).

★ **Caution** In the “MUL reg1, reg2, reg3” instruction, do not use registers in combinations that satisfy all the following conditions. If the instruction is executed with all the following conditions satisfied, the operation is not guaranteed.

- reg1 = reg3
- reg1 ≠ reg2
- reg1 ≠ r0
- reg3 ≠ r0

**<Multiply instruction>**

**Multiply halfword by register/immediate (5-bit)**

# MULH

**Multiply Halfword**

**Instruction format** (1) MULH reg1, reg2
(2) MULH imm5, reg2

**Operation** (1) GR [reg2] (32) ← GR [reg2] (16) × GR [reg1] (16)
(2) GR [reg2] ← GR [reg2] × sign-extend (imm5)

**Format** (1) Format I
(2) Format II

**Opcode**

| | 15 | 0 |
|---|---|---|
| (1) | rrrrr000111RRRRR | |

| | 15 | 0 |
|---|---|---|
| (2) | rrrrr010111iiiii | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** (1) Multiplies the lower halfword data of general-purpose register reg2 by the halfword data of general-purpose register reg1, and stores the result in general-purpose register reg2 as word data.
The data of general-purpose register reg1 is not affected.
Do not specify r0 as the destination register reg2.
(2) Multiplies the lower halfword data of general-purpose register reg2 by a 5-bit immediate data, sign-extended to halfword length, and stores the result in general-purpose register reg2.
Do not specify r0 as the destination register reg2.

**Remark** The higher 16 bits of general-purpose registers reg1 and reg2 are ignored in this operation.

**<Multiply instruction>**

**Multiply halfword by immediate (16-bit)**

# MULHI

**Multiply Halfword Immediate**

**Instruction format** MULHI imm16, reg1, reg2

**Operation** GR [reg2] ← GR [reg1] × imm16

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110111RRRRR | iiiiiiiiiiiiiiii |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Multiplies the lower halfword data of general-purpose register reg1 by the 16-bit immediate data, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.
Do not specify r0 as the destination register reg2.

**Remark** The higher 16 bits of general-purpose register reg1 are ignored in this operation.

**<Multiply instruction>**

**Multiply word by register/immediate (9-bit)**

# MULU

**Multiply Word Unsigned**

**Instruction format** (1) MULU reg1, reg2, reg3
(2) MULU imm9, reg2, reg3

**Operation** (1) GR [reg3] || GR [reg2] ← GR [reg2] × GR [reg1]
(2) GR [reg3] || GR [reg2] ← GR [reg2] × zero-extend (imm9)

**Format** (1) Format XI
(2) Format XII

**Opcode**

| | 15 0 | 31 16 |
|---|---|---|
| (1) | rrrrr111111RRRRR | wwwww01000100010 |

| | 15 0 | 31 16 |
|---|---|---|
| (2) | rrrrr111111iiiii | wwwww01001IIII10 |

iiiii is the lower 5 bits of 9-bit immediate data.
IIII is the higher 4 bits of 9-bit immediate data.

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation**

★ (1) Multiplies the word data of general-purpose register reg2 by the word data of general-purpose register reg1, and stores the higher 32 bits of the result (64-bit data) in general-purpose register reg3 and the lower 32 bits in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

★ (2) Multiplies the word data of general-purpose register reg2 by a 9-bit immediate data, sign-extended to word length, and stores the higher 32 bits of the result (64-bit data) in general-purpose registers reg3 and the lower 32 bits in general-purpose register reg2.

**Remark** If the address of reg2 is the same as the address of reg3, the higher 32 bits of the result are stored in reg2 (= reg3).

★ **Caution** In the “MULU reg1, reg2, reg3” instruction, do not use registers in combinations that satisfy all the following conditions. If the instruction is executed with all the following conditions satisfied, the operation is not guaranteed.

- reg1 = reg3
- reg1 ≠ reg2
- reg1 ≠ r0
- reg3 ≠ r0

**<Special instruction>**

**No operation**

# NOP

**No Operation**

**Instruction format** NOP

**Operation** Executes nothing and consumes at least one clock.

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 0000000000000000 | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Executes nothing and consumes at least one clock cycle.

**Remark** The contents of the PC are incremented by two. The opcode is the same as that of MOV r0, r0.

**<Logical operation instruction>**

**NOT**

# NOT

**Not**

**Instruction format** NOT reg1, reg2

**Operation** GR [reg2] ← NOT (GR [reg1])

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr000001RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** Logically negates (takes the 1's complement of) the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Bit manipulation instruction>**

# NOT1

**NOT bit**

**Not Bit**

**Instruction format** (1) NOT1 bit#3, disp16 [reg1]
(2) NOT1 reg2, [reg1]

**Operation** (1) adr ← GR [reg1] + sign-extend (disp16)
Z flag ← Not (Load-memory-bit (adr, bit#3))
Store-memory-bit (adr, bit#3, Z flag)
(2) adr ← GR [reg1]
Z flag ← Not (Load-memory-bit (adr, reg2))
Store-memory-bit (adr, reg2, Z flag)

**Format** (1) Format VIII
(2) Format IX

**Opcode**

| | 15 0 | 31 16 |
|---|---|---|
| (1) | 01bbb111110RRRRR | ddddddddddddddddd |

| | 15 0 | 31 16 |
|---|---|---|
| (2) | rrrrr111111RRRRR | 0000000011100010 |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | 1 if bit specified by operands = 0, 0 if bit specified by operands = 1 |
| SAT | – |

**Explanation** (1) Adds the data of general-purpose register reg1 to a 16-bit displacement, sign-extended to word length to generate a 32-bit address. Then reads the byte data referenced by the generated address, inverts the bit specified by the 3-bit bit number (0 → 1 or 1 → 0), and writes back to the original address.
(2) Reads the data of general-purpose register reg1 to generate a 32-bit address. Then reads the byte data referenced by the generated address, inverts the bit specified by the data of lower 3 bits of reg2 (0 → 1 or 1 → 0), and writes back to the original address.

**Remark** The Z flag of the PSW indicates whether the specified bit was 0 or 1 before this instruction was executed, and does not indicate the contents of the specified bit after this instruction has been executed.

**<Logical operation instruction>**

**OR**

# OR

**Or**

**Instruction format** OR reg1, reg2

**Operation** GR [reg2] ← GR [reg2] OR GR [reg1]

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr001000RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** ORs the word data of general-purpose register reg2 with the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Logical operation instruction>**

**OR immediate (16-bit)**

# ORI

**Or Immediate**

**Instruction format** ORI imm16, reg1, reg2

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

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110100RRRRR | iiiiiiiiiiiiiiii |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** ORs the word data of general-purpose register reg1 with the value of the 16-bit immediate data, zero-extended to word length, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Special instruction>**

# PREPARE

**Function prepare**

**Stack Frame Generation**

**Instruction format** (1) PREPARE list12, imm5
(2) PREPARE list12, imm5, sp/imm[Note]

**Note** sp/imm is specified by sub-opcode bits 20 and 19.

**Operation**

(1) Store-memory (sp – 4, GR [reg in list12], Word) sp ← sp – 4
repeat 1 step above until all regs in list12 is stored
sp ← sp – zero-extend (imm5)

(2) Store-memory (sp – 4, GR [reg in list12], Word) sp ← sp – 4
repeat 1 step above until all regs in list12 is stored
sp ← sp – zero-extend (imm5)
ep ← sp/imm

**Format** Format XIII

**Opcode**

(1)

| 15 … 0 | 31 … 16 |
|---|---|
| 0000011110iiiiiL | LLLLLLLLLLL00001 |

(2)

| 15 … 0 | 31 … 16 | Optional(47 to 32 or 63 to 32) |
|---|---|---|
| 0000011110iiiiiL | LLLLLLLLLLLff011 | imm16 / imm32 |

In the case of 32-bit immediate data (imm32), bits 47 to 32 are the lower 16 bits of imm32, bits 63 to 48 are the higher 16 bits of imm32.

ff = 00: load sp to ep
ff = 01: load 16-bit immediate data (bits 47 to 32), sign-extended, to ep
ff = 10: load 16-bit immediate data (bits 47 to 32), logically shifted left by 16, to ep
ff = 11: load 32-bit immediate data (bits 63 to 32) to ep

LLLLLLLLLLL indicates the bit value corresponding to the register list (list12) (for example, “L” of bit 21 in an opcode indicates the value of bit 21 of list12). list12 is a 32-bit register list defined as follows.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 … 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| r24 | r25 | r26 | r27 | r20 | r21 | r22 | r23 | r28 | r29 | r31 | – | r30 |

Bits 31 to 21 and bit 0 correspond to each bit of the general-purpose registers (r21 to r31). The register corresponding to the set bit (1) is specified as the manipulation target. For example, when r20 and r30 are specified, list12 values are as follows (the set values of bits 20 to 1 to which registers do not correspond can be 0 or 1 (don’t care)).

- If the values of all the bits to which registers do not correspond are set to 0: 08000001H
- If the values of all the bits to which registers do not correspond are set to 1: 081FFFFFH

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation**

(1) Pushes (subtracts 4 from sp and stores the data in that address) the general-purpose registers listed in list12. Then subtracts the data of 5-bit immediate imm5, logically shifted left by 2 and zero-extended to word length, from sp.

(2) Pushes (subtracts 4 from sp and stores the data in that address) the general-purpose registers listed in list12. Then subtracts the data of 5-bit immediate imm5, logically shifted left by 2 and zero-extended to word length, from sp.
Next, loads the data specified by the 3rd operand (sp/imm) to ep.

**Remark**

The general-purpose registers in list12 are stored in the upward direction (r20, r21, ... r31).
The 5-bit immediate imm5 is used to make a stack frame for auto variables and temporary data.
The lower 2 bits of the address specified by sp are always masked by 0 even if misaligned access is enabled.
If an interrupt occurs before updating sp, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction (sp and ep will retain their original values prior to the start of execution).

**Caution**

If an interrupt is generated during instruction execution, due to manipulation of the stack, the execution of that instruction may stop after the read/write cycle and register value rewriting are complete.

**<Special instruction>**

# RETI

**Return from trap or interrupt**

**Return from Trap or Interrupt**

**Instruction format** RETI

**Operation**

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

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 0000011111100000 | 0000000101000000 |

**Flag**

| | |
|---|---|
| CY | Value read from FEPSW or EIPSW is restored. |
| OV | Value read from FEPSW or EIPSW is restored. |
| S | Value read from FEPSW or EIPSW is restored. |
| Z | Value read from FEPSW or EIPSW is restored. |
| SAT | Value read from FEPSW or EIPSW is restored. |

**Explanation** This instruction reads the restored PC and PSW from the appropriate system register, and operation returns from a software exception or interrupt routine. The operations of this instruction are as follows.

(1) If the EP flag of the PSW is 1, the restored PC and PSW are read from EIPC and EIPSW, regardless of the status of the NP flag of the PSW.
If the EP flag of the PSW is 0 and the NP flag of the PSW is 1, the restored PC and PSW are read from FEPC and FEPSW.
If the EP flag of the PSW is 0 and the NP flag of the PSW is 0, the restored PC and PSW are read from EIPC and EIPSW.

(2) Once the restored PC and PSW values are set to the PC and PSW, the operation returns to the address immediately before the trap or interrupt occurred.

**Caution** When returning from a non-maskable interrupt or software exception routine using the RETI instruction, the NP and EP flags of the PSW must be set accordingly to restore the PC and PSW.

- When returning from a non-maskable interrupt routine using the RETI instruction:
  NP = 1 and EP = 0
- When returning from a software exception routine using the RETI instruction:
  EP = 1

Use the LDSR instruction for setting the flags.
Interrupts are not acknowledged in the latter half of the ID stage during LDSR execution because of the operation of the interrupt controller.

**<Logical operation instruction>**

**Shift arithmetic right by register/immediate (5-bit)**

# SAR

**Shift Arithmetic Right**

**Instruction format** (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

**Format** (1) Format IX
(2) Format II

**Opcode**

(1)

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | 0000000010100000 |

(2)

| 15 0 |
|---|
| rrrrr010101iiiii |

**Flag**

| | |
|---|---|
| CY | 1 if the bit shifted out last is 1; otherwise, 0.<br>However, if the number of shifts is 0, the result is 0. |
| OV | 0 |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation**

(1) Arithmetically shifts the word data of general-purpose register reg2 to the right by 'n' positions, where 'n' is a value from 0 to +31, specified by the lower 5 bits of general-purpose register reg1 (after the shift, the MSB prior to shift execution is copied and set as the new MSB value), and then writes the result to general-purpose register reg2. If the number of shifts is 0, general-purpose register reg2 retains the same value prior to instruction execution. The data of general-purpose register reg1 is not affected.

(2) Arithmetically shifts the word data of general-purpose register reg2 to the right by 'n' positions, where 'n' is a value from 0 to +31, specified by the 5-bit immediate data, zero-extended to word length (after the shift, the MSB prior to shift execution is copied and set as the new MSB value), and then writes the result to general-purpose register reg2. If the number of shifts is 0, general-purpose register reg2 retains the same value prior to instruction execution.

**<Logical operation instruction>**

**Shift and set flag condition**

# SASF

**Shift and Set Flag Condition**

**Instruction format** SASF cccc, reg2

**Operation** if conditions are satisfied
then GR [reg2] ← (GR [reg2] Logically shift left by 1) OR 00000001H
else GR [reg2] ← (GR [reg2] Logically shift left by 1) OR 00000000H

**Format** Format IX

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr1111110cccc | 0000001000000000 |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** General-purpose register reg2 is logically shifted left by 1, and its LSB is set to 1 if the condition specified by condition code “cccc” is satisfied; otherwise, general-purpose register reg2 is logically shifted left by 1, and its LSB is set to 0.
One of the codes shown in **Table 5-5 Condition Codes** should be specified as the condition code “cccc”.

**Remark** See SETF instruction.

**<Saturated operation instruction>**

**Saturated add register/immediate (5-bit)**

# SATADD

**Saturated Add**

**Instruction format** (1) SATADD reg1, reg2
(2) SATADD imm5, reg2

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

**Format** (1) Format I
(2) Format II

**Opcode**

| | 15 | 0 |
|---|---|---|
| (1) | rrrrr000110RRRRR | |

| | 15 | 0 |
|---|---|---|
| (2) | rrrrr010001iiiii | |

**Flag**

| | |
|---|---|
| CY | 1 if a carry occurs from MSB; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of the saturated operation is negative; otherwise, 0. |
| Z | 1 if the result of the saturated operation is 0; otherwise, 0. |
| SAT | 1 if OV = 1; otherwise, not affected. |

**Explanation**

(1) Adds the word data of general-purpose register reg1 to the word data of general-purpose register reg2, and stores the result in general-purpose register reg2. However, if the result exceeds the maximum positive value 7FFFFFFFH, 7FFFFFFFH is stored in reg2; if the result exceeds the maximum negative value 80000000H, 80000000H is stored in reg2. The SAT flag is set to 1. The data of general-purpose register reg1 is not affected.
Do not specify r0 as the destination register reg2.

(2) Adds a 5-bit immediate data, sign-extended to word length, to the word data of general-purpose register reg2, and stores the result in general-purpose register reg2. However, if the result exceeds the maximum positive value 7FFFFFFFH, 7FFFFFFFH is stored in reg2; if the result exceeds the maximum negative value 80000000H, 80000000H is stored in reg2. The SAT flag is set to 1.
Do not specify r0 as the destination register reg2.

**Remark** The SAT flag is a cumulative flag. Once the result of the saturated operation instruction has been saturated, this flag is set to 1 and is not cleared to 0 even if the result of the subsequent operation is not saturated.
Even if the SAT flag is set to 1, the saturated operation instruction is executed normally.

**Caution** To clear the SAT flag to 0, load data to the PSW by using the LDSR instruction.

**<Saturated operation instruction>**

**Saturated subtract**

# SATSUB

**Saturated Subtract**

**Instruction format** SATSUB reg1, reg2

**Operation** GR [reg2] ← saturated (GR [reg2] – GR [reg1])

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr000101RRRRR | |

**Flag**

| | |
|---|---|
| CY | 1 if a borrow to MSB occurs; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of the saturated operation is negative; otherwise, 0. |
| Z | 1 if the result of the saturated operation is 0; otherwise, 0. |
| SAT | 1 if OV = 1; otherwise, not affected. |

**Explanation** Subtracts the word data of general-purpose register reg1 from the word data of general-purpose register reg2, and stores the result in general-purpose register reg2. However, if the result exceeds the maximum positive value 7FFFFFFFH, 7FFFFFFFH is stored in reg2; if the result exceeds the maximum negative value 80000000H, 80000000H is stored in reg2. The SAT flag is set to 1. The data of general-purpose register reg1 is not affected.
Do not specify r0 as the destination register reg2.

**Remark** The SAT flag is a cumulative flag. Once the result of the operation of the saturated operation instruction has been saturated, this flag is set to 1 and is not cleared to 0 even if the result of the subsequent operations is not saturated.
Even if the SAT flag is set to 1, the saturated operation instruction is executed normally.

**Caution** To clear the SAT flag to 0, load data to the PSW by using the LDSR instruction.

**<Saturated operation instruction>**

**Saturated subtract immediate**

# SATSUBI

**Saturated Subtract Immediate**

**Instruction format** SATSUBI imm16, reg1, reg2

**Operation** GR [reg2] ← saturated (GR [reg1] – sign-extend (imm16))

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110011RRRRR | iiiiiiiiiiiiiiii |

**Flag**

CY 1 if a borrow to MSB occurs; otherwise, 0.
OV 1 if overflow occurs; otherwise, 0.
S 1 if the result of the saturated operation is negative; otherwise, 0.
Z 1 if the result of the saturated operation is 0; otherwise, 0.
SAT 1 if OV = 1; otherwise, not affected.

**Explanation** Subtracts the 16-bit immediate data, sign-extended to word length, from the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. However, if the result exceeds the maximum positive value 7FFFFFFFH, 7FFFFFFFH is stored in reg2; if the result exceeds the maximum negative value 80000000H, 80000000H is stored in reg2. The SAT flag is set to 1. The data of general-purpose register reg1 is not affected.
Do not specify r0 as the destination register reg2.

**Remark** The SAT flag is a cumulative flag. Once the result of the operation of the saturated operation instruction has been saturated, this flag is set to 1 and is not cleared to 0 even if the result of the subsequent operations is not saturated.
Even if the SAT flag is set to 1, the saturated operation instruction is executed normally.

**Caution** To clear the SAT flag to 0, load data to the PSW by using the LDSR instruction.

**<Saturated operation instruction>**

**Saturated subtract reverse**

# SATSUBR

**Saturated Subtract Reverse**

**Instruction format** SATSUBR reg1, reg2

**Operation** GR [reg2] ← saturated (GR [reg1] – GR [reg2])

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr000100RRRRR | |

**Flag**

| | |
|---|---|
| CY | 1 if a borrow to MSB occurs; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of the saturated operation is negative; otherwise, 0. |
| Z | 1 if the result of the saturated operation is 0; otherwise, 0. |
| SAT | 1 if OV = 1; otherwise, not affected. |

**Explanation** Subtracts the word data of general-purpose register reg2 from the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. However, if the result exceeds the maximum positive value 7FFFFFFFH, 7FFFFFFFH is stored in reg2; if the result exceeds the maximum negative value 80000000H, 80000000H is stored in reg2. The SAT flag is set to 1. The data of general-purpose register reg1 is not affected.
Do not specify r0 as the destination register reg2.

**Remark** The SAT flag is a cumulative flag. Once the result of the operation of the saturated operation instruction has been saturated, this flag is set to 1 and is not cleared to 0 even if the result of the subsequent operations is not saturated.
Even if the SAT flag is set to 1, the saturated operation instruction is executed normally.

**Caution** To clear the SAT flag to 0, load data to the PSW by using the LDSR instruction.

**<Bit manipulation instruction>**

# SET1

**Set bit**

**Set Bit**

**Instruction format**

(1) SET1 bit#3, disp16 [reg1]
(2) SET1 reg2, [reg1]

**Operation**

(1) adr ← GR [reg1] + sign-extend (disp16)
Z flag ← Not (Load-memory-bit (adr, bit#3))
Store-memory-bit (adr, bit#3, 1)
(2) adr ← GR [reg1]
Z flag ← Not (Load-memory-bit (adr, reg2))
Store-memory-bit (adr, reg2, 1)

**Format**

(1) Format VIII
(2) Format IX

**Opcode**

| | 15 0 | 31 16 |
|---|---|---|
| (1) | 00bbb111110RRRRR | ddddddddddddddddd |

| | 15 0 | 31 16 |
|---|---|---|
| (2) | rrrrr111111RRRRR | 0000000011100000 |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | 1 if bit specified by operands = 0, 0 if bit specified by operands = 1 |
| SAT | – |

**Explanation**

(1) Adds the 16-bit displacement, sign-extended to word length, to the data of general-purpose register reg1 to generate a 32-bit address. Then reads the byte data referenced by the generated address, sets the bit specified by the 3-bit bit number to 1, and writes back to the original address.
(2) Reads the data of general-purpose register reg1 to generate a 32-bit address. Then reads the byte data referenced by the generated address, sets the bit specified by the data of lower 3 bits of reg2 to 1, and writes back to the original address.

**Remark**

The Z flag of the PSW indicates whether the specified bit was 0 or 1 before this instruction was executed, and does not indicate the content of the specified bit after this instruction has been executed.

**<Arithmetic operation instruction>**

**Set flag condition**

# SETF

**Set Flag Condition**

**Instruction format** SETF cccc, reg2

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

**Format** Format IX

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr1111110cccc | 0000000000000000 |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** General-purpose register reg2 is set to 1 if the condition specified by condition code “cccc” is satisfied; otherwise, 0 is stored in the register. One of the codes shown in **Table 5-5 Condition Codes** should be specified as the condition code “cccc”.

**Remark** Here are some examples of using this instruction.

(1) Translation of two or more condition clauses
If A of the statement “if (A)” in C language consists of two or more condition clauses ($a_1$, $a_2$, $a_3$, and so on), it is usually translated to a sequence of if ($a_1$) then, if ($a_2$) then. The object code executes a “conditional branch” by checking the result of evaluation equivalent to $a_n$. Since a pipeline processor takes more time to execute “condition judgment” + “branch” than to execute an ordinary operation, the result of evaluating each condition clause if ($a_n$) is stored in register Ra. By performing a logical operation to $Ra_n$ after all the condition clauses have been evaluated, the delay due to the pipeline can be prevented.

(2) Double-length operation
To execute a double-length operation such as Add with Carry, the result of the CY flag can be stored in general-purpose register reg2. Therefore, a carry from the lower bits can be expressed as a numeric value.

**Table 5-5. Condition Codes**

| Condition Code (cccc) | Condition Name | Condition 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 (unconditional) |
| 1101 | SA | SAT = 1 |
| 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 |

**<Logical operation instruction>**

**Shift logical left by register/immediate (5-bit)**

# SHL

**Shift Logical Left**

**Instruction format** (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 IX
(2) Format II

**Opcode**

| | 15 | 0 | 31 | 16 |
|---|---|---|---|---|
| (1) | rrrrr111111RRRRR | | 0000000011000000 | |

| | 15 | 0 |
|---|---|---|
| (2) | rrrrr010110iiiii | |

**Flag**

| | |
|---|---|
| CY | 1 if the bit shifted out last is 1; otherwise, 0.<br>However, if the number of shifts is 0, the result is 0. |
| OV | 0 |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation**

(1) Logically shifts the word data of general-purpose register reg2 to the left by ‘n’ positions, where ‘n’ is a value from 0 to +31, specified by the lower 5 bits of general-purpose register reg1 (0 is shifted to the LSB side), and then writes the result to general-purpose register reg2. If the number of shifts is 0, general-purpose register reg2 retains the same value prior to instruction execution. The data of general-purpose register reg1 is not affected.

(2) Logically shifts the word data of general-purpose register reg2 to the left by ‘n’ positions, where ‘n’ is a value from 0 to +31, specified by the 5-bit immediate data, zero-extended to word length (0 is shifted to the LSB side), and then writes the result to general-purpose register reg2. If the number of shifts is 0, general-purpose register reg2 retains the value prior to instruction execution.

**<Logical operation instruction>**

**Shift logical right by register/immediate (5-bit)**

# SHR

**Shift Logical Right**

**Instruction format**
(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 IX
(2) Format II

**Opcode**

(1)

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | 0000000010000000 |

(2)

| 15 0 |
|---|
| rrrrr010100iiiii |

**Flag**

| | |
|---|---|
| CY | 1 if the bit shifted out last is 1; otherwise, 0.<br>However, if the number of shifts is 0, the result is 0. |
| OV | 0 |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation**
(1) Logically shifts the word data of general-purpose register reg2 to the right by ‘n’ positions where ‘n’ is a value from 0 to +31, specified by the lower 5 bits of general-purpose register reg1 (0 is shifted to the MSB side). This instruction then writes the result to general-purpose register reg2. If the number of shifts is 0, general-purpose register reg2 retains the same value prior to instruction execution. The data of general-purpose register reg1 is not affected.
(2) Logically shifts the word data of general-purpose register reg2 to the right by ‘n’ positions, where ‘n’ is a value from 0 to +31, specified by the 5-bit immediate data, zero-extended to word length (0 is shifted to the MSB side). This instruction then writes the result to general-purpose register reg2. If the number of shifts is 0, general-purpose register reg2 retains the same value prior to instruction execution.

**<Load instruction>**

**Short format load byte**

# SLD.B

**Load**

**Instruction format** SLD.B disp7 [ep], reg2

**Operation**
adr ← ep + zero-extend (disp7)
GR [reg2] ← sign-extend (Load-memory (adr, Byte))

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr0110ddddddd | |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 7-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Byte data is read from the generated address, sign-extended to word length, and stored in reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**Caution** (1) If an interrupt is generated during instruction execution, the execution of that instruction may stop after the end of the read/write cycle. In this case, the instruction is re-executed after returning from the interrupt. Therefore, except in cases when clearly no interrupt is generated, the LD instruction should be used for accessing I/O, FIFO types, or other resources whose status is changed by the read cycle (the bus cycle is not re-executed even if an interrupt is generated while the LD or store instruction is being executed).

★ (2) For the restriction on the conflict between the sld instruction and an interrupt request, refer to **APPENDIX A NOTES**.

**<Load instruction>**

**Short format load byte unsigned**

# SLD.BU

**Load**

**Instruction format** SLD.BU disp4 [ep], reg2

**Operation**
adr ← ep + zero-extend (disp4)
GR [reg2] ← zero-extend (Load-memory (adr, Byte))

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr0000110dddd | |

rrrrr must not be 00000.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 4-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Byte data is read from the generated address, zero-extended to word length, and stored in reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**Caution** (1) If an interrupt is generated during instruction execution, the execution of that instruction may stop after the end of the read/write cycle. In this case, the instruction is re-executed after returning from the interrupt. Therefore, except in cases when clearly no interrupt is generated, the LD instruction should be used for accessing I/O, FIFO types, or other resources whose status is changed by the read cycle (the bus cycle is not re-executed even if an interrupt is generated while the LD or store instruction is being executed).

★ (2) For the restriction on the conflict between the sld instruction and an interrupt request, refer to **APPENDIX A NOTES**.

**<Load instruction>**

**Short format load halfword**

# SLD.H

**Load**

**Instruction format** SLD.H disp8 [ep], reg2

**Operation**
adr ← ep + zero-extend (disp8)
GR [reg2] ← sign-extend (Load-memory (adr, Halfword))

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr1000ddddddd | |

ddddddd is the higher 7 bits of disp8.

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Adds 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Halfword data is read from the generated address, sign-extended to word length, and stored in reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

Caution (1) The result of adding the element pointer and the 8-bit displacement zero-extended to word length can be of two types depending on the type of data to be accessed (halfword, word) and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled in type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

Also, if an interrupt is generated during instruction execution, the execution of that instruction may stop after the end of the read/write cycle. In this case, the instruction is re-executed after returning from the interrupt. Therefore, except in cases when clearly no interrupt is generated, the LD instruction should be used for accessing I/O, FIFO types, or other resources whose status is changed by the read cycle (the bus cycle is not re-executed even if an interrupt is generated while the LD or store instruction is being executed).

★ (2) For the restriction on the conflict between the sld instruction and an interrupt request, refer to **APPENDIX A NOTES**.

**<Load instruction>**

**Short format load halfword unsigned**

# SLD.HU

**Load**

**Instruction format** SLD.HU disp5 [ep], reg2

**Operation**
adr ← ep + zero-extend (disp5)
GR [reg2] ← zero-extend (Load-memory (adr, Halfword))

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr0000111dddd | |

dddd is the higher 4 bits of disp5. rrrrr must not be 00000.

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Adds 5-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Halfword data is read from the generated address, zero-extended to word length, and stored in reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**Caution** (1) The result of adding the element pointer and the 8-bit displacement zero-extended to word length can be of two types depending on the type of data to be accessed (halfword, word) and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled in type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

Also, if an interrupt is generated during instruction execution, the execution of that instruction may stop after the end of the read/write cycle. In this case, the instruction is re-executed after returning from the interrupt. Therefore, except in cases when clearly no interrupt is generated, the LD instruction should be used for accessing I/O, FIFO types, or other resources whose status is changed by the read cycle (the bus cycle is not re-executed even if an interrupt is generated while the LD or store instruction is being executed).

★ (2) For the restriction on the conflict between the sld instruction and an interrupt request, refer to **APPENDIX A NOTES**.

**<Load instruction>**

**Short format load word**

# SLD.W

**Load**

**Instruction format** SLD.W disp8 [ep], reg2

**Operation**
adr ← ep + zero-extend (disp8)
GR [reg2] ← Load-memory (adr, Word)

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr1010ddddd0 | |

dddddd is the higher 6 bits of disp8.

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Adds 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Word data is read from the generated address and stored in reg2.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**Caution** (1) The result of adding the element pointer and the 8-bit displacement zero-extended to word length can be of two types depending on the type of data to be accessed (halfword, word) and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled in type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

Also, if an interrupt is generated during instruction execution, the execution of that instruction may stop after the end of the read/write cycle. In this case, the instruction is re-executed after returning from the interrupt. Therefore, except in cases when clearly no interrupt is generated, the LD instruction should be used for accessing I/O, FIFO types, or other resources whose status is changed by the read cycle (the bus cycle is not re-executed even if an interrupt is generated while the LD or store instruction is being executed).

★ (2) For the restriction on the conflict between the sld instruction and an interrupt request, refer to **APPENDIX A NOTES**.

**<Store instruction>**

**Short format store byte**

# SST.B

**Store**

**Instruction format** SST.B reg2, disp7 [ep]

**Operation**
adr ← ep + zero-extend (disp7)
Store-memory (adr, GR [reg2], Byte)

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr0111ddddddd | |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 7-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address, and stores the data of the lowest byte of reg2 in the generated address.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Store instruction>**

**Short format store halfword**

# SST.H

**Store**

**Instruction format** SST.H reg2, disp8 [ep]

**Operation**
adr ← ep + zero-extend (disp8)
Store-memory (adr, GR [reg2], Halfword)

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr1001ddddddd | |

ddddddd is the higher 7 bits of disp8.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address, and stores the lower halfword data of reg2 in the generated address.

**Caution** The result of adding the element pointer and the 8-bit displacement zero-extended to word length can be of two types depending on the type of data to be accessed (halfword, word) and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)
  (when misaligned access is enabled in type D, E, and F products) ★

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Store instruction>**

**Short format store word**

# SST.W

**Store**

**Instruction format** SST.W reg2, disp8 [ep]

**Operation**
adr ← ep + zero-extend (disp8)
Store-memory (adr, GR [reg2], Word)

**Format** Format IV

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr1010dddddd1 | |

dddddd is the higher 6 bits of disp8.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address, and stores the word data of reg2 in the generated address.

**Caution** The result of adding the element pointer and the 8-bit displacement zero-extended to word length can be of two types depending on the type of data to be accessed (halfword, word) and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)
  (when misaligned access is enabled in type D, E, and F products) ★

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Store instruction>**

**Store byte**

# ST.B

**Store**

**Instruction format** ST.B reg2, disp16 [reg1]

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
Store-memory (adr, GR [reg2], Byte)

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111010RRRRR | dddddddddddddddd |

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 16-bit displacement, sign-extended to word length, to the data of general-purpose register reg1 to generate a 32-bit address, and stores the lowest byte data of general-purpose register reg2 in the generated address.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Store instruction>**

**Store halfword**

# ST.H

**Store**

**Instruction format** ST.H reg2, disp16 [reg1]

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
Store-memory (adr, GR [reg2], Halfword)

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111011RRRRR | ddddddddddddddd0 |

ddddddddddddddd is the higher 15 bits of disp16.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 16-bit displacement, sign-extended to word length, to the data of general-purpose register reg1 to generate a 32-bit address, and stores the lower halfword data of general-purpose register reg2 in the generated address.

**Caution** The result of adding the data of general-purpose register reg1 and the 16-bit displacement sign-extended to word length can be of two types depending on the type of data to be accessed (halfword, word), and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)

★ (when misaligned access is enabled in type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Store instruction>**

**Store word**

# ST.W

**Store**

**Instruction format** ST.W reg2, disp16 [reg1]

**Operation**
adr ← GR [reg1] + sign-extend (disp16)
Store-memory (adr, GR [reg2], Word)

**Format** Format VII

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111011RRRRR | ddddddddddddddd1 |

ddddddddddddddd is the higher 15 bits of disp16.

**Flag**
CY –
OV –
S –
Z –
SAT –

**Explanation** Adds 16-bit displacement, sign-extended to word length, to the data of general-purpose register reg1 to generate a 32-bit address, and stores the word data of general-purpose register reg2 in the generated address.

**Caution** The result of adding the data of general-purpose register reg1 and the 16-bit displacement sign-extended to word length can be of two types depending on the type of data to be accessed (halfword, word), and the misalign mode setting.

- Lower bits are masked by 0 and address is generated (when misaligned access is disabled)
- Lower bits are not masked and address is generated (when misaligned access is enabled)
  ★ (when misaligned access is enabled in type D, E, and F products)

For details on misaligned access, see **3.3 Data Alignment**.

**Remark** If an interrupt occurs during instruction execution, execution is aborted, and the interrupt is serviced. Upon returning from the interrupt, the execution is restarted from the beginning, with the return address being the address of this instruction.

★ [For type D, E, and F products]
Depending on the resource to be accessed (internal ROM, internal RAM, on-chip peripheral I/O, external memory), the bus cycle may be switched (this will not occur if the same resource is accessed).

★ [For type A, B, and C products]
The bus cycle sequence for accessing the different resources connected to each bus (VFB, VDB, VSB, NPB, instruction cache bus, data cache bus) may be switched (this will not occur if the same bus is accessed).

**<Special instruction>**

# STSR

**Store contents of system register**

**Store Contents of System Register**

**Instruction format** STSR regID, reg2

**Operation** GR [reg2] ← SR [regID]

**Format** Format IX

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | 0000000001000000 |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Stores the contents of a system register specified by a system register number (regID) in general-purpose register reg2. The contents of the system register are not affected.

**Caution** The system register number regID is a number which identifies a system register. Accessing a system register which is reserved is prohibited and will lead to undefined results.

**<Logical operation instruction>**

**Subtract**

# SUB

**Subtract**

**Instruction format** SUB reg1, reg2

**Operation** GR [reg2] ← GR [reg2] – GR [reg1]

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr001101RRRRR | |

**Flag**

| | |
|---|---|
| CY | 1 if a borrow to MSB occurs; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** Subtracts the word data of general-purpose register reg1 from the word data of general-purpose register reg2, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Logical operation instruction>**

**Subtract reverse**

# SUBR

**Subtract Reverse**

**Instruction format** SUBR reg1, reg2

**Operation** GR [reg2] ← GR [reg1] – GR [reg2]

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr001100RRRRR | |

**Flag**

| | |
|---|---|
| CY | 1 if a borrow to MSB occurs; otherwise, 0. |
| OV | 1 if overflow occurs; otherwise, 0. |
| S | 1 if the result of an operation is negative; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** Subtracts the word data of general-purpose register reg2 from the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Special instruction>**

**Jump with table look up**

# SWITCH

**Jump with Table Look Up**

**Instruction format** SWITCH reg1

**Operation**
adr ← (PC + 2) + (GR [reg1] logically shift left by 1)
PC ← (PC + 2) + (sign-extend (Load-memory (adr, Halfword))) logically shift left by 1

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 00000000010RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation**

<1> Adds the table entry address (address following SWITCH instruction) and data of general-purpose register reg1 logically shifted left by 1, and generates 32-bit table entry address.

<2> Loads the halfword data pointed to the address generated in <1>.

<3> Sign-extends the loaded halfword data to word length, and adds the table entry address after logically shifting it left by 1 bit (next address following SWITCH instruction) to generate a 32-bit target address.

<4> Then jumps to the target address generated in <3>.

**<Logical operation instruction>**

**Sign extend byte**

# SXB

**Sign Extend Byte**

**Instruction format** SXB reg1

**Operation** GR [reg1] ← sign-extend (GR [reg1] (7:0))

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 00000000101RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Sign-extends the lowest byte of general-purpose register reg1 to word length.

**<Logical operation instruction>**

**Sign extend halfword**

# SXH

**Sign Extend Halfword**

**Instruction format** SXH reg1

**Operation** GR [reg1] ← sign-extend (GR [reg1] (15:0))

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 00000000111RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Sign-extends the lower halfword of general-purpose register reg1 to word length.

**<Special instruction>**

**Trap**

# TRAP

**Trap**

**Instruction format** TRAP vector

★ **Operation**

EIPC ← PC + 4 (restored PC)
EIPSW ← PSW
ECR.EICC ← exception code (40H to 4FH, 50H to 5FH)
PSW.EP ← 1
PSW.ID ← 1
PC ← 00000040H (vector = 00H to 0FH (exception code: 40H to 4FH))
00000050H (vector = 10H to 1FH (exception code: 50H to 5FH))

**Format** Format X

**Opcode**

| 15 0 | 31 16 |
|---|---|
| 00000111111iiiii | 0000000100000000 |

**Flag**

CY –
OV –
S –
Z –
SAT –

**Explanation** Saves the restored PC and PSW to EIPC and EIPSW, respectively; sets the exception code (EICC of ECR) and the flags of the PSW (sets the EP and ID flags to 1); jumps to the handler address corresponding to the trap vector (00H to 1FH) specified by “vector”, and starts exception processing.
The flags of the PSW other than the EP and ID flags are not affected.
The restored PC is the address of the instruction following the TRAP instruction.

**<Logical operation instruction>**

# TST

**Test**

**Test**

**Instruction format** TST reg1, reg2

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

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr001011RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** ANDs the word data of general-purpose register reg2 with the word data of general-purpose register reg1. The result is not stored, and only the flags are changed. The data of general-purpose registers reg1 and reg2 is not affected.

**<Bit manipulation instruction>**

# TST1

**Test bit**

**Test Bit**

**Instruction format** (1) TST1 bit#3, disp16 [reg1]
(2) TST1 reg2, [reg1]

**Operation** (1) adr ← GR [reg1] + sign-extend (disp16)
Z flag ← Not (Load-memory-bit (adr, bit#3))
(2) adr ← GR [reg1]
Z flag ← Not (Load-memory-bit (adr, reg2))

**Format** (1) Format VIII
(2) Format IX

**Opcode**

(1)

| 15 0 | 31 16 |
|---|---|
| 11bbb111110RRRRR | dddddddddddddddd |

(2)

| 15 0 | 31 16 |
|---|---|
| rrrrr111111RRRRR | 0000000011100110 |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | 1 if bit specified by operands = 0, 0 if bit specified by operands = 1 |
| SAT | – |

**Explanation** (1) Adds the data of general-purpose register reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Performs a test on the bit specified by the 3-bit bit number, at the byte data location referenced by the generated address. If the specified bit is 0, the Z flag of the PSW is set to 1; if the bit is 1, the Z flag is cleared to 0. The byte data, including the specified bit, is not affected.

(2) Reads the data of general-purpose register reg1 to generate a 32-bit address. Performs a test on the bit specified by the lower 3 bits of reg2, at the byte data location referenced by the generated address. If the specified bit is 0, the Z flag of the PSW is set to 1; if the bit is 1, the Z flag is cleared to 0. The byte data, including the specified bit, is not affected.

**<Logical operation instruction>**

**Exclusive OR**

# XOR

**Exclusive Or**

**Instruction format** XOR reg1, reg2

**Operation** GR [reg2] ← GR [reg2] XOR GR [reg1]

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| rrrrr001001RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** Exclusively ORs the word data of general-purpose register reg2 with the word data of general-purpose register reg1, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Logical operation instruction>**

**Exclusive OR immediate (16-bit)**

# XORI

**Exclusive Or Immediate**

**Instruction format** XORI imm16, reg1, reg2

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

**Format** Format VI

**Opcode**

| 15 0 | 31 16 |
|---|---|
| rrrrr110101RRRRR | iiiiiiiiiiiiiiii |

**Flag**

| | |
|---|---|
| CY | – |
| OV | 0 |
| S | 1 if the MSB of the word data of the operation result is 1; otherwise, 0. |
| Z | 1 if the result of an operation is 0; otherwise, 0. |
| SAT | – |

**Explanation** Exclusively ORs the word data of general-purpose register reg1 with a 16-bit immediate data, zero-extended to word length, and stores the result in general-purpose register reg2. The data of general-purpose register reg1 is not affected.

**<Logical operation instruction>**

**Zero extend byte**

# ZXB

**Zero Extend Byte**

**Instruction format** ZXB reg1

**Operation** GR [reg1] ← zero-extend (GR [reg1] (7:0))

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 00000000100RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Zero-extends the lowest byte of general-purpose register reg1 to word length.

**<Logical operation instruction>**

**Zero extend halfword**

# ZXH

**Zero Extend Halfword**

**Instruction format** ZXH reg1

**Operation** GR [reg1] ← zero-extend (GR [reg1] (15:0))

**Format** Format I

**Opcode**

| 15 | 0 |
|---|---|
| 00000000110RRRRR | |

**Flag**

| | |
|---|---|
| CY | – |
| OV | – |
| S | – |
| Z | – |
| SAT | – |

**Explanation** Zero-extends the lower halfword of general-purpose register reg1 to word length.

## 5.4 Number of Instruction Execution Clock Cycles

A list of the number of instruction execution clocks when the internal ROM or internal RAM is used is shown below. The number of instruction execution clock cycles differs depending on the combination of instructions. For details, see **CHAPTER 8 PIPELINE**.

Table 5-6 shows the number of instruction execution clock cycles.

**Table 5-6. List of Number of Instruction Execution Clock Cycles (1/3)**

| Type of Instruction | Mnemonic | Operand | Byte | Number of Execution Clocks | | |
|---|---|---|---|---|---|---|
| | | | | i | r | l |
| Load instructions | LD.B | disp16 [reg1] , reg2 | 4 | 1 | 1 | **Note 1** |
| | LD.H | disp16 [reg1] , reg2 | 4 | 1 | 1 | **Note 1** |
| | LD.W | disp16 [reg1] , reg2 | 4 | 1 | 1 | **Note 1** |
| | LD.BU | disp16 [reg1] , reg2 | 4 | 1 | 1 | **Note 1** |
| | LD.HU | disp16 [reg1] , reg2 | 4 | 1 | 1 | **Note 1** |
| | SLD.B | disp7 [ep] , reg2 | 2 | 1 | 1 | **Note 2** |
| | SLD.BU | disp4 [ep] , reg2 | 2 | 1 | 1 | **Note 2** |
| | SLD.H | disp8 [ep] , reg2 | 2 | 1 | 1 | **Note 2** |
| | SLD.HU | disp5 [ep] , reg2 | 2 | 1 | 1 | **Note 2** |
| | SLD.W | disp8 [ep] , reg2 | 2 | 1 | 1 | **Note 2** |
| Store instructions | ST.B | reg2, disp16 [reg1] | 4 | 1 | 1 | 1 |
| | ST.H | reg2, disp16 [reg1] | 4 | 1 | 1 | 1 |
| | ST.W | reg2, disp16 [reg1] | 4 | 1 | 1 | 1 |
| | SST.B | reg2, disp7 [ep] | 2 | 1 | 1 | 1 |
| | SST.H | reg2, disp8 [ep] | 2 | 1 | 1 | 1 |
| | SST.W | reg2, disp8 [ep] | 2 | 1 | 1 | 1 |
| Multiply instructions | MUL | reg1, reg2, reg3 | 4 | 1 | 2 Note 3 | 2 |
| | MUL | imm9, reg2, reg3 | 4 | 1 | 2 Note 3 | 2 |
| | MULH | reg1, reg2 | 2 | 1 | 1 | 2 |
| | MULH | imm5, reg2 | 2 | 1 | 1 | 2 |
| | MULHI | imm16, reg1, reg2 | 4 | 1 | 1 | 2 |
| | MULU | reg1, reg2, reg3 | 4 | 1 | 2 Note 3 | 2 |
| | MULU | imm9, reg2, reg3 | 4 | 1 | 2 Note 3 | 2 |
| Arithmetic operation instructions | ADD | reg1, reg2 | 2 | 1 | 1 | 1 |
| | ADD | imm5, reg2 | 2 | 1 | 1 | 1 |
| | ADDI | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | CMOV | cccc, reg1, reg2, reg3 | 4 | 1 | 1 | 1 |
| | CMOV | cccc, imm5, reg2, reg3 | 4 | 1 | 1 | 1 |
| | CMP | reg1, reg2 | 2 | 1 | 1 | 1 |
| | CMP | imm5, reg2 | 2 | 1 | 1 | 1 |

**Table 5-6. List of Number of Instruction Execution Clock Cycles (2/3)**

| Type of Instruction | Mnemonic | Operand | Byte | Number of Execution Clocks | | |
|---|---|---|---|---|---|---|
| | | | | i | r | l |
| Arithmetic operation instructions | DIV | reg1, reg2, reg3 | 4 | 35 | 35 | 35 |
| | DIVH | reg1, reg2 | 2 | 35 | 35 | 35 |
| | DIVH | reg1, reg2, reg3 | 4 | 35 | 35 | 35 |
| | DIVHU | reg1, reg2, reg3 | 4 | 34 | 34 | 34 |
| | DIVU | reg1, reg2, reg3 | 4 | 34 | 34 | 34 |
| | MOV | reg1, reg2 | 2 | 1 | 1 | 1 |
| | MOV | imm5, reg2 | 2 | 1 | 1 | 1 |
| | MOV | imm32, reg1 | 6 | 2 | 2 | 2 |
| | MOVEA | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | MOVHI | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | SASF | cccc, reg2 | 4 | 1 | 1 | 1 |
| | SETF | cccc, reg2 | 4 | 1 | 1 | 1 |
| | SUB | reg1, reg2 | 2 | 1 | 1 | 1 |
| | SUBR | reg1, reg2 | 2 | 1 | 1 | 1 |
| Saturated operation instructions | SATADD | reg1, reg2 | 2 | 1 | 1 | 1 |
| | SATADD | imm5, reg2 | 2 | 1 | 1 | 1 |
| | SATSUB | reg1, reg2 | 2 | 1 | 1 | 1 |
| | SATSUBI | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | SATSUBR | reg1, reg2 | 2 | 1 | 1 | 1 |
| Logical operation instructions | AND | reg1, reg2 | 2 | 1 | 1 | 1 |
| | ANDI | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | BSH | reg2, reg3 | 4 | 1 | 1 | 1 |
| | BSW | reg2, reg3 | 4 | 1 | 1 | 1 |
| | HSW | reg2, reg3 | 4 | 1 | 1 | 1 |
| | NOT | reg1, reg2 | 2 | 1 | 1 | 1 |
| | OR | reg1, reg2 | 2 | 1 | 1 | 1 |
| | ORI | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | SAR | reg1, reg2 | 4 | 1 | 1 | 1 |
| | SAR | imm5, reg2 | 2 | 1 | 1 | 1 |
| | SHL | reg1, reg2 | 4 | 1 | 1 | 1 |
| | SHL | imm5, reg2 | 2 | 1 | 1 | 1 |
| | SHR | reg1, reg2 | 4 | 1 | 1 | 1 |
| | SHR | imm5, reg2 | 2 | 1 | 1 | 1 |
| | SXB | reg1 | 2 | 1 | 1 | 1 |
| | SXH | reg1 | 2 | 1 | 1 | 1 |
| | TST | reg1, reg2 | 2 | 1 | 1 | 1 |
| | XOR | reg1, reg2 | 2 | 1 | 1 | 1 |
| | XORI | imm16, reg1, reg2 | 4 | 1 | 1 | 1 |
| | ZXB | reg1 | 2 | 1 | 1 | 1 |
| | ZXH | reg1 | 2 | 1 | 1 | 1 |

**Table 5-6. List of Number of Instruction Execution Clock Cycles (3/3)**

| Type of Instruction | Mnemonic | Operand | Byte | Number of Execution Clocks | | |
|---|---|---|---|---|---|---|
| | | | | i | r | l |
| Branch instructions | Bcond | disp9 (When condition is satisfied) | 2 | 2[Note 4] | 2[Note 4] | 2[Note 4] |
| | | disp9 (When condition is not satisfied) | 2 | 1 | 1 | 1 |
| ★ | JARL | disp22, reg2 | 4 | 2[Note 5] | 2[Note 5] | 2[Note 5] |
| ★ | JMP | [reg1] | 2 | 3[Note 5] | 3[Note 5] | 3[Note 5] |
| ★ | JR | disp22 | 4 | 2[Note 5] | 2[Note 5] | 2[Note 5] |
| Bit manipulation instructions | CLR1 | bit#3, disp16 [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | CLR1 | reg2, [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | NOT1 | bit#3, disp16 [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | NOT1 | reg2, [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | SET1 | bit#3, disp16 [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | SET1 | reg2, [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | TST1 | bit#3, disp16 [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| | TST1 | reg2, [reg1] | 4 | 3[Note 6] | 3[Note 6] | 3[Note 6] |
| ★ | Special instructions | CALLT | imm6 | 2 | 4[Note 5] | 4[Note 5] | 4[Note 5] |
| ★ | | CTRET | – | 4 | 3[Note 5] | 3[Note 5] | 3[Note 5] |
| | | DI | – | 4 | 1 | 1 | 1 |
| | | DISPOSE | imm5, list12 | 4 | n+1[Note 7] | n+1[Note 7] | n+1[Note 7] |
| | | DISPOSE | imm5, list12, [reg1] | 4 | n+3[Note 7] | n+3[Note 7] | n+3[Note 7] |
| | | EI | – | 4 | 1 | 1 | 1 |
| | | HALT | – | 4 | 1 | 1 | 1 |
| | | LDSR | reg2, regID | 4 | 1 | 1 | 1 |
| | | NOP | – | 2 | 1 | 1 | 1 |
| | | PREPARE | list12, imm5 | 4 | n+1[Note 7] | n+1[Note 7] | n+1[Note 7] |
| | | PREPARE | list12, imm5, sp | 4 | n+2[Note 7] | n+2[Note 7] | n+2[Note 7] |
| | | PREPARE | list12, imm5, imm16 | 6 | n+2[Note 7] | n+2[Note 7] | n+2[Note 7] |
| | | PREPARE | list12, imm5, imm32 | 8 | n+3[Note 7] | n+3[Note 7] | n+3[Note 7] |
| ★ | | RETI | – | 4 | 3[Note 5] | 3[Note 5] | 3[Note 5] |
| | | STSR | regID, reg2 | 4 | 1 | 1 | 1 |
| | | SWITCH | reg1 | 2 | 5 | 5 | 5 |
| ★ | | TRAP | vector | 4 | 3[Note 5] | 3[Note 5] | 3[Note 5] |
| ★ | Debug function instructions[Note 8] | DBRET | – | 4 | 3[Note 5] | 3[Note 5] | 3[Note 5] |
| ★ | | DBTRAP | – | 2 | 3[Note 5] | 3[Note 5] | 3[Note 5] |
| Undefined instruction code | | | | 4 | 3 | 3 | 3 |

**Notes 1.** Depends on the number of wait states (2 if no wait states).

**2.** Depends on the number of wait states (1 if no wait states).

**3.** Shortened by 1 clock if reg2 = reg3 (lower 32 bits of results are not written to register) or reg3 = r0 (higher 32 bits of results are not written to register).

★ **4.** [Type D, E, and F products]
4 when there is an instruction that rewrites the PSW contents immediately before.
[Type A, B, and C products]
3 when there is an instruction that rewrites the PSW contents immediately before.

★ **5.** +1 clock for type D products.
+2 clocks for type E products.

**6.** In case of no wait states (3 + number of read access wait states).

**7.** n is the total number of cycles to load registers in list12. (Depends on the number of wait states; n is the number of registers in list12 if no wait states. The operation when n = 0 is the same as when n = 1).

★ **8.** Type C products do not support instructions for the debug function.

**Remarks 1.** Operand conventions

| Symbol | Meaning |
|---|---|
| reg1 | General-purpose register (used as source register) |
| reg2 | General-purpose register (mainly used as destination register. Some are also used as source registers.) |
| reg3 | General-purpose register (mainly used as remainder of division results or higher 32 bits of multiply results) |
| bit#3 | 3-bit data for bit number specification |
| imm× | ×-bit immediate data |
| disp× | ×-bit displacement data |
| regID | System register number |
| vector | 5-bit data for trap vector (00H to 1FH) specification |
| cccc | 4-bit data condition code specification |
| sp | Stack pointer (r3) |
| ep | Element pointer (r30) |
| list× | List of registers (× is a maximum number of registers) |

**2.** Execution clock conventions

| Symbol | Meaning |
|---|---|
| i | When other instruction is executed immediately after executing an instruction (issue) |
| r | When the same instruction is repeatedly executed immediately after the instruction has been executed (repeat) |
| l | When a subsequent instruction uses the result of execution of the preceding instruction immediately after its execution (latency) |

# CHAPTER 6 INTERRUPTS AND EXCEPTIONS

Interrupts are events that occur independently of program execution and are divided into two types: maskable interrupts and non-maskable interrupts (NMI). In contrast, exceptions are events whose occurrence is dependent on program execution and are divided into three types: software exceptions, exception traps, and debug traps.

When an interrupt or exception occurs, control is transferred to a handler whose address is determined by the source of the interrupt or exception. The source of the interrupt/exception is specified by the exception code that is stored in the exception cause register (ECR). Each handler analyzes the ECR register and performs appropriate interrupt servicing or exception processing. The restored PC and restored PSW are written to the status saving registers (EIPC, EIPSW or FEPC, FEPSW).

To restore execution from interrupt or software exception processing, use the RETI instruction. To restore execution from an exception trap or debug trap, use the DBRET instruction. Read the restored PC and restored PSW from the status saving registers, and transfer control to the restored PC.

**Table 6-1. Interrupt/Exception Codes**

| Interrupt/Exception Source | | | Classification | Exception Code | Handler Address | Restored PC |
|---|---|---|---|---|---|---|
| Name | | Trigger | | | | |
| Non-maskable interrupt (NMI)[Note 1] | | NMI0 input | Interrupt | 0010H | 00000010H | next PC[Note 2] |
| | | NMI1 input | Interrupt | 0020H | 00000020H | next PC[Notes 2, 3] |
| | | NMI2 input[Note 4] | Interrupt | 0030H | 00000030H | next PC[Notes 2, 3] |
| Maskable interrupt | | **Note 5** | Interrupt | **Note 5** | **Note 6** | next PC[Note 2] |
| Software exception | TRAP0n (n = 0 to FH) | TRAP instruction | Exception | 004nH | 00000040H | next PC |
| | TRAP1n (n = 0 to FH) | TRAP instruction | Exception | 005nH | 00000050H | next PC |
| Exception trap (ILGOP) | | Illegal instruction code | Exception | 0060H | 00000060H | next PC[Note 7] |
| Debug trap[Note 8] | | DBTRAP instruction[Note 8] | Exception | 0060H | 00000060H | next PC |

**Notes 1.** The implemented non-maskable interrupt sources differ depending on the product.

**2.** Except when an interrupt is acknowledged during execution of the one of the instructions listed below (if an interrupt is acknowledged during instruction execution, execution is stopped, and then resumed after the completion of interrupt servicing. In this case, the address of the interrupted instruction is the restored PC.).

- Load instructions (SLD.B, SLD.BU, SLD.H, SLD.HU, SLD.W), divide instructions (DIV, DIVH, DIVU, DIVHU)
- PREPARE, DISPOSE instruction (only if an interrupt is generated before the stack pointer is updated)

**3.** The PC cannot be restored by the RETI instruction. Perform a system reset after interrupt servicing.

**4.** Acknowledged even if the NP flag of the PSW is set to 1.

**5.** Differs depending on the type of interrupt.

**6.** The higher 16 bits are 0000H and the lower 16 bits are the same value as the exception code.

**7.** The execution address of the illegal instruction is obtained by “Restored PC – 4”.

★ **8.** Not supported in type C products

**Remark** Restored PC: PC value saved to the EIPC or FEPC when interrupt/exception processing is started
next PC: PC value at which processing is started after interrupt/exception processing

## 6.1 Interrupt Servicing

### 6.1.1 Maskable interrupts

A maskable interrupt can be masked by the interrupt control register of the interrupt controller (INTC).

The INTC issues an interrupt request to the CPU, based on the acknowledged interrupt with the highest priority.

If a maskable interrupt occurs due to interrupt request input (INT input), the CPU performs the following steps, and transfers control to the handler routine.

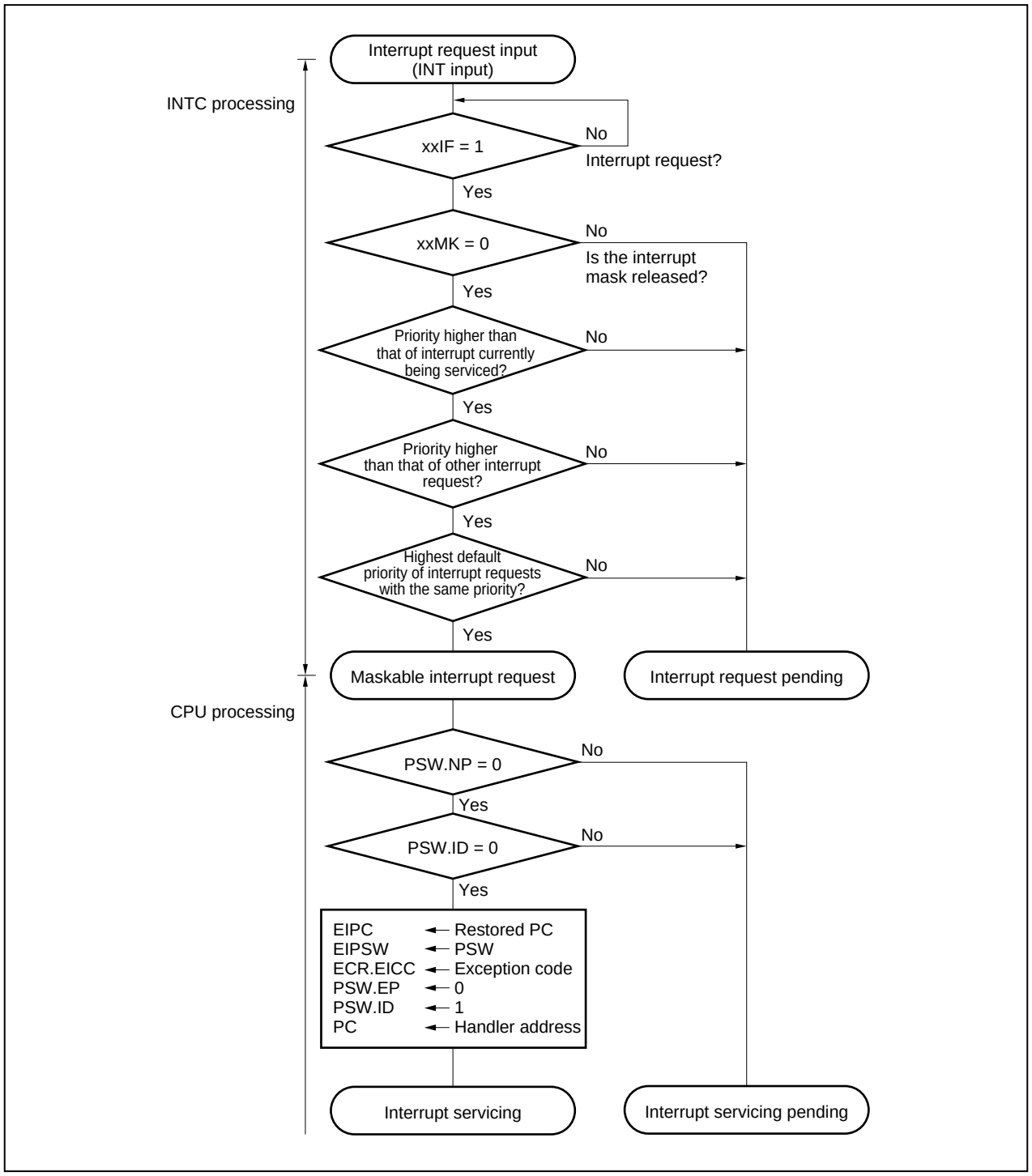
(1) Saves restored PC to EIPC.
(2) Saves current PSW to EIPSW.
(3) Writes exception code to lower halfword of ECR (EICC).
(4) Sets ID flag of PSW to 1 and clears EP flag to 0.
(5) Sets handler address for each interrupt to PC and transfers control.

EIPC and EIPSW are used as the status saving registers. INT inputs are held pending in the interrupt controller (INTC) when one of the following two conditions occur: when the INT input is masked by its interrupt controller, or when an interrupt service routine is currently being executed (when the NP flag of the PSW is 1 or when the ID flag of the PSW is 1). Interrupts are enabled by clearing the mask condition or by setting the NP and ID flags of the PSW to 0 with the LDSR instruction, at which point new maskable interrupt servicing is started by the pending INT input.

The EIPC and EIPSW registers must be saved by program to enable multiple interrupt servicing because there is only one set of EIPC and EIPSW is provided.

The maskable interrupt servicing format is shown below.

**Figure 6-1. Maskable Interrupt Servicing Format**



### 6.1.2 Non-maskable interrupts

A non-maskable interrupt cannot be disabled by an instruction and therefore can always be acknowledged. Non-maskable interrupts are generated by NMI input.

When a non-maskable interrupt is generated, the CPU performs the following steps, and transfers control to the handler routine.

(1) Saves restored PC to FEPC.
(2) Saves current PSW to FEPSW.
(3) Writes exception code (0010H) to higher halfword of ECR (FECC).
(4) Sets NP and ID flags of PSW to 1 and clears EP flag to 0.
(5) Sets handler address for the non-maskable interrupt to PC and transfers control.

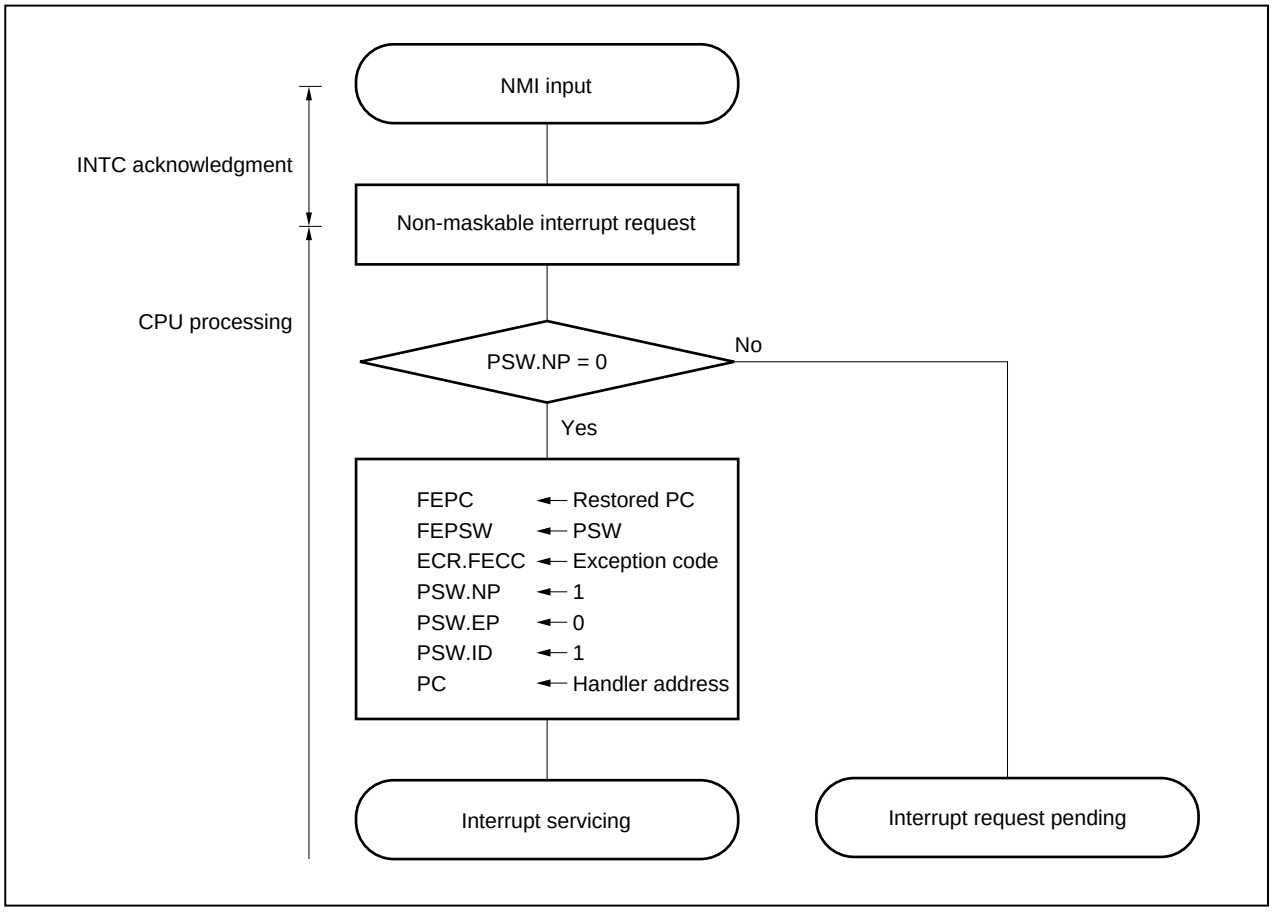
FEPC and FEPSW are used as the status saving registers.

Non-maskable interrupts are held pending in the interrupt controller when another non-maskable interrupt is currently being executed (when the NP flag of the PSW is 1). Non-maskable interrupts are enabled by setting the NP flag of the PSW to 0 with the RETI and LDSR instructions, at which point new non-maskable interrupt servicing is started by the pending non-maskable interrupt request.

★ In the case of type A, B, or C products, NMI2 servicing is executed regardless of the value of the NP flag only when NMI2 is generated during the interrupt servicing of NMI0 and NMI1.

The non-maskable interrupt servicing format is shown below.

**Figure 6-2. Non-Maskable Interrupt Servicing Format**



## 6.2 Exception Processing

### 6.2.1 Software exceptions

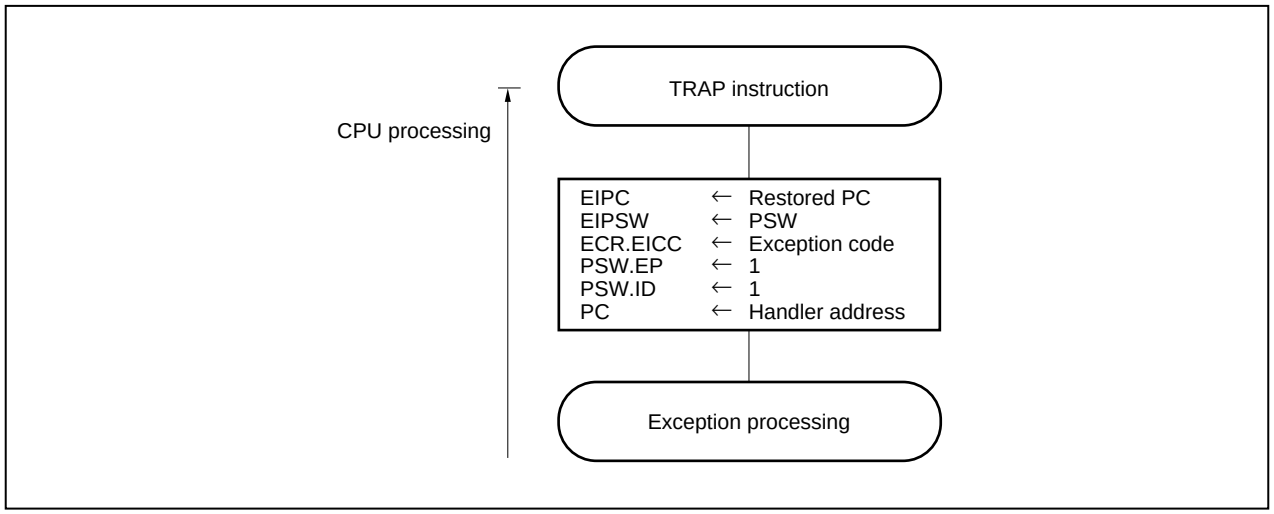
A software exception is generated when the TRAP instruction is executed and is always acknowledged.

If a software exception occurs, the CPU performs the following steps, and transfers control to the handler routine.

(1) Saves restored PC to EIPC.
(2) Saves current PSW to EIPSW.
(3) Writes exception code to lower 16 bits (EICC) of ECR (interrupt source).
(4) Sets EP and ID flags of PSW to 1.
(5) Sets handler address (00000040H or 00000050H) for software exception to PC and transfers control.

The software exception processing format is shown below.

**Figure 6-3. Software Exception Processing Format**

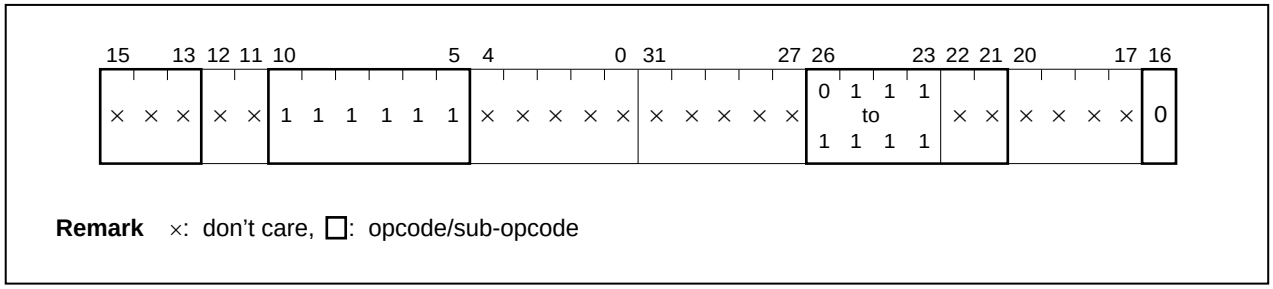


### 6.2.2 Exception trap

An exception trap is an exception requested when an instruction is illegally executed. The illegal opcode trap (ILGOP) is the exception trap in the V850E1 core.

An illegal opcode instruction has an instruction code with an opcode (bits 10 through 5) of 111111B and a sub-opcode (bits 26 through 23) of 0111B through 1111B and a sub-opcode (bit 16) of 0B. When this kind of illegal opcode instruction is executed, an exception trap occurs.

**Figure 6-4. Illegal Instruction Code**



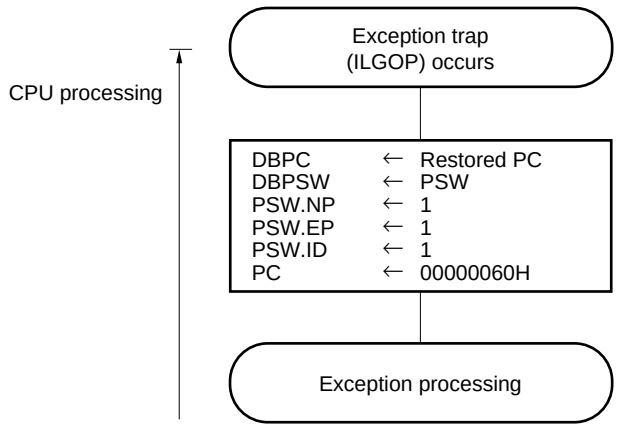

**Remark** ×: don't care, □: opcode/sub-opcode

If an exception trap occurs, the CPU performs the following steps, and transfers control to the handler routine (debug monitor routine).

(1) Saves restored PC to DBPC.
(2) Saves current PSW to DBPSW.
(3) Sets NP, EP, and ID flags of PSW to 1.
★ (4) Sets DM bit of DIR register to 1.
(5) Sets handler address (00000060H) for exception trap to PC and transfers control to debug monitor routine.

The exception trap processing format is shown below.

**Figure 6-5. Exception Trap Processing Format**




**Caution The operation when executing an instruction not defined as an instruction or illegal instruction is not guaranteed.**

**Remark** The execution address of the illegal instruction is obtained by "Restored PC – 4".

### 6.2.3 Debug trap

★ A debug trap is an exception generated when the DBTRAP instruction is executed or when a debug function trap occurs, and is always acknowledged.

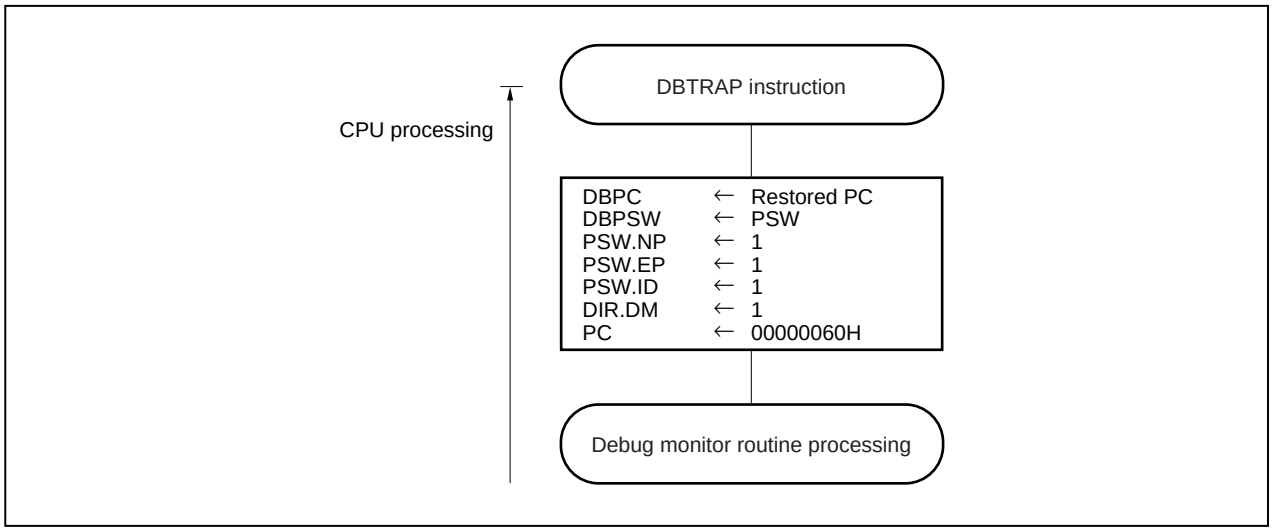
If a debug trap occurs, the CPU performs the following steps.

(1) Saves restored PC to DBPC.
(2) Saves current PSW to DBPSW.
(3) Sets NP, EP, and ID flags of PSW to 1.
(4) Sets DM flag of DIR to 1.
(5) Sets handler address (00000060H) for debug trap to PC and transfers control to debug monitor routine.

★ **Caution Type C products do not support a debug trap.**

The debug trap processing format is shown below.

**Figure 6-6. Debug Trap Processing Format**



## 6.3 Restoring from Interrupt/Exception Processing

### 6.3.1 Restoring from interrupt and software exception

All restoration from interrupt servicing and software exceptions is executed by the RETI instruction.

With the RETI instruction, the CPU performs the following steps, and transfers control to the address of the restored PC.

(1) If the EP flag of the PSW is 0 and the NP flag of the PSW is 1, the restored PC and PSW are read from FEPC and FEPSW. Otherwise, the restored PC and PSW are read from EIPC and EIPSW.
(2) Control is transferred to the address of the restored PC and PSW.

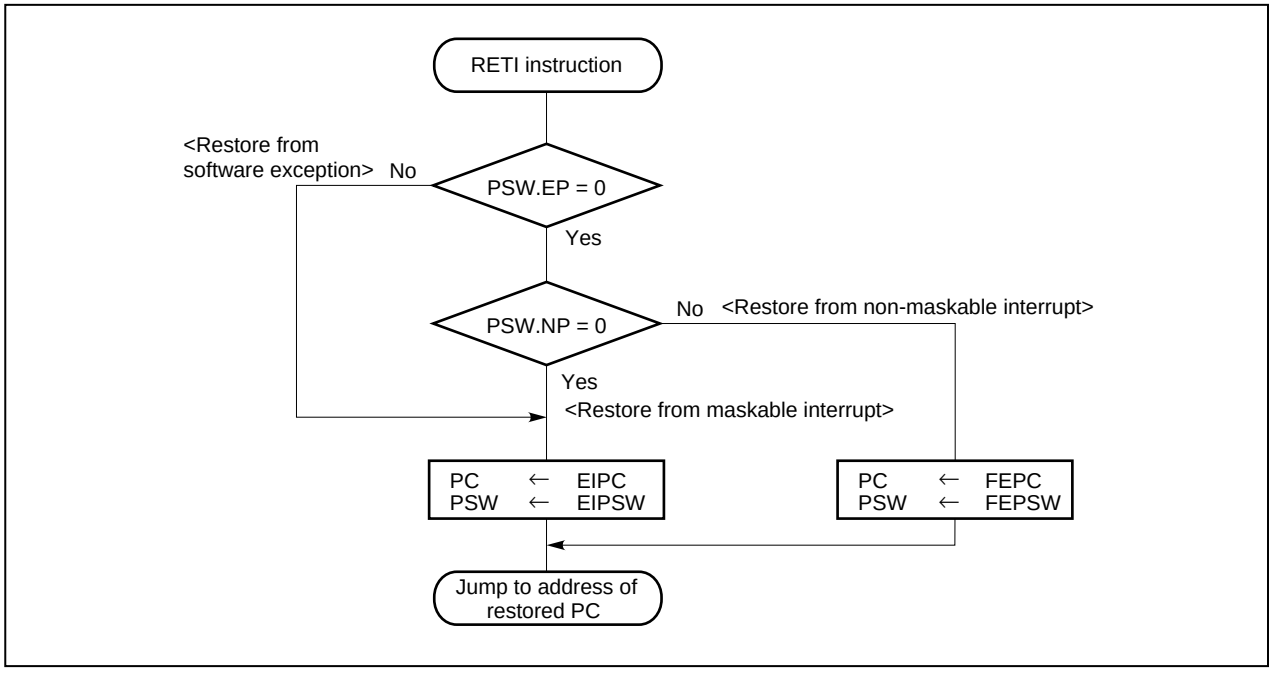
When execution has returned from each interrupt servicing, the NP and EP flags of the PSW must be set to the following values by using the LDSR instruction immediately before the RETI instruction, in order to restore the PC and PSW normally:

- To restore from non-maskable interrupt servicing[Note]: NP flag of PSW = 1, EP flag = 0
- To restore from maskable interrupt servicing: NP flag of PSW = 0, EP flag = 0
- To restore from exception processing: EP flag of PSW = 1

★ **Note** In the case of type A, B, or C products, NMI1 and NMI2 cannot be restored by the RETI instruction. Execute a system reset after interrupt servicing. NMI2 can be acknowledged even if the NP flag of the PSW is set to 1.

The restoration from interrupt/exception processing format is shown below.

**Figure 6-7. Restoration from Interrupt/Software Exception Processing Format**



### 6.3.2 Restoring from exception trap and debug trap

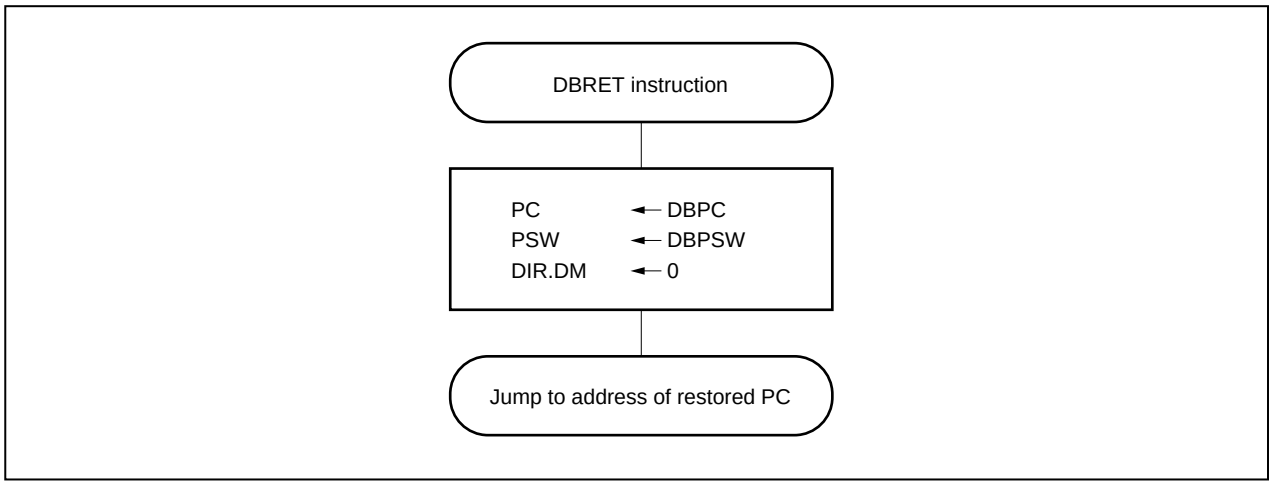
Restoration from an exception trap and debug trap is executed by the DBRET instruction.

With the DBRET instruction, the CPU performs the following steps, and transfers control to the address of the restored PC.

(1) The restored PC and PSW are read from DBPC and DBPSW.
(2) Control is transferred to the address of the restored PC and PSW.
(3) If restoring from exception trap or debug trap, the DM flag of DIR is cleared to 0.

The restoration from exception trap/debug trap processing format is shown below.

**Figure 6-8. Restoration from Exception Trap/Debug Trap Processing Format**



# CHAPTER 7 RESET

## 7.1 Register Status After Reset

When a low-level signal is input to the reset pin, the system is reset, and program registers and system registers are set in the status shown in Table 7-1. When the reset signal goes high, the reset status is cleared, and program execution begins. If necessary, initialize the contents of each register by program control.

**Table 7-1. Register Status After Reset**

| Register | | Status After Reset (Initial Value) |
|---|---|---|
| Program registers | General-purpose register (r0) | 00000000H (Fixed) |
| | General-purpose register (r1 to r31) | Undefined |
| | Program counter (PC) | 00000000H |
| System registers | Interrupt status saving register (EIPC) | 0xxxxxxxH |
| | Interrupt status saving register (EIPSW) | 00000xxxH |
| | NMI status saving register (FEPC) | 0xxxxxxxH |
| | NMI status saving register (FEPSW) | 00000xxxH |
| | Exception cause register (ECR) | 00000000H |
| | Program status word (PSW) | 00000020H |
| | CALLT caller status saving register (CTPC) | 0xxxxxxxH |
| | CALLT caller status saving register (CTPSW) | 00000xxxH |
| | Exception/debug trap status saving register (DBPC) | 0xxxxxxxH |
| | Exception/debug trap status saving register (DBPSW) | 00000xxxH |
| | CALLT base pointer (CTBP) | 0xxxxxxxH |
| | Debug interface register (DIR) | 00000040H |
| | Breakpoint control register 0 (BPC0) | 00xxxxx0H |
| | Breakpoint control register 1 (BPC1) | 00xxxxx0H |
| | Program ID register (ASID) | 000000xxH |
| | Breakpoint address setting register 0 (BPAV0) | 0xxxxxxxH |
| | Breakpoint address setting register 1 (BPAV1) | 0xxxxxxxH |
| | Breakpoint address mask register 0 (BPAM0) | 0xxxxxxxH |
| | Breakpoint address mask register 1 (BPAM1) | 0xxxxxxxH |
| | Breakpoint data setting register 0 (BPDV0) | Undefined |
| | Breakpoint data setting register 1 (BPDV1) | Undefined |
| | Breakpoint data mask register 0 (BPDM0) | Undefined |
| | Breakpoint data mask register 1 (BPDM1) | Undefined |

**Remark** x: Undefined

## 7.2 Starting Up

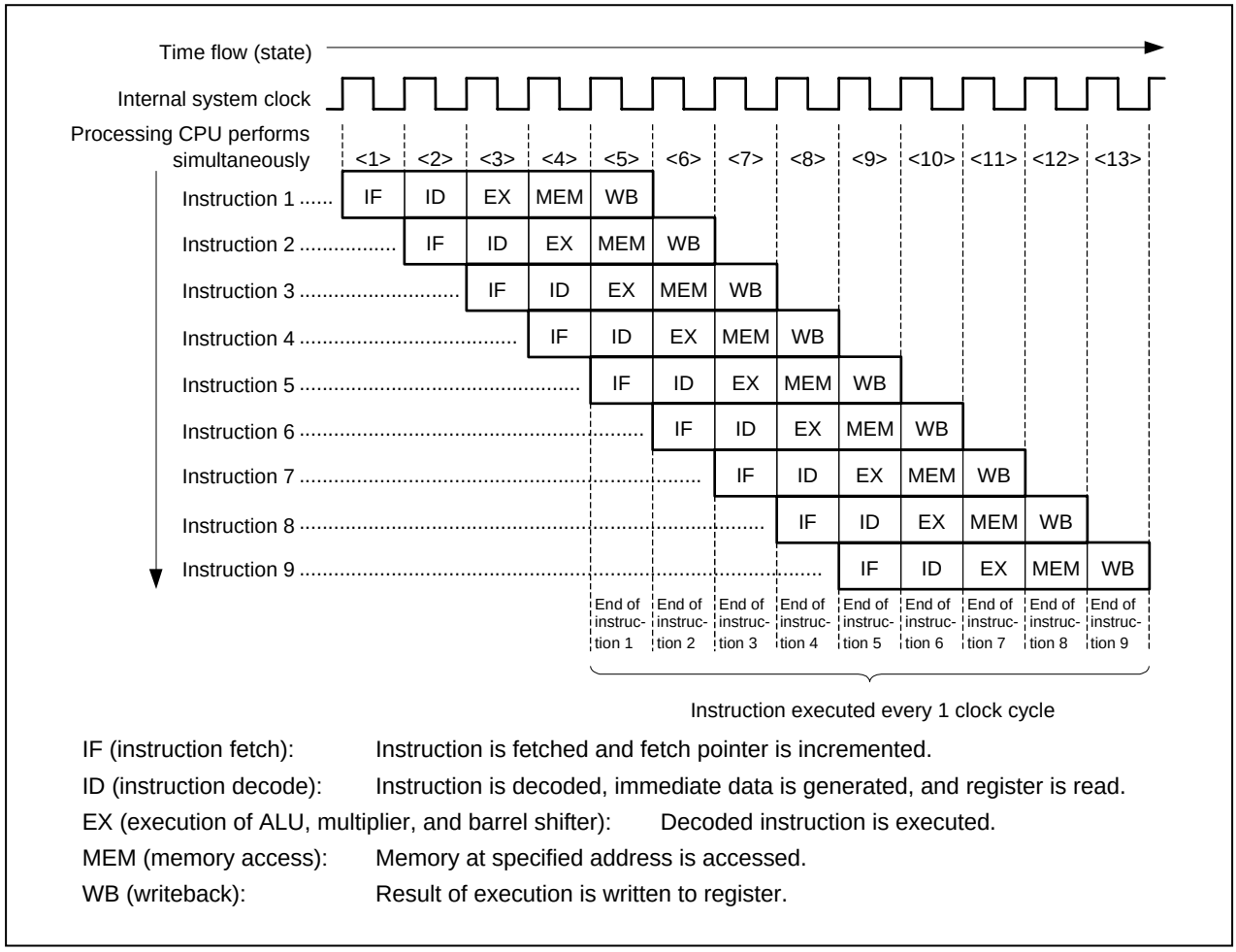
The CPU begins program execution from address 00000000H after it has been reset.

Immediately after reset, no interrupt requests are acknowledged. To enable interrupts by program, clear the ID flag of the PSW to 0.

# CHAPTER 8 PIPELINE

The V850E1 CPU is based on RISC architecture and executes almost all instructions in one clock cycle under control of a 5-stage pipeline. The instruction execution sequence usually consists of five stages from fetch (IF) to writeback (WB). The execution time of each stage differs depending on the type of the instruction and the type of the memory to be accessed. As an example of pipeline operation, Figure 8-1 shows the processing of the CPU when 9 standard instructions are executed in succession.

**Figure 8-1. Example of Executing Nine Standard Instructions**




IF (instruction fetch): Instruction is fetched and fetch pointer is incremented.

ID (instruction decode): Instruction is decoded, immediate data is generated, and register is read.

EX (execution of ALU, multiplier, and barrel shifter): Decoded instruction is executed.

MEM (memory access): Memory at specified address is accessed.

WB (writeback): Result of execution is written to register.

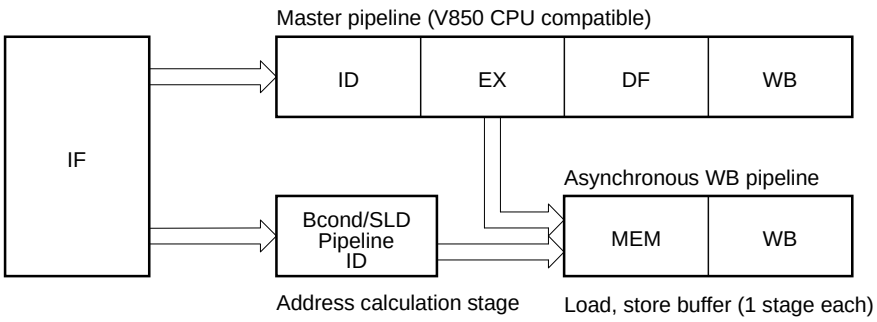
<1> through <13> in the figure above indicate the states of the CPU. In each state, writeback (WB) of instruction n, memory access (MEM) of instruction n+1, execution (EX) of instruction n+2, decoding (ID) of instruction n+3, and fetching (IF) of instruction n+4 are simultaneously performed. It takes five clock cycles to process a standard instruction, from the IF stage to the WB stage. Because five instructions can be processed at the same time, however, a standard instruction can be executed in 1 clock on average.

## 8.1 Features

By optimizing the pipeline, the V850E1 CPU improves the CPI (cycle per instruction) rate over the previous V850 CPU.

The pipeline configuration of the V850E1 CPU is shown in Figure 8-2.

**Figure 8-2. Pipeline Configuration**




**Remark** DF (data fetch): Execution data is transferred to the WB stage.

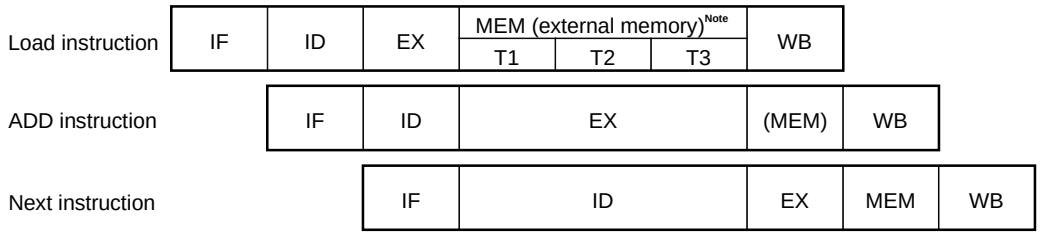
### 8.1.1 Non-blocking load/store

As the pipeline does not stop during external memory access, efficient processing is possible.

For example, Figure 8-3 shows a comparison of pipeline operations between the V850 CPU and the V850E1 CPU when an ADD instruction is executed after the execution of a load instruction for external memory.

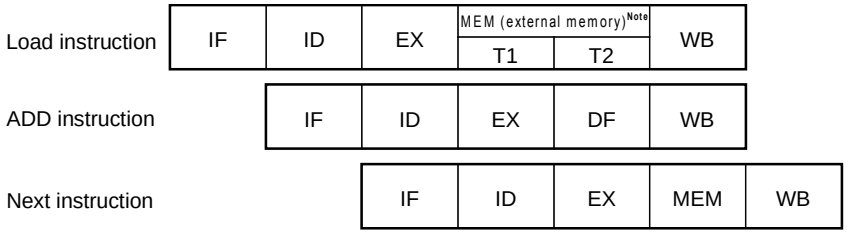
**Figure 8-3. Non-Blocking Load/Store**

**(a) Previous version (V850 CPU): Pipeline is stopped until MEM stage is complete**




**Note** The basic bus cycle for the external memory is 3 clocks.

**(b) V850E1 CPU: Efficient pipeline processing through use of asynchronous WB pipeline**




**Note** The basic bus cycle for the external memory of MEMC is 2 clocks.

**(1) V850 CPU**

The EX stage of the ADD instruction is usually executed in 1 clock. However, a wait time is generated in the EX stage of the ADD instruction during execution of the MEM stage of the previous load instruction. This is because the same stage of the 5 instructions on the pipeline cannot be executed in the same internal clock interval. This also causes a wait time to be generated in the ID stage of the next instruction after the ADD instruction.

**(2) V850E1 CPU**

An asynchronous WB pipeline for the instructions that are necessary for the MEM stage is provided in addition to the master pipeline. The MEM stage of the load instruction is therefore processed by this asynchronous WB pipeline. Because the ADD instruction is processed by the master pipeline, a wait time is not generated, making it possible to execute instructions efficiently as shown in Figure 8-3.

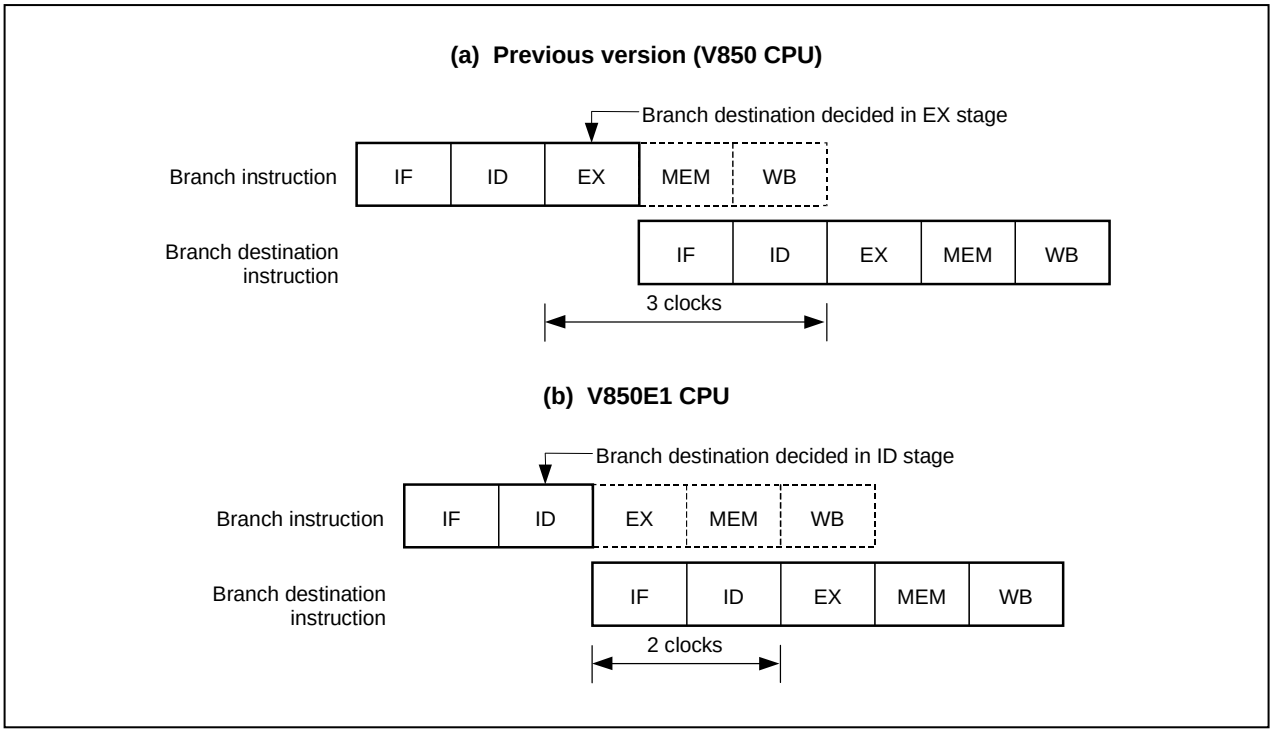
### 8.1.2 2-clock branch

When executing a branch instruction, the branch destination is decided in the ID stage.

In the case of the conventional V850 CPU, the branch destination of when the branch instruction is executed was decided after execution of the EX stage, but in the case of the V850E1 CPU, due to the addition of an address calculation stage for branch/SLD instruction, the branch destination is decided in the ID stage. Therefore, it is possible to fetch the branch destination instruction 1 clock faster than in the conventional V850 CPU.

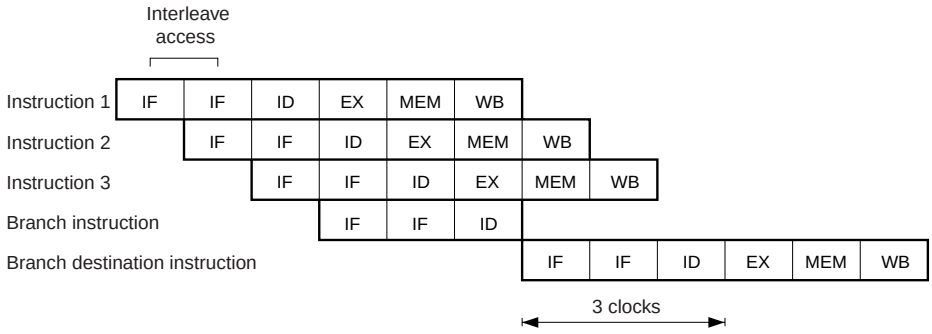
Figure 8-4 shows a comparison between the V850 CPU and the V850E1 CPU for pipeline operations with branch instructions.

**Figure 8-4. Pipeline Operations with Branch Instructions**




★ **Remark** Type D and E products execute interleave access to the internal flash memory or internal mask ROM. Therefore, it takes two clocks (three clocks for type E products) to fetch an instruction immediately after an interrupt has occurred or after a branch destination instruction has been executed. Consequently, it takes three clocks (four clocks for type E products) to execute the ID stage of the branch destination instruction.

**Example**

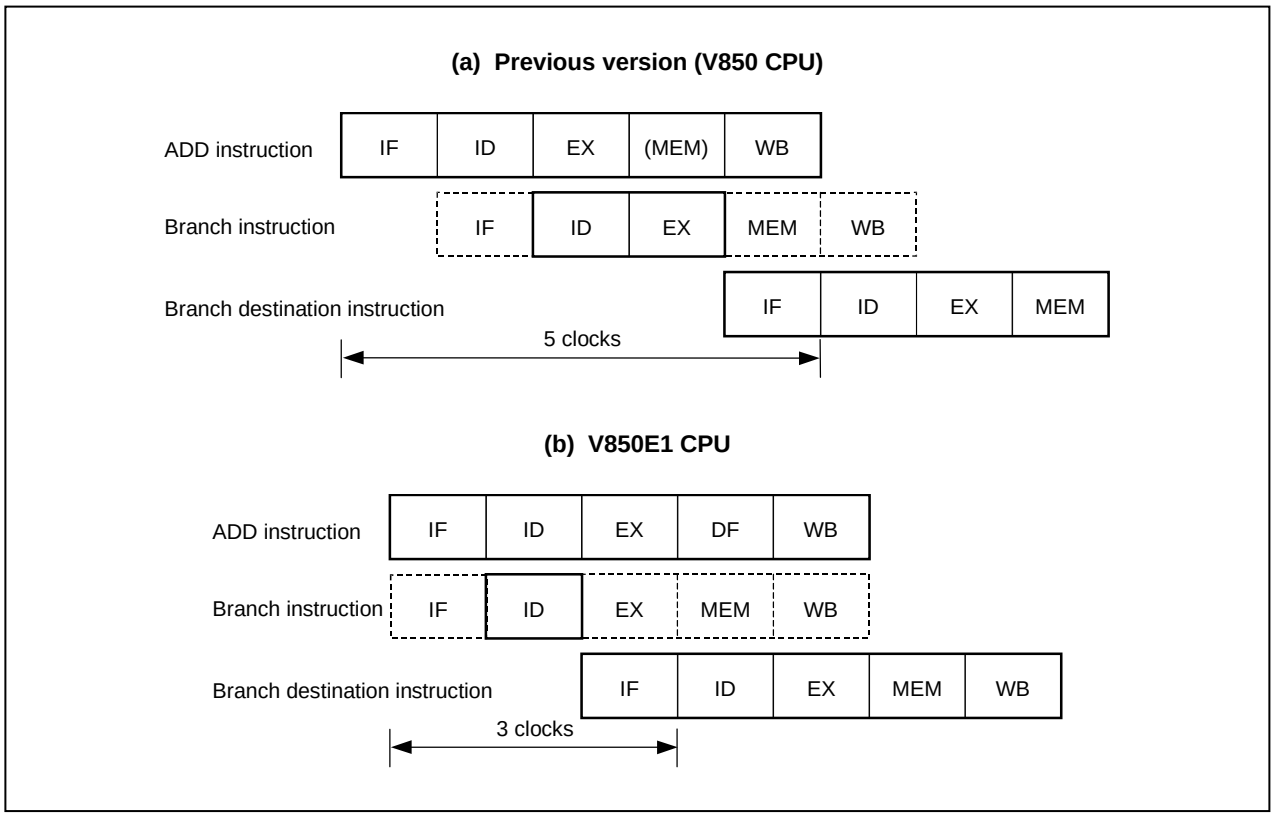


### 8.1.3 Efficient pipeline processing

Because the V850E1 CPU has an ID stage for branch/SLD instructions in addition to the ID stage on the master pipeline, it is possible to perform efficient pipeline processing.

Figure 8-5 shows an example of a pipeline operation where the next branch instruction was fetched in the IF stage of the ADD instruction (instruction fetch from the ROM directly connected to the dedicated bus is performed in 32-bit units. Both ADD instructions and branch instructions in Figure 8-5 use a 16-bit format instruction).

**Figure 8-5. Parallel Execution of Branch Instructions**




**(1) V850 CPU**

Although the instruction codes up to the next branch instruction are fetched in the IF stage of the ADD instruction, the ID stage of the ADD instruction and the ID stage of the branch instruction cannot be executed together within the same clock. Therefore, it takes 5 clocks from the branch instruction fetch to the branch destination instruction fetch.

**(2) V850E1 CPU**

Because V850E1 CPU has an ID stage for branch/SLD instructions in addition to the ID stage on the master pipeline, parallel execution of the ID stage of the ADD instruction and the ID stage of the branch instruction within the same clock is possible. Therefore, it takes only 3 clocks from branch instruction fetch start to branch destination instruction completion.

**Caution Be aware that the SLD and Bcond instructions are sometimes executed at the same time as other 16-bit format instructions. For example, if the SLD and NOP instructions are executed simultaneously, the NOP instruction may keep the delay time from being generated.**

## 8.2 Pipeline Flow During Execution of Instructions

This section explains the pipeline flow during the execution of instructions.

In pipeline processing, the CPU is already processing the next instruction when the memory or I/O write cycle is generated. As a result, I/O manipulations and interrupt request masking will be reflected later than next instruction is issued (ID stage).

★ **(1) Type A, B, and C products**

When a dedicated interrupt controller (INTC) is connected to the NPB (NEC peripheral bus), maskable interrupt acknowledgment is disabled from the next instruction because the CPU detects access to the INTC and performs interrupt request mask processing.

★ **(2) Type D, E, and F products**

When interrupt mask manipulation is performed, maskable interrupt acknowledgment is disabled from the next instruction because the CPU detects access to the internal INTC (ID stage) and performs interrupt request mask processing.

### 8.2.1 Load instructions

**Caution Due to non-blocking control, there is no guarantee that the bus cycle is complete between the MEM stages. However, when accessing the peripheral I/O area, blocking control is effected, making it possible to wait for the end of the bus cycle at the MEM stage.**

★ **For type A, B, and C products, non-blocking control is used for access to the programmable peripheral I/O area.**

#### (1) LD instructions

[Instructions] LD.B, LD.BU, LD.H, LD.HU, LD.W

[Pipeline]

| | <1> | <2> | <3> | <4> | <5> | <6> |
|---|---|---|---|---|---|---|
| LD instruction | IF | ID | EX | MEM | WB | |
| Next instruction | | IF | ID | EX | MEM | WB |

[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. If an instruction using the execution result is placed immediately after the LD instruction, a data wait time occurs.

#### (2) SLD instructions

[Instructions] SLD.B, SLD.BU, SLD.H, SLD.HU, SLD.W

[Pipeline]

| | <1> | <2> | <3> | <4> | <5> | <6> |
|---|---|---|---|---|---|---|
| SLD instruction | IF | ID | MEM | WB | | |
| Next instruction | | IF | ID | EX | MEM | WB |

[Description] The pipeline consists of 4 stages, IF, ID, MEM, and WB. If an instruction using the execution result is placed immediately after the SLD instruction, a data wait time occurs.

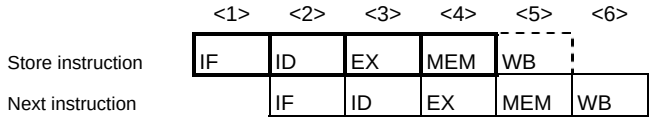
### 8.2.2 Store instructions

**Caution Due to non-blocking control, there is no guarantee that the bus cycle is complete between the MEM stages. However, when accessing the peripheral I/O area, blocking control is effected, making it possible to wait for the end of the bus cycle at the MEM stage.**

★ **For the type A, B, and C products, non-blocking control is used for access to the programmable peripheral I/O area.**

[Instructions] ST.B, ST.H, ST.W, SST.B, SST.H, SST.W

[Pipeline]

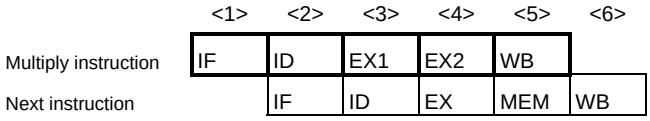



[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the WB stage, because no data is written to registers.

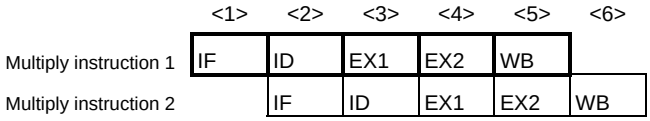
### 8.2.3 Multiply instructions

[Instructions] MUL, MULH, MULHI, MULU

[Pipeline] (a) When next instruction is not multiply instruction




(b) When next instruction is multiply instruction



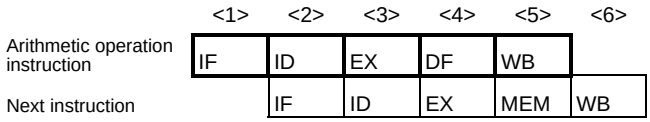

[Description] The pipeline consists of 5 stages, IF, ID, EX1, EX2, and WB. The EX stage takes 2 clocks because it is executed by a multiplier. The EX1 and EX2 stages (different from the normal EX stage) can operate independently. Therefore, the number of clocks for instruction execution is always 1 clock, even if several multiply instructions are executed in a row. However, if an instruction using the execution result is placed immediately after a multiply instruction, a data wait time occurs.

### 8.2.4 Arithmetic operation instructions

#### (1) Instructions other than divide/move word instructions

[Instructions] ADD, ADDI, CMOV, CMP, MOV, MOVEA, MOVHI, SASF, SETF, SUB, SUBR

[Pipeline]

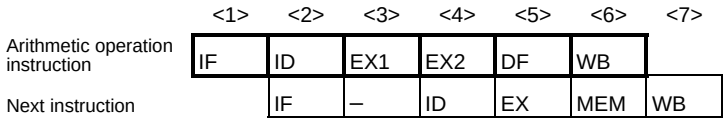



[Description] The pipeline consists of 5 stages, IF, ID, EX, DF, and WB.

#### (2) Move word instruction

[Instructions] MOV imm32

[Pipeline]



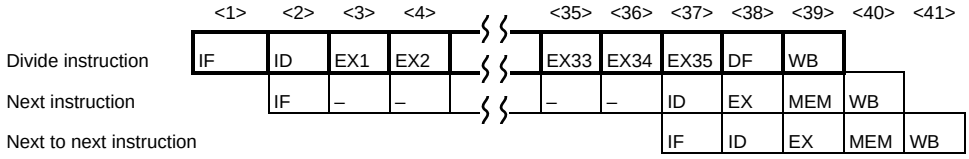

–: Idle inserted for wait

[Description] The pipeline consists of 6 stages, IF, ID, EX1, EX2 (normal EX stage), DF, and WB.

#### (3) Divide instructions

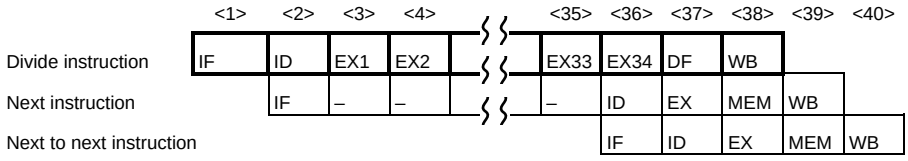
[Instructions] DIV, DIVH, DIVHU, DIVU

[Pipeline] (a) DIV, DIVH instructions




–: Idle inserted for wait

(b) DIVHU, DIVU instructions




–: Idle inserted for wait

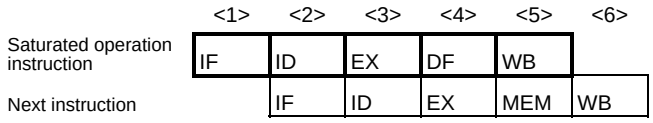
[Description] The pipeline consists of 39 stages, IF, ID, EX1 to EX35 (normal EX stage), DF, and WB for DIV and DIVH instructions. The pipeline consists of 38 stages, IF, ID, EX1 to EX34 (normal EX stage), DF, and WB for DIVHU and DIVU instructions.

[Remark] If an interrupt occurs while a divide instruction is being executed, execution of the instruction is stopped, and the interrupt is serviced, assuming that the return address is the first address of that instruction. After interrupt servicing has been completed, the divide instruction is executed again. In this case, general-purpose registers reg1 and reg2 hold the value before the instruction was executed.

### 8.2.5 Saturated operation instructions

[Instructions] SATADD, SATSUB, SATSUBI, SATSUBR

[Pipeline]

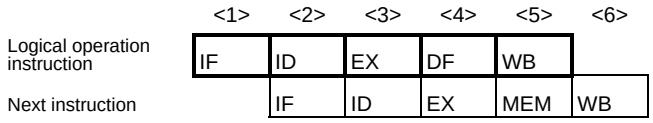



[Description] The pipeline consists of 5 stages, IF, ID, EX, DF, and WB.

### 8.2.6 Logical operation instructions

[Instructions] AND, ANDI, BSH, BSW, HSW, NOT, OR, ORI, SAR, SHL, SHR, SXB, SXH, TST, XOR, XORI, ZXB, ZXH

[Pipeline]



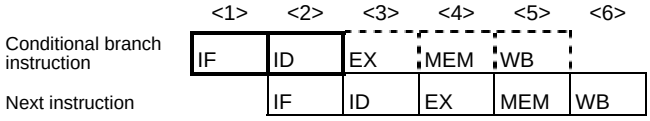

[Description] The pipeline consists of 5 stages, IF, ID, EX, DF, and WB.

### 8.2.7 Branch instructions

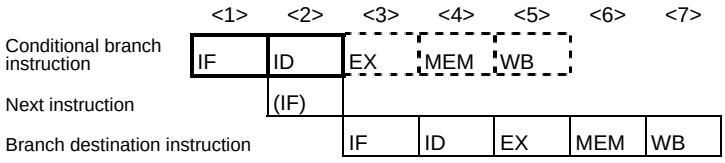
#### (1) Conditional branch instructions (except BR instruction)

[Instructions] Bcond instructions (BC, BE, BGE, BGT, BH, BL, BLE, BLT, BN, BNC, BNE, BNH, BNL, BNV, BNZ, BP, BSA, BV, BZ)

[Pipeline] (a) When the condition is not satisfied




(b) When the condition is satisfied




(IF): Instruction fetch that is not executed

[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the EX, MEM, and WB stages, because the branch destination is decided in the ID stage.

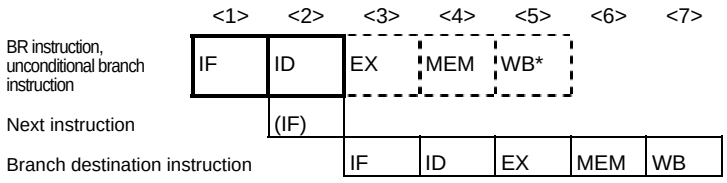
(a) When the condition is not satisfied
The number of execution clocks for the branch instruction is 1.

(b) When the condition is satisfied
The number of execution clocks for the branch instruction is 2. The IF stage of the next instruction of the branch instruction is not executed.
If an instruction overwriting the contents of the PSW occurs immediately before, the number of execution clocks is 3 because of flag hazard occurrence.

**(2) BR instruction, unconditional branch instructions (except JMP instruction)**

[Instructions] BR, JARL, JR

[Pipeline]

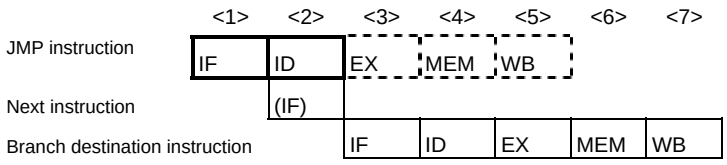



(IF): Instruction fetch that is not executed
WB*: No operation is performed in the case of the JR and BR instructions but in the case of the JARL instruction, data is written to the restored PC.

[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the EX, MEM, and WB stages, because the branch destination is decided in the ID stage. However, in the case of the JARL instruction, data is written to the restored PC in the WB stage. Also, the IF stage of the next instruction of the branch instruction is not executed.

**(3) JMP instruction**

[Pipeline]



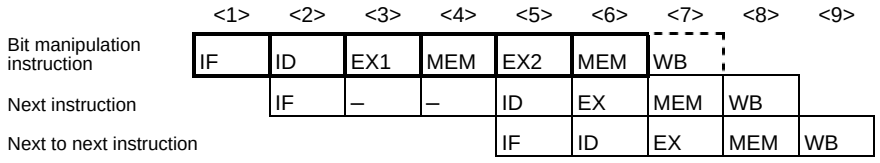

(IF): Instruction fetch that is not executed

[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the EX, MEM, and WB stages, because the branch destination is decided in the ID stage.

### 8.2.8 Bit manipulation instructions

#### (1) CLR1, NOT1, SET1 instructions

[Pipeline]

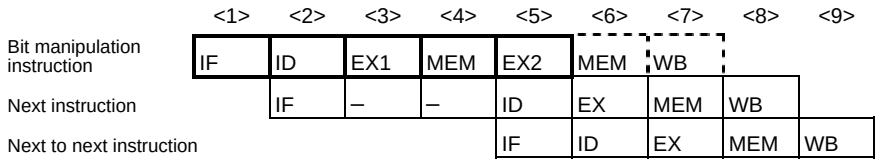



–: Idle inserted for wait

[Description] The pipeline consists of 7 stages, IF, ID, EX1, MEM, EX2 (normal stage), MEM, and WB. However, no operation is performed in the WB stage, because no data is written to registers.
In the case of these instructions, the memory access is read-modify-write, the EX stage requires a total of 2 clocks, and the MEM stage requires a total of 2 cycles.

#### (2) TST1 instruction

[Pipeline]



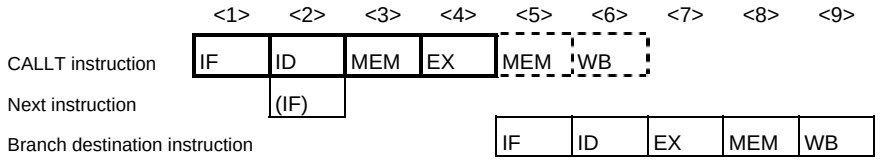

–: Idle inserted for wait

[Description] The pipeline consists of 7 stages, IF, ID, EX1, MEM, EX2 (normal stage), MEM, and WB. However, no operation is performed in the second MEM and WB stages, because there is no second memory access and no data is written to registers.
In all, this instruction requires 2 clocks.

### 8.2.9 Special instructions

#### (1) CALLT instruction

[Pipeline]

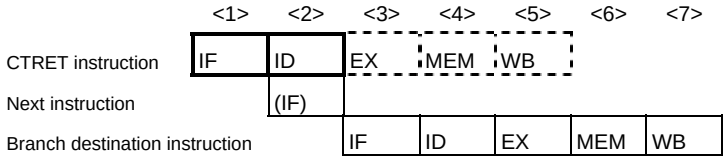



(IF): Instruction fetch that is not executed

[Description] The pipeline consists of 6 stages, IF, ID, MEM, EX, MEM, and WB. However, no operation is performed in the second MEM and WB stages, because there is no memory access and no data is written to registers.

### (2) CTRET instruction

[Pipeline]

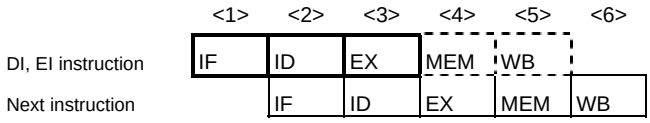



(IF): Instruction fetch that is not executed

[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the EX, MEM, and WB stages, because the branch destination is decided in the ID stage.

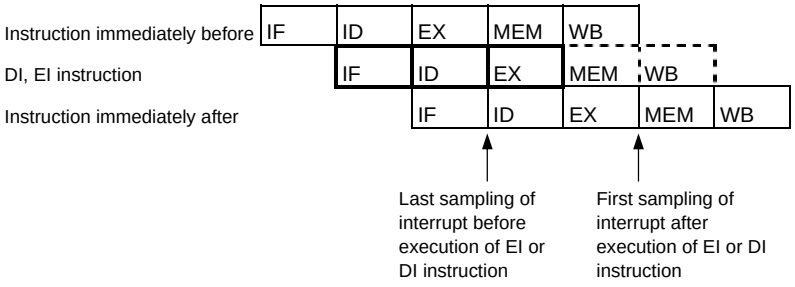
### (3) DI, EI instructions

[Pipeline]



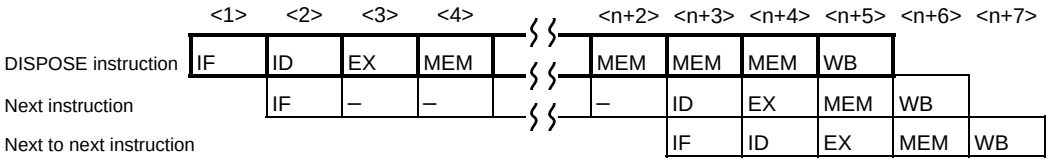

[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the MEM and WB stages, because memory is not accessed and data is not written to registers.

[Remark] Both the DI and EI instructions do not sample an interrupt request. An interrupt is sampled as follows while these instructions are being executed.



### (4) DISPOSE instruction

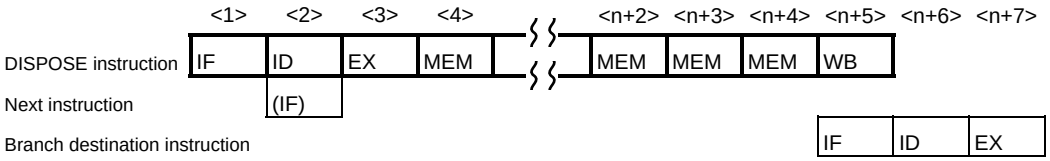
[Pipeline] (a) When branch is not executed




–: Idle inserted for wait

n: Number of registers specified by register list (list12)

(b) When branch is executed




(IF): Instruction fetch that is not executed

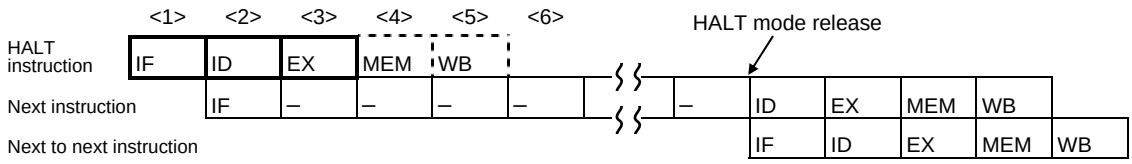
–: Idle inserted for wait

n: Number of registers specified by register list (list12)

[Description] The pipeline consists of n + 5 stages (n: register list number), IF, ID, EX, n + 1 times MEM, and WB. The MEM stage requires n + 1 cycles.

### (5) HALT instruction

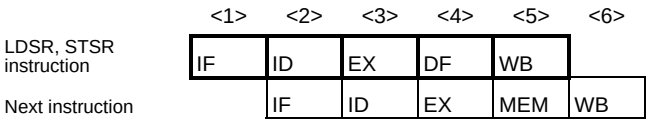
[Pipeline]




[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM and WB. No operation is performed in the MEM and WB stages, because memory is not accessed and no data is written to registers. Also, for the next instruction, the ID stage is delayed until the HALT mode is released.

### (6) LDSR, STSR instructions

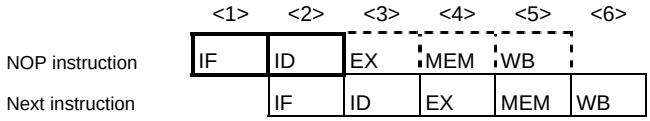
[Pipeline]




[Description] The pipeline consists of 5 stages, IF, ID, EX, DF, and WB. If the STSR instruction using the EIPC and FEPC system registers is placed immediately after the LDSR instruction setting these registers, a data wait time occurs.

### (7) NOP instruction

[Pipeline]

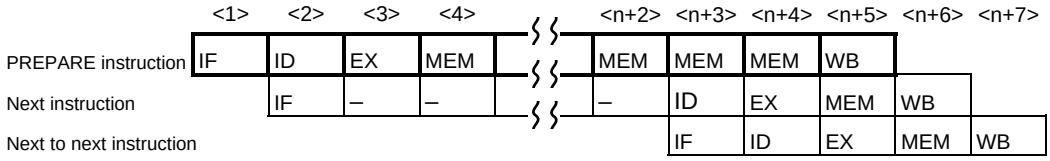



[Description] The pipeline consists of 5 stages, IF, ID, EX, MEM, and WB. However, no operation is performed in the EX, MEM, and WB stages, because no operation and no memory access is executed, and no data is written to registers.

**Caution Be aware that the SLD and Bcond instructions are sometimes executed at the same time as other 16-bit format instructions. For example, if the SLD and NOP instructions are executed simultaneously, the NOP instruction may keep the delay time from being generated.**

### (8) PREPARE instruction

[Pipeline]

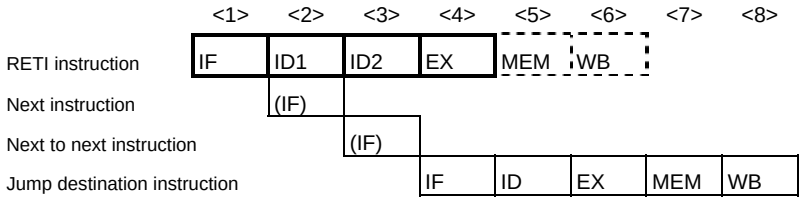



–: Idle inserted for wait
n: Number of registers specified by register list (list12)

[Description] The pipeline consists of n + 5 stages (n: register list number), IF, ID, EX, n + 1 times MEM, and WB. The MEM stage requires n + 1 cycles.

### (9) RETI instruction

[Pipeline]

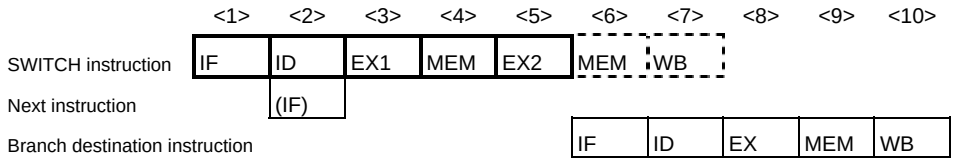



(IF): Instruction fetch that is not executed
ID1: Register selection
ID2: Read EIPC/FEPC

[Description] The pipeline consists of 6 stages, IF, ID1, ID2, EX, MEM, and WB. However, no operation is performed in the MEM and WB stages, because memory is not accessed and no data is written to registers. The ID stage requires 2 clocks. Also, the IF stages of the next instruction and the instruction after that are not executed.

### (10) SWITCH instruction

[Pipeline]

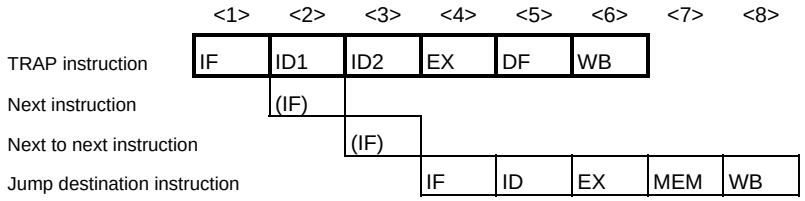



(IF): Instruction fetch that is not executed

[Description] The pipeline consists of 7 stages, IF, ID, EX1 (normal EX stage), MEM, EX2, MEM, and WB. However, no operation is performed in the second MEM and WB stages, because there is no memory access and no data is written to registers.

### (11) TRAP instruction

[Pipeline]



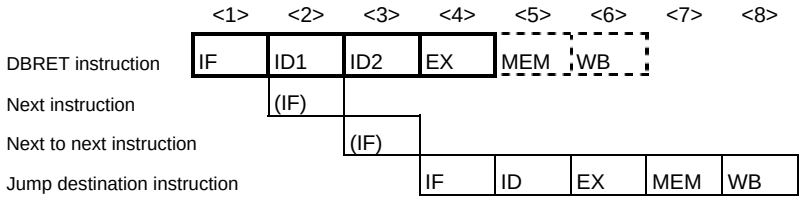

(IF): Instruction fetch that is not executed
ID1: Exception code (004nH, 005nH) detection (n = 0 to FH)
ID2: Address generation

[Description] The pipeline consists of 6 stages, IF, ID1, ID2, EX, DF, and WB. The ID stage requires 2 clocks. Also, the IF stages of the next instruction and the instruction after that are not executed.

### 8.2.10 Debug function instructions

#### (1) DBRET instruction

[Pipeline]

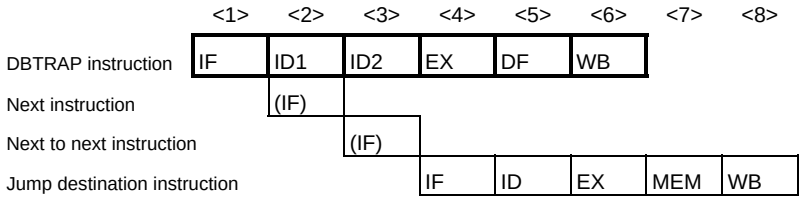



(IF): Instruction fetch that is not executed
ID1: Register selection
ID2: Read DBPC

[Description] The pipeline consists of 6 stages, IF, ID1, ID2, EX, MEM, and WB. However, no operation is performed in the MEM and WB stages, because the memory is not accessed and no data is written to registers. The ID stage requires 2 clocks. Also, the IF stages of the next instruction and the instruction after that are not executed.

#### (2) DBTRAP instruction

[Pipeline]




(IF): Instruction fetch that is not executed
ID1: Exception code (0060H) detection
ID2: Address generation

[Description] The pipeline consists of 6 stages, IF, ID1, ID2, EX, MEM, and WB. The ID stage requires 2 clocks. Also, the IF stages of the next instruction and the instruction after that are not executed.

## 8.3 Pipeline Disorder

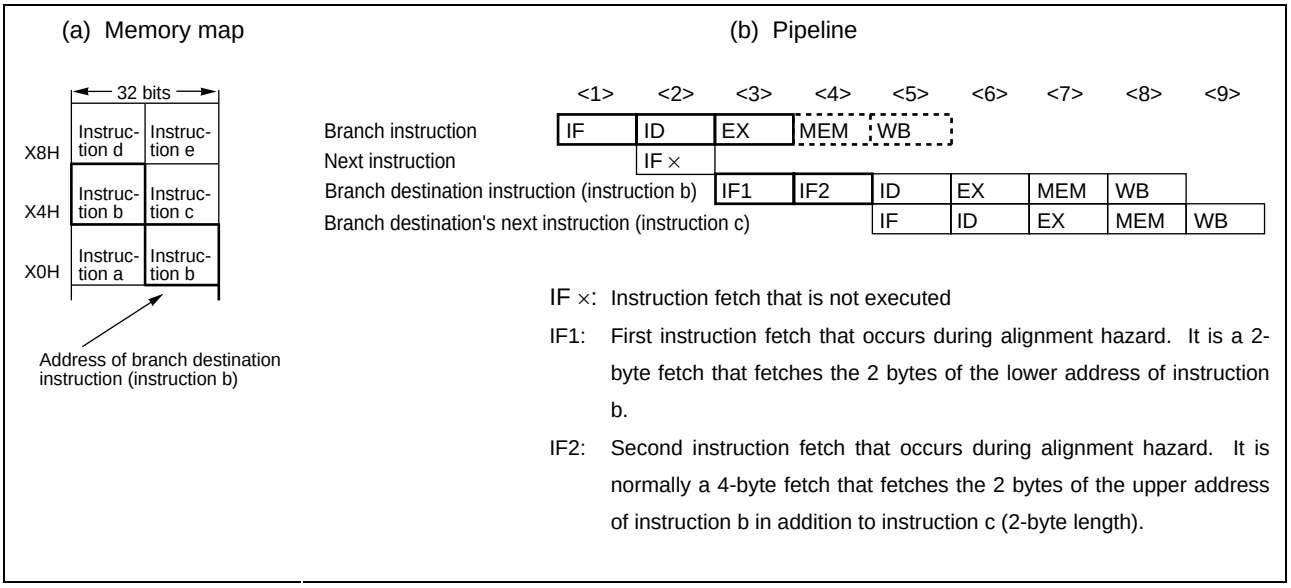
The pipeline consists of 5 stages from IF (Instruction Fetch) to WB (Write Back). Each stage basically requires 1 clock for processing, but the pipeline may become disordered, causing the number of execution clocks to increase. This section describes the main causes of pipeline disorder.

### 8.3.1 Alignment hazard

If the branch destination instruction address is not word aligned (A1 = 1, A0 = 0) and is 4 bytes in length, it is necessary to repeat IF twice in order to align instructions in word units. This is called an alignment hazard.

For example, assume that the instructions a to e are placed from address X0H, and that instruction b consists of 4 bytes, and the other instructions each consist of 2 bytes. In this case, instruction b is placed at X2H (A1 = A0 = 0), and is not word aligned (A1 = 0, A0 = 0). Therefore, when this instruction b becomes the branch destination instruction, an alignment hazard occurs. When an alignment hazard occurs, the number of execution clocks of the branch instruction becomes 4.

**Figure 8-6. Alignment Hazard Example**




IF ×: Instruction fetch that is not executed

IF1: First instruction fetch that occurs during alignment hazard. It is a 2-byte fetch that fetches the 2 bytes of the lower address of instruction b.

IF2: Second instruction fetch that occurs during alignment hazard. It is normally a 4-byte fetch that fetches the 2 bytes of the upper address of instruction b in addition to instruction c (2-byte length).

Alignment hazards can be prevented via the following handling in order to obtain faster instruction execution.

- Use 2-byte branch destination instructions.
- Use 4-byte instructions placed at word boundaries (A1 = 0, A0 = 0) for branch destination instructions.

### 8.3.2 Referencing execution result of load instruction

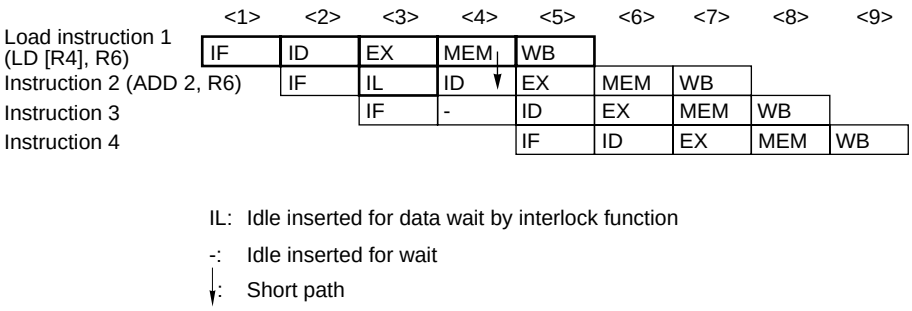
For load instructions (LD, SLD), data read in the MEM stage is saved during the WB stage. Therefore, if the contents of the same register are used by the instruction immediately after the load instruction, it is necessary to delay the use of the register by this later instruction until the load instruction has finished using that register. This is called a hazard.

The V850E1 CPU has an interlock function to automatically handle this hazard by delaying the ID stage of the next instruction.

The V850E1 CPU also has a short path that allows the data read during the MEM stage to be used in the ID stage of the next instruction. This short path allows data to be read by the load instruction during the MEM stage and used in the ID stage of the next instruction at the same timing.

As a result of the above, when using the execution result in the instruction following immediately after, the number of execution clocks of the load instruction is 2.

**Figure 8-7. Example of Execution Result of Load Instruction**



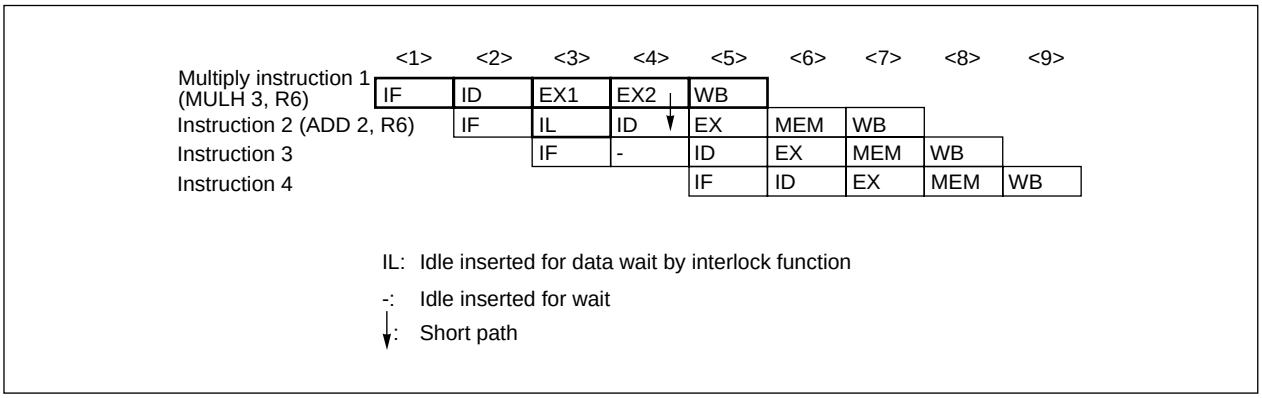

As shown in Figure 8-7, when an instruction placed immediately after a load instruction uses the execution result of the load instruction, a data wait time occurs due to the interlock function, and the execution speed is lowered. This drop in execution speed can be avoided by placing instructions that use the execution result of a load instruction at least 2 instructions after the load instruction.

### 8.3.3 Referencing execution result of multiply instruction

For multiply instructions (MULH, MULHI), the operation result is saved to the register in the WB stage. Therefore, if the contents of the same register are used by the instruction immediately after the multiply instruction, it is necessary to delay the use of the register by this later instruction until the multiply instruction has finished using that register (occurrence of hazard).

The V850E1 CPU's interlock function delays the ID stage of the instruction following immediately after. A short path is also provided that allows the EX2 stage of the multiply instruction and the multiply instruction's operation result to be used in the ID stage of the instruction following immediately after at the same timing.

**Figure 8-8. Example of Execution Result of Multiply Instruction**




As shown in Figure 8-8, when an instruction placed immediately after a multiply instruction uses the execution result of the multiply instruction, a data wait time occurs due to the interlock function, and the execution speed is lowered. This drop in execution speed can be avoided by placing instructions that use the execution result of a multiply instruction at least 2 instructions after the multiply instruction.

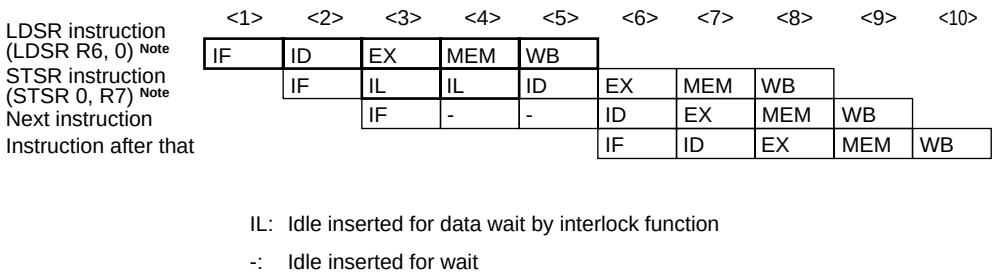
### 8.3.4 Referencing execution result of LDSR instruction for EIPC and FEPC

When using the LDSR instruction to set the data of the EIPC and FEPC system registers, and immediately after referencing the same system registers with the STSR instruction, the use of the system registers for the STSR instruction is delayed until the setting of the system registers with the LDSR instruction is completed (occurrence of hazard).

The V850E1 CPU's interlock function delays the ID stage of the STSR instruction immediately after.

As a result of the above, when using the execution result of the LDSR instruction for EIPC and FEPC for an STSR instruction following immediately after, the number of execution clocks of the LDSR instruction becomes 3.

**Figure 8-9. Example of Referencing Execution Result of LDSR Instruction for EIPC and FEPC**




IL: Idle inserted for data wait by interlock function

-: Idle inserted for wait

**Note** System register 0 used for the LDSR and STSR instructions indicates EIPC.

As shown in Figure 8-9, when an STSR instruction is placed immediately after an LDSR instruction that uses the operand EIPC or FEPC, and that STSR instruction uses the LDSR instruction execution result, the interlock function causes a data wait time to occur, and the execution speed is lowered. This drop in execution speed can be avoided by placing STSR instructions that reference the execution result of the preceding LDSR instruction at least 3 instructions after the LDSR instruction.

### 8.3.5 Cautions when creating programs

When creating programs, pipeline disorder can be avoided and instruction execution speed can be raised by observing the following cautions.

- Place instructions that use the execution result of load instructions (LD, SLD) at least 2 instructions after the load instruction.
- Place instructions that use the execution result of multiply instructions (MULH, MULHI) at least 2 instructions after the multiply instruction.
- If using the STSR instruction to read the setting results written to the EIPC or FEPC registers with the LDSR instruction, place the STSR instruction at least 3 instructions after the LDSR instruction.
- For the first branch destination instruction, use a 2-byte instruction, or a 4-byte instruction placed at a word boundary.

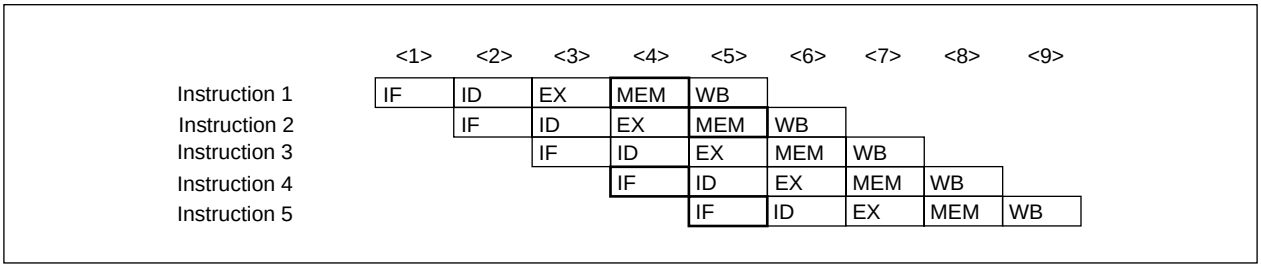
## 8.4 Additional Items Related to Pipeline

### 8.4.1 Harvard architecture

The V850E1 CPU uses Harvard architecture to operate an instruction fetch path from internal ROM and a memory access path to internal RAM independently. This eliminates path arbitration conflicts between the IF and MEM stages and allows orderly pipeline operation.

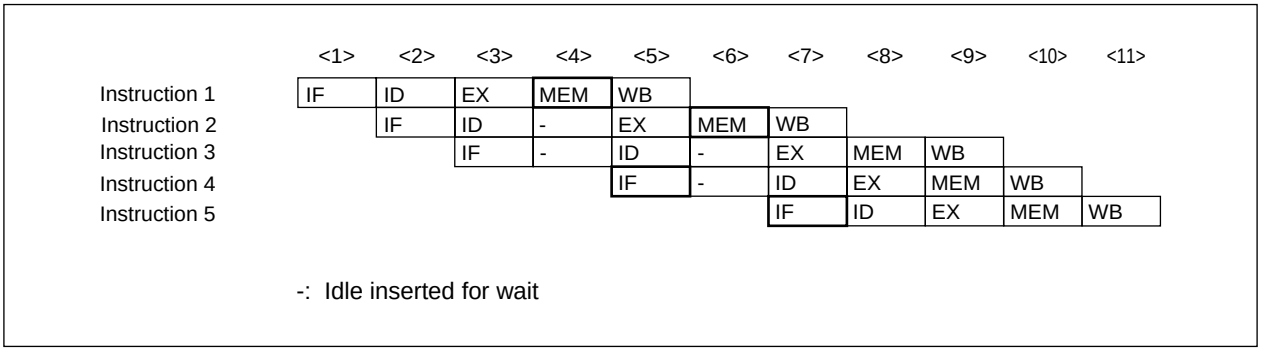
**(1) V850E1 CPU (Harvard architecture)**

The MEM stage of instruction 1 and the IF stage of instruction 4, as well as the MEM stage of instruction 2 and the IF stage of instruction 5 can be executed simultaneously with an orderly pipeline operation.




**(2) Not V850E1 CPU (other than Harvard architecture)**

The MEM stage of instruction 1 and the IF stage of instruction 4, in addition to the MEM stage of instruction 2 and the IF stage of instruction 5 are in conflict, causing path waiting to occur and slower execution time due to disorderly pipeline operation.

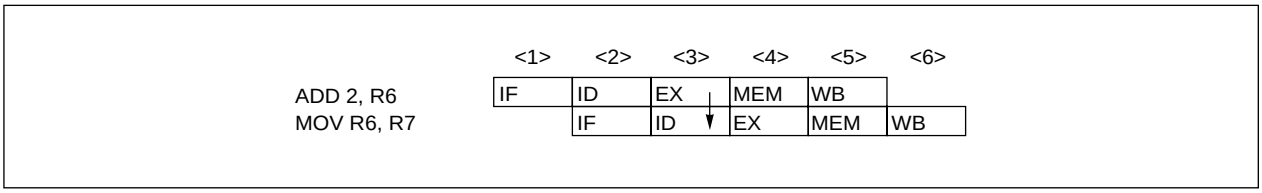


### 8.4.2 Short path

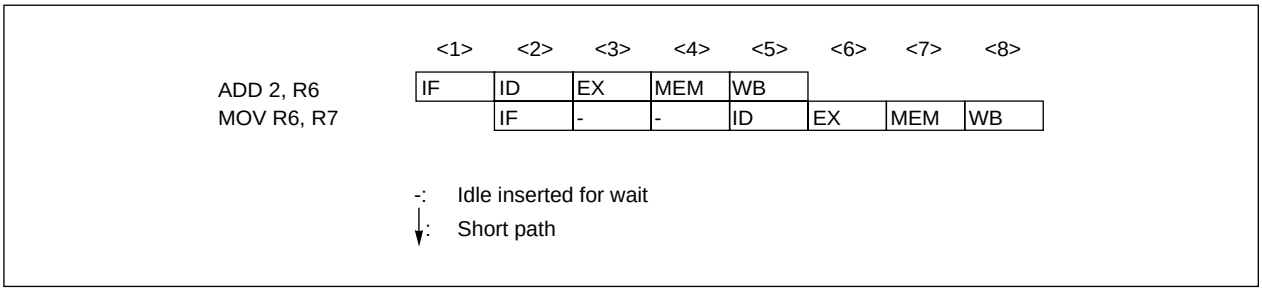
The V850E1 CPU provides on chip a short path that allows the use of the execution result of the preceding instruction by the following instruction before writeback (WB) is completed for the previous instruction.

**Example 1.** Execution result of arithmetic operation instruction and logical operation used by instruction following immediately after

- V850E1 CPU (on-chip short path)
  The execution result of the preceding instruction can be used for the ID stage of the instruction following immediately after as soon as the result is out (EX stage), without having to wait for writeback to be completed.

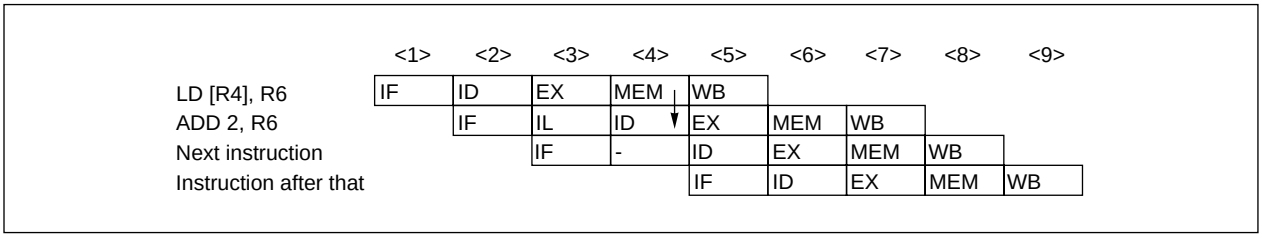



- Not V850E1 CPU (No short path)
  The ID stage of the instruction following immediately after is delayed until writeback of the previous instruction is completed.

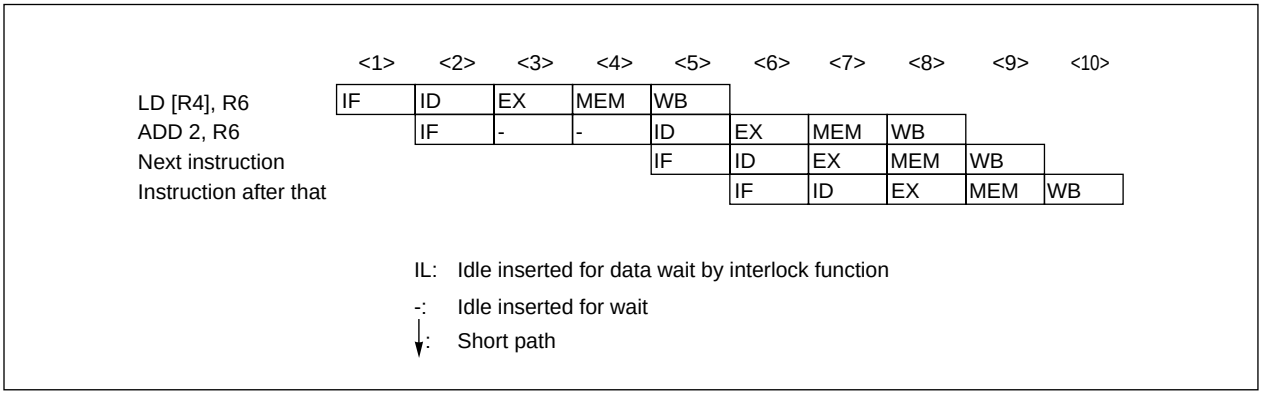


**Example 2.** Data read from memory by the load instruction used by instruction following immediately after

- V850E1 CPU (on-chip short path)
  The execution result of the preceding instruction can be used for the ID stage of the instruction following immediately after as soon as the result is out (MEM stage), without having to wait for writeback to be completed.




- Not V850E1 CPU (No short path)
  The ID stage of the instruction following immediately after is delayed until writeback of the previous instruction is completed.



★

# CHAPTER 9 SHIFTING TO DEBUG MODE

The V850E1 CPU sets the handler address (00000060H) to the program counter (PC) when a debug trap, exception trap, or debug break occurs, and then shifts to the debug mode.

Moreover, setting single-step operation makes it possible to shift to debug mode each time an instruction executed.

**Caution When the V850E1 CPU shifts to the debug mode, the data cache is held, and the data and tags are not updated. If the external memory of the cacheable area is accessed in the debug mode, the coherency is corrupted because the data cache is valid only while the external memory is being accessed. Therefore, to manipulate cacheable area data in a debug monitor routine, clear the data cache (for write through) or flush and clear (for writeback) before restoring to the user mode.**

## 9.1 How to Shift to Debug Mode

**(1) Debug trap**

Execution of the DBTRAP instruction generates a debug trap and shifts the V850E1 CPU to the debug mode (see **6.2.3 Debug trap**).

**(2) Exception trap**

Invalid execution of instructions generates an exception trap and shifts the V850E1 CPU to the debug mode (see **6.2.2 Exception trap**).

**(3) Debug break**

The following three types of debug breaks are available.

- Break due to setting breakpoints (2 channels)
- Break due to misalign access exception occurrence
- Break due to alignment error exception occurrence

The following system registers are used to set debug breaks.

- Debug interface register (DIR)
- Breakpoint control registers 0, 1 (BPC0, BPC1)
- Breakpoint address setting registers 0, 1 (BPAV0, BPAV1)
- Breakpoint address mask registers 0, 1 (BPAM0, BPAM1)
- Breakpoint data setting registers 0, 1 (BPDV0, BPDV1)
- Breakpoint data mask registers 0, 1 (BPDM0, BPDM1)

**Remark** Registers, except for the ASID register, can be read or written only in debug mode (the DIR register can be read in user mode). Therefore, perform the initial settings of each register and reading/writing at an arbitrary timing after shifting to debug mode by a debug trap (execution of DBTRAP instruction).

**(a) Break due to setting breakpoints (2 channels)**

The V850E1 CPU shifts to the debug mode based on the breakpoint settings (2 channels) validated when the following break conditions are satisfied. The BPCn register is used to set each condition (n = 0, 1).

**Caution While the IE bit of the BPCn register is set to 1, the system does not shift to the debug mode if the BP ASID bit value and the program ID set to the ASID register do not match; even if the break conditions match.**

**Table 9-1. Break Conditions**

| Type | Break Condition | | Break Timing | BPxxn Register Setting[Note 2] | | | | Setting of MD, FE, RE, WE Bits of BPCn Register | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Address[Note 1] | Data | | BP AVn | BP AMn | BP DVn | BP DMn | MD | FE | RE, WE |
| Execution trap | Arbitrary execution address | Specific instruction code | Immediately before execution | <1> | <1> | √ | <0> | 0 | 1 | 0[Note 5] |
| | | Specific instruction code range | | <1> | <1> | √ | √ | | | |
| | Specific execution address | Arbitrary instruction code | | √ | <0> | <1> | <1> | Any | | |
| | | Specific instruction code | | √ | <0> | √ | <0> | 0 | | |
| | | Specific instruction code range | | √ | <0> | √ | √ | | | |
| | Specific execution address range | Arbitrary instruction code | | √ | √ | <1> | <1> | Any | | |
| | | Specific instruction code | | √ | √ | √ | <0> | 0 | | |
| | | Specific instruction code range | | √ | √ | √ | √ | | | |
| Access trap | Arbitrary access address | Specific data | After execution[Note 3] | <1> | <1> | √ | <0> | 0 | 0 | 0/1[Note 6] |
| | | Specific data range | Immediately after execution | <1> | <1> | √ | v | | | |
| | Specific access address | Arbitrary data | After execution[Note 3] | √ | <0> | <1> | <1> | Any[Note 4] | | |
| | | Specific data | | √ | <0> | √ | <0> | 0 | | |
| | | Specific data range | | √ | <0> | √ | √ | | | |
| | Specific access address range | Arbitrary data | Immediately after execution | √ | √ | <1> | <1> | Any[Note 4] | | |
| | | Specific data | After execution[Note 3] | √ | √ | √ | <0> | 0 | | |
| | | Specific data range | | √ | √ | √ | √ | | | |

**Notes 1.** The execution address indicates the address of an instruction fetch, and the access address indicates the address at which an access occurs in accordance with instruction execution.

**2.** Set as follows.

√: Set the break conditions.

<0>: Clear all bits to 0.

<1>: It is not necessary to set the conditions, but set all bits to 1 because the initial value is undefined (bits 31 to 28 of the BPAVn and BPAMn registers are fixed to 0, and cannot be set to 1).

For an execution trap or for an access trap that targets a 64 MB data area, bits 27 and 26 of the BPAVn and BPAMn registers are ignored. However, set them to 1 because the initial value is undefined.

**3.** Data write: Immediately after execution
Data read: After several instructions are executed (slip)

**4.** When the MD bit is set to 1, match judgment by the data comparator is ignored. Therefore, the break latency is accelerated by 1 clock (a break occurs at the MEM stage when MD = 0, and at the EX stage when MD = 1).

**5.** Always set to 0 (operation is not guaranteed when set to 1).

**6.** Set in accordance with the access type (read only, write only, or read/write)

**Cautions 1. The match timing of break conditions differs between an execution trap and an access trap (at the ID stage for an execution trap, and at the MEM stage for an access trap). Therefore, even if the sequential break mode is set, the V850E1 CPU may not operate normally when an execution trap occurs after an access trap.**

**2. In the range break mode, set either the execution trap or access trap to channels 0 and 1.**

**Remarks 1.** n = 0, 1

**2.** When multiple break conditions are set, the debug mode is entered if at least one of them is satisfied.

**3.** Channels 0 and 1 can be linked to perform the following two operations (however, simultaneous operations are not possible).

(i) Break by sequential execution (range break mode)
This break is set by setting the SQ bit of the debug interface register (DIR) to 1. The debug mode is entered only when the break conditions of channels 0 and 1 match in that order.

(ii) Break by simultaneous execution (range break mode)
This break is set by setting the RE bit of the debug interface register (DIR) to 1. The debug mode is entered only when the break conditions of channels 0 and 1 match at the same time.

**(b) Break due to misalign access exception occurrence**

This break is set by setting the MA bit of the debug interface register (DIR) to 1. The debug mode is entered when a misalign access occurs during execution of the load and store instructions (independent of the enable/disable setting of misaligned access to the CPU).

**(c) Break due to alignment error exception occurrence**

This break is set by setting the AE bit of the debug interface register (DIR) to 1.
The V850E1 CPU shifts to the debug mode when an alignment error occurs.
An alignment error occurs in the following case.

- When the stack pointer (SP) is forcibly aligned to other than a word boundary during PREPARE or DISPOSE instruction execution

**Remark** Misaligned access to the CPU is enabled/disabled via hardware settings (pin input) (in the V850E1 core, set according to the level input to the IFIMAEN pin).

In debug breaks except for access traps, the address of the instruction that caused the break is saved to DBPC (because debug mode is entered before instruction execution is complete). Therefore, the instruction that caused a break is executed after shifting from debug mode to user mode, but an additional debug break does not occur (ignored).

**(4) Single-step operation**

The single-step operation is set by setting the SS flag of the PSW to 1, and the debug mode is entered when each instruction is executed. The single-step operation is set/cleared using the following procedure.

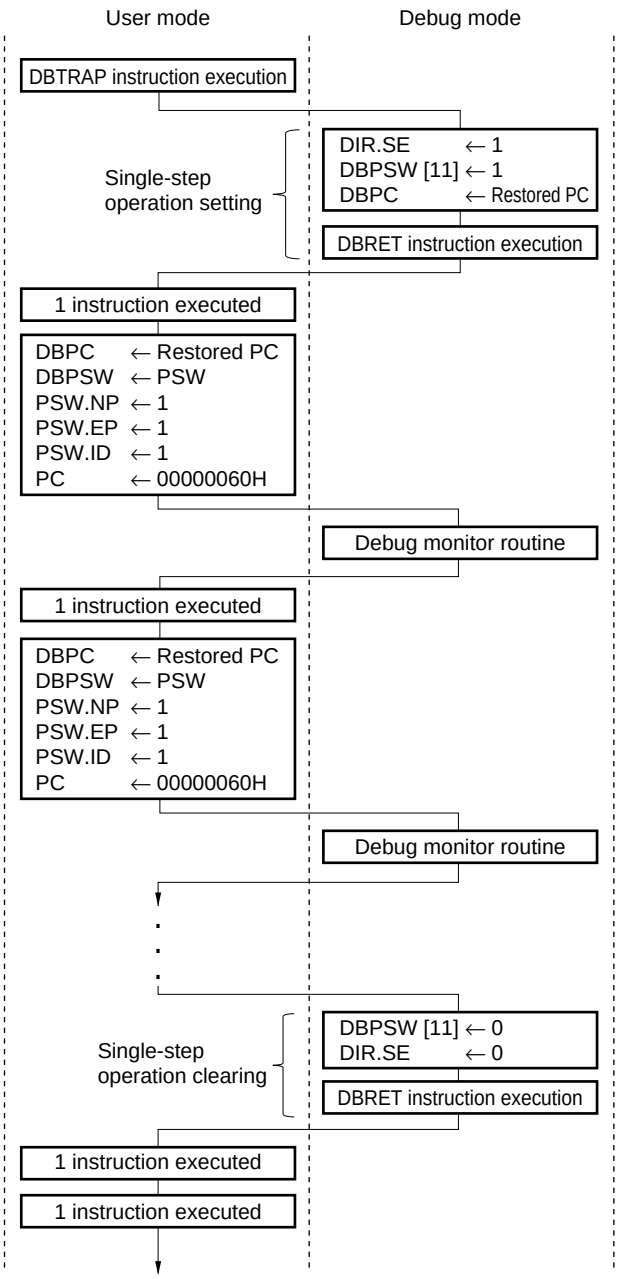
**(a) Single-step operation setting procedure**

<1> Shift to debug mode via a debug trap (DBTRAP instruction execution).
<2> Set the SE bit of the DIR register to 1 to control the SS flag of the PSW.
<3> Set bit 11 of the DBPSW register to 1 to set the SS flag of the PSW to 1 when shifting to the user mode.
<4> Transfer the restored PC value to the DBPC register.
<5> Shift to the user mode via the DBRET instruction (the SS flag of the PSW is set to 1 while shifting and the single-step operation is set).

**(b) Single-step operation clearing procedure**

<1> When operating in the debug mode, clear bit 11 of the DBPSW register to 0 (this manipulation clears the SS flag of the PSW to 0 when shifting to the user mode).
<2> Clear the SE bit of the DIR register to 0 (however, if this manipulation is omitted, the SS flag of the PSW can be set to 1).
<3> Shift to the user mode via the DBRET instruction (the SS flag of the PSW is cleared to 0 while shifting and the single-step operation is cleared).

**Figure 9-1. Single-Step Operation Execution Flow**

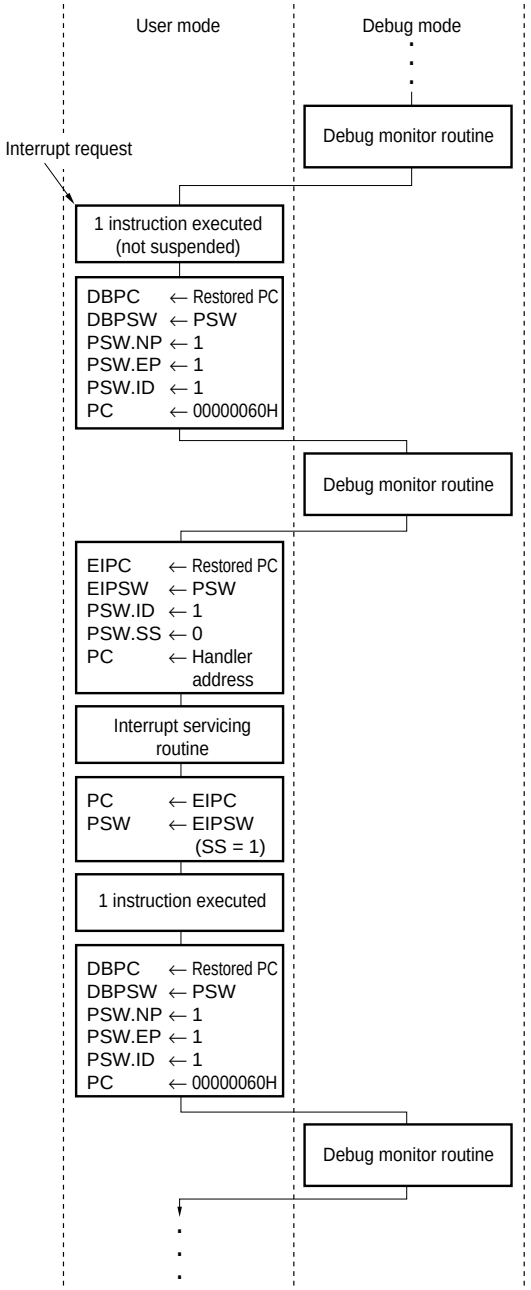



**Remark** The SS flag of the PSW is automatically cleared to 0 when an interrupt request is generated in user mode in a single-step operation. Therefore, the single-step operation is not performed in the interrupt servicing routine (the SS flag is set to 1 again due to the restore processing from the interrupt servicing routine (EIPSW → PSW)).

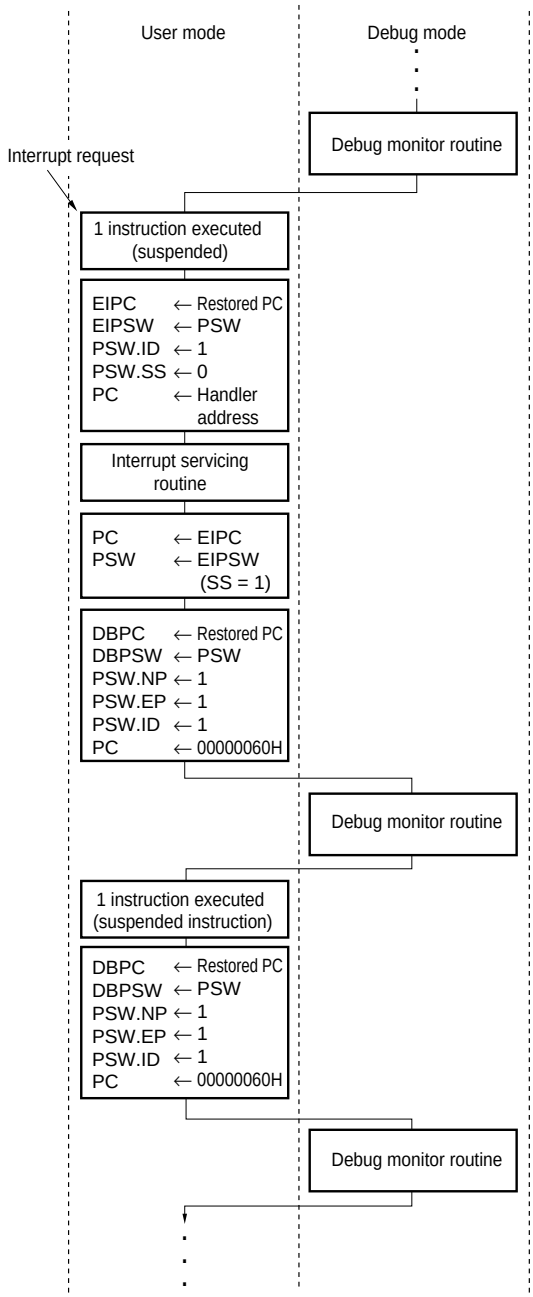
The processing flow may vary depending on the instruction that is executed when an interrupt occurs (see **Figure 9-2**).

**Figure 9-2. Processing Flow When Interrupt Request Is Generated During Single-Step Operation**

**(a) Instruction that does not suspend the execution by interrupt request**




**(b) Instruction that suspends the execution by interrupt request**




**Remark** For the instructions that suspend the execution by interrupt request (see **Table 6-1 Interrupt/Exception Codes**), the interrupt servicing may be performed without waiting for the completion of that instruction execution, and the debug mode may be entered executing no instruction after restoring from the interrupt servicing routine.

## 9.2 Cautions

The set value of the BPDVn register differs in accordance with the address to be accessed in misaligned access or access by a bit manipulation instruction (n = 0, 1).

In misaligned access, memory access cycles are generated divided into several cycles. In write access, only the address, data, and access type (halfword/byte) of the divided first cycle are compared as break conditions. Also in access by a bit manipulation instruction, the set value of the BPDVn register differs in accordance with the address to be accessed.

The following shows an example of setting break conditions for each access address according to the access size.

**Table 9-2. Break Condition Setting Example**

| Access Size (Sample Data) | Access Address[Note 1] | Bus Cycle | TY Bit of BPCn Register | | BPAVn Register[Note 1] | BPDVn Register[Note 2] | |
|---|---|---|---|---|---|---|---|
| | | | Write | Read | | Write | Read |
| Word (44332211H) | 0H | W | 1, 1 (W) | 1, 1 (W) | 0H | 44332211H | 44332211H |
| | 1H | B→HW→B | 0, 1 (B) | | 1H | xxxx11xxH | |
| | 2H | HW→HW | 1, 0 (HW) | | 2H | 2211xxxxH | |
| | 3H | B→HW→B | 0, 1 (B) | | 3H | 11xxxxxxH | |
| Halfword (2211H) | 0H | HW | 1, 0 (HW) | 1, 0 (HW) | 0H | xxxx2211H | xxxx2211H |
| | 1H | B→B | 0, 1 (B) | | 1H | xxxx11xxH | |
| | 2H | HW | 1, 0 (HW) | | 2H | 2211xxxxH | |
| | | | | | | xxxx2211H[Note 3] | |
| | 3H | B→B | 0, 1 (B) | | 3H | 11xxxxxxH | |
| Byte (11H) | 0H | B | 0, 1 (B) | | 0H | xxxxxx11H | xxxxxx11H |
| | 1H | | | | 1H | xxxx11xxH | |
| | | | | | | xxxxxx11H[Note 4] | |
| | 2H | | | | 2H | xx11xxxxH | |
| | | | | | | xxxxxx11H[Note 4] | |
| | 3H | | | | 3H | 11xxxxxxH | |
| | | | | | | xxxxxx11H[Note 4] | |
| Byte (11H) | 0H | B | 0, 1 (B) | | 0H | xxxxxx11H | |
| | 1H | | | | 1H | xxxx11xxH | |
| | 2H | | | | 2H | xx11xxxxH | |
| | 3H | | | | 3H | 11xxxxxxH | |

**Notes 1.** Indicates the value of the lower two bits.

**2.** "x" indicates being masked by the BPDMn register.

**3.** Valid only during halfword align access.

**4.** Valid only during byte align access.

**Remarks 1.** W: Word data transfer cycle
HW: Halfword data transfer cycle
B: Byte data transfer cycle

**2.** n = 0, 1

For example, when write-accessing address 03FFEFF1H of the word data 44332211H, the first memory access means writing the byte data 11H to address 03FFEFF1H. A setting example when this access is specified as a break condition of channel 0 is shown below.

- BPAV0 register: 03FFEFF1H
- BPAM0 register: 00000000H
- BPDV0 register: xxxx11xxH (x: don't care)
- BPDM0 register: FFFF00FFH
- TY bit of BPC0 register: 0, 1 (byte access)

★

# APPENDIX A NOTES

## A.1 Restriction on Conflict Between sld Instruction and Interrupt request

### A.1.1 Description

If a conflict occurs between the decode operation of an instruction in <2> immediately before the sld instruction following an instruction in <1> and an interrupt request before the instruction in <1> is complete, the execution result of the instruction in <1> may not be stored in a register.

Instruction <1>

- ld instruction: ld.b, ld.h, ld.w, ld.bu, ld.hu
- sld instruction: sld.b, sld.h, sld.w, sld.bu, sld.hu
- Multiplication instruction: mul, mulh, mulhi, mulu

Instruction <2>

| | | | |
|---|---|---|---|
| mov reg1, reg2 | not reg1, reg2 | satsubr reg1, reg2 | satsub reg1, reg2 |
| satadd reg1, reg2 | satadd imm5, reg2 | or reg1, reg2 | xor reg1, reg2 |
| and reg1, reg2 | tst reg1, reg2 | subr reg1, reg2 | sub reg1, reg2 |
| add reg1, reg2 | add imm5, reg2 | cmp reg1, reg2 | cmp imm5, reg2 |
| mulh reg1, reg2 | shr imm5, reg2 | sar imm5, reg2 | shl imm5, reg2 |

<Example>

<i> ld.w [r11], r10
•
•
•
<ii> mov r10, r28
<iii> sld.w 0x28, r10

If the decode operation of the mov instruction <ii> immediately before the sld instruction <iii> and an interrupt request conflict before execution of the ld instruction <i> is complete, the execution result of instruction <i> may not be stored in a register.

### A.1.2 Countermeasure

When executing the sld instruction immediately after instruction <ii>, avoid the above operation using either of the following methods.

- Insert a nop instruction immediately before the sld instruction.
- Do not use the same register as the sld instruction destination register in the above instruction <ii> executed immediately before the sld instruction.

# APPENDIX B INSTRUCTION LIST

The instruction function list in alphabetical order is shown in Table B-1, and instruction list in format order is shown in Table B-2.

**Table B-1. Instruction Function List (in Alphabetical Order) (1/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| ADD | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | – | Add. Adds the word data of reg1 to the word data of reg2, and stores the result in reg2. |
| ADD | imm5, reg2 | II | 0/1 | 0/1 | 0/1 | 0/1 | – | Add. Adds the 5-bit immediate data, sign-extended to word length, to the word data of reg2, and stores the result in reg2. |
| ADDI | imm16, reg1, reg2 | VI | 0/1 | 0/1 | 0/1 | 0/1 | – | Add Immediate. Adds the 16-bit immediate data, sign-extended to word length, to the word data of reg1, and stores the result in reg2. |
| AND | reg1, reg2 | I | – | 0 | 0/1 | 0/1 | – | And. ANDs the word data of reg2 with the word data of reg1, and stores the result in reg2. |
| ANDI | imm16, reg1, reg2 | VI | – | 0 | 0 | 0/1 | – | And. ANDs the word data of reg1 with the 16-bit immediate data, zero-extended to word length, and stores the result in reg2. |
| Bcond | disp9 | III | – | – | – | – | – | Branch on Condition Code. Tests a condition flag specified by an instruction. Branches if a specified condition is satisfied; otherwise, executes the next instruction. The branch destination PC holds the sum of the current PC value and 9-bit displacement which is the 8-bit immediate shifted 1 bit and sign-extended to word length. |
| BSH | reg2, reg3 | XII | 0/1 | 0 | 0/1 | 0/1 | – | Byte Swap Halfword. Performs endian conversion. |
| BSW | reg2, reg3 | XII | 0/1 | 0 | 0/1 | 0/1 | – | Byte Swap Word. Performs endian conversion. |
| CALLT | imm6 | II | – | – | – | – | – | Call with Table Look Up. Based on CTBP contents, updates PC value and transfers control. |
| CLR1 | bit#3, disp16 [reg1] | VIII | – | – | – | 0/1 | – | Clear Bit. Adds the data of reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Then clears the bit, specified by the instruction bit field, of the byte data referenced by the generated address. |

**Table B-1. Instruction Function List (in Alphabetical Order) (2/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| CLR1 | reg2 [reg1] | IX | – | – | – | 0/1 | – | Clear Bit. First, reads the data of reg1 to generate a 32-bit address. Then clears the bit, specified by the data of lower 3 bits of reg2 of the byte data referenced by the generated address. |
| CMOV | cccc, reg1, reg2, reg3 | XI | – | – | – | – | – | Conditional Move. reg3 is set to reg1 if a condition specified by condition code “cccc” is satisfied; otherwise, set to the data of reg2. |
| CMOV | cccc, imm5, reg2, reg3 | XII | – | – | – | – | – | Conditional Move. reg3 is set to the data of 5-immediate, sign-extended to word length, if a condition specified by condition code “cccc” is satisfied; otherwise, set to the data of reg2. |
| CMP | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | – | Compare. Compares the word data of reg2 with the word data of reg1, and indicates the result by using the PSW flags. To compare, the contents of reg1 are subtracted from the word data of reg2. |
| CMP | imm5, reg2 | II | 0/1 | 0/1 | 0/1 | 0/1 | – | Compare. Compares the word data of reg2 with the 5-bit immediate data, sign-extended to word length, and indicates the result by using the PSW flags. To compare, the contents of the sign-extended immediate data are subtracted from the word data of reg2. |
| CTRET | (None) | X | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Restore from CALLT. Restores the restored PC and PSW from the appropriate system register and restores from a routine called by CALLT. |
| DBRET[Note] | (None) | X | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Return from debug trap. Restores the restored PC and PSW from the appropriate system register and restores from a debug monitor routine. |
| DBTRAP[Note] | (None) | I | – | – | – | – | – | Debug trap. Saves the restored PC and PSW to the appropriate system register and transfers control by setting the PC to handler address (00000060H). |
| DI | (None) | X | – | – | – | – | – | Disables Interrupt. Sets the ID flag of the PSW to 1 to disable the acknowledgment of maskable interrupts from acceptance; interrupts are immediately disabled at the start of this instruction execution. |
| DISPOSE | imm5, list12 | XIII | – | – | – | – | – | Function Dispose. Adds the data of 5-bit immediate imm5, logically shifted left by 2 and zero-extended to word length, to sp. Then pop (load data from the address specified by sp and adds 4 to sp) general-purpose registers listed in list12. |

★ **Note** Not supported in type C products

**Table B-1. Instruction Function List (in Alphabetical Order) (3/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| DISPOSE | imm5, list12, [reg1] | XIII | – | – | – | – | – | Function Dispose. Adds the data of 5-bit immediate imm5, logically shifted left by 2 and zero-extended to word length, to sp. Then pop (load data from the address specified by sp and adds 4 to sp) general-purpose registers listed in list12, transfers control to the address specified by reg1. |
| DIV | reg1, reg2, reg3 | XI | – | 0/1 | 0/1 | 0/1 | – | Divide Word. Divides the word data of reg2 by the word data of reg1, and stores the quotient in reg2 and the remainder in reg3. |
| DIVH | reg1, reg2 | I | – | 0/1 | 0/1 | 0/1 | – | Divide Halfword. Divides the word data of reg2 by the lower halfword data of reg1, and stores the quotient in reg2. |
| DIVH | reg1, reg2, reg3 | XI | – | 0/1 | 0/1 | 0/1 | – | Divide Halfword. Divides word data of reg2 by lower halfword data of reg1, and stores the quotient in reg2 and the remainder in reg3. |
| DIVHU | reg1, reg2, reg3 | XI | – | 0/1 | 0/1 | 0/1 | – | Divide Halfword Unsigned. Divides word data of reg2 by lower halfword data of reg1, and stores the quotient in reg2 and the remainder in reg3. |
| DIVU | reg1, reg2, reg3 | XI | – | 0/1 | 0/1 | 0/1 | – | Divide Word Unsigned. Divides the word data of reg2 by the word data of reg1, and stores the quotient in reg2 and the remainder in reg3. |
| EI | (None) | X | – | – | – | – | – | Enable Interrupt. Clears the ID flag of the PSW to 0 and enables the acknowledgment of maskable interrupts at the beginning of next instruction. |
| HALT | (None) | X | – | – | – | – | – | Halt. Stops the operating clock of the CPU and places the CPU in the HALT mode. |
| HSW | reg2, reg3 | XII | 0/1 | 0 | 0/1 | 0/1 | – | Halfword Swap Word. Performs endian conversion. |
| JARL | disp22, reg2 | V | – | – | – | – | – | Jump and Register Link. Saves the current PC value plus 4 to general-purpose register reg2, adds a 22-bit displacement, sign-extended to word length, to the current PC value, and transfers control to the PC. Bit 0 of the 22-bit displacement is masked to 0. |
| JMP | [reg1] | I | – | – | – | – | – | Jump Register. Transfers control to the address specified by reg1. Bit 0 of the address is masked to 0. |
| JR | disp22 | V | – | – | – | – | – | Jump Relative. Adds a 22-bit displacement, sign-extended to word length, to the current PC value, and transfers control to the PC. Bit 0 of the 22-bit displacement is masked to 0. |

**Table B-1. Instruction Function List (in Alphabetical Order) (4/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| LD.B | disp16 [reg1], reg2 | VII | – | – | – | – | – | Byte Load. Adds the data of reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Byte data is read from the generated address, sign-extended to word length, and then stored in reg2. |
| LD.BU | disp16 [reg1], reg2 | VII | – | – | – | – | – | Unsigned Byte Load. Adds the data of reg1 and the 16-bit displacement sign-extended to word length, and generates a 32-bit address. Then reads the byte data from the generated address, zero-extends it to word length, and stores it in reg2. |
| LD.H | disp16 [reg1], reg2 | VII | – | – | – | – | – | Halfword Load. Adds the data of reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Halfword data is read from this 32-bit address with bit 0 masked to 0, sign-extended to word length, and stored in reg2. |
| LD.HU | disp16 [reg1], reg2 | VII | – | – | – | – | – | Unsigned Halfword Load. Adds the data of reg1 and the 16-bit displacement sign-extended to word length to generate a 32-bit address. Reads the halfword data from the address masking bit 0 of this 32-bit address to 0, zero-extends it to word length, and stores it in reg2. |
| LD.W | disp16 [reg1], reg2 | VII | – | – | – | – | – | Word Load. Adds the data of reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Word data is read from this 32-bit address with bits 0 and 1 masked to 0, and stored in reg2. |
| LDSR | reg2, regID | IX | – | – | – | – | – | Load to System Register. Set the word data of reg2 to a system register specified by regID. If regID is PSW, the values of the corresponding bits of reg2 are set to the respective flags of the PSW. |
| MOV | reg1, reg2 | I | – | – | – | – | – | Move. Transfers the word data of reg1 in reg2. |
| MOV | imm5, reg2 | II | – | – | – | – | – | Move. Transfers the value of a 5-bit immediate data, sign-extended to word length, in reg2. |
| MOV | imm32, reg1 | VI | – | – | – | – | – | Move. Transfers the 32-bit immediate data in reg1. |
| MOVEA | imm16, reg1, reg2 | VI | – | – | – | – | – | Move Effective Address. Adds a 16-bit immediate data, sign-extended to word length, to the word data of reg1, and stores the result in reg2. |

**Table B-1. Instruction Function List (in Alphabetical Order) (5/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| MOVHI | imm16, reg1, reg2 | VI | – | – | – | – | – | Move High Halfword. Adds word data, in which the higher 16 bits are defined by the 16-bit immediate data while the lower 16 bits are set to 0, to the word data of reg1 and stores the result in reg2. |
| MUL | reg1, reg2, reg3 | XI | – | – | – | – | – | Multiply Word. Multiplies the word data of reg2 by the word data of reg1, and stores the result in reg2 and reg3. |
| MUL | imm9, reg2, reg3 | XII | – | – | – | – | – | Multiply Word. Multiplies the word data of reg2 by the 9-bit immediate data sign-extended to word length, and stores the result in reg2 and reg3. |
| MULH | reg1, reg2 | I | – | – | – | – | – | Multiply Halfword. Multiplies the lower halfword data of reg2 by the lower halfword data of reg1, and stores the result in reg2 as word data. |
| MULH | imm5, reg2 | II | – | – | – | – | – | Multiply Halfword. Multiplies the lower halfword data of reg2 by a 5-bit immediate data, sign-extended to halfword length, and stores the result in reg2 as word data. |
| MULHI | imm16, reg1, reg2 | VI | – | – | – | – | – | Multiply Halfword Immediate. Multiplies the lower halfword data of reg1 by a 16-bit immediate data, and stores the result in reg2. |
| MULU | reg1, reg2, reg3 | XI | – | – | – | – | – | Multiply Word Unsigned. Multiplies the word data of reg2 by the word data of reg1, and stores the result in reg2 and reg3. |
| MULU | imm9, reg2, reg3 | XII | – | – | – | – | – | Multiply Word Unsigned. Multiplies the word data of reg2 by the 9-bit immediate data sign-extended to word length, and store the result in reg2 and reg3. |
| NOP | (None) | I | – | – | – | – | – | No Operation. |
| NOT | reg1, reg2 | I | – | 0 | 0/1 | 0/1 | – | Not. Logically negates (takes 1's complement of) the word data of reg1, and stores the result in reg2. |
| NOT1 | bit#3, disp16 [reg1] | VIII | – | – | – | 0/1 | – | Not Bit. First, adds the data of reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. The bit specified by the 3-bit bit number is inverted at the byte data location referenced by the generated address. |
| NOT1 | reg2, [reg1] | IX | – | – | – | 0/1 | – | Not Bit. First, reads reg1 to generate a 32-bit address. The bit specified by the lower 3 bits of reg2 of the byte data of the generated address is inverted. |

**Table B-1. Instruction Function List (in Alphabetical Order) (6/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| OR | reg1, reg2 | I | – | 0 | 0/1 | 0/1 | – | Or. ORs the word data of reg2 with the word data of reg1, and stores the result in reg2. |
| ORI | imm16, reg1, reg2 | VI | – | 0 | 0/1 | 0/1 | – | Or Immediate. ORs the word data of reg1 with the 16-bit immediate data, zero-extended to word length, and stores the result in reg2. |
| PREPARE | list12, imm5 | XIII | – | – | – | – | – | Function Prepare. The general-purpose register displayed in list12 is saved (4 is subtracted from sp, and the data is stored in that address). Next, the data is logically shifted 2 bits to the left, and the 5-bit immediate data zero-extended to word length is subtracted from sp. |
| PREPARE | list12, imm5, sp/imm | XIII | – | – | – | – | – | Function Prepare. The general-purpose register displayed in list12 is saved (4 is subtracted from sp, and the data is stored in that address). Next, the data is logically shifted 2 bits to the left, and the 5-bit immediate data zero-extended to word length is subtracted from sp. Then, the data specified by the third operand is loaded to ep. |
| RETI | (None) | X | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Return from Trap or Interrupt. Reads the restored PC and PSW from the appropriate system register, and restores from interrupt or exception processing routine. |
| SAR | reg1, reg2 | IX | 0/1 | 0 | 0/1 | 0/1 | – | Shift Arithmetic Right. Arithmetically shifts the word data of reg2 to the right by ‘n’ positions, where ‘n’ is specified by the lower 5 bits of reg1 (the MSB prior to shift execution is copied and set as the new MSB), and then writes the result in reg2. |
| SAR | imm5, reg2 | II | 0/1 | 0 | 0/1 | 0/1 | – | Shift Arithmetic Right. Arithmetically shifts the word data of reg2 to the right by ‘n’ positions specified by the lower 5-bit immediate data, zero-extended to word length (the MSB prior to shift execution is copied and set as the new MSB), and then writes the result in reg2. |
| SASF | cccc, reg2 | IX | – | – | – | – | – | Shift and Set Flag Condition. reg2 is logically shifted left by 1, and its LSB is set to 1 in a condition specified by condition code “cccc” is satisfied; otherwise, LSB is set to 0. |

**Table B-1. Instruction Function List (in Alphabetical Order) (7/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| SATADD | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Saturated Add. Adds the word data of reg1 to the word data of reg2, and stores the result in reg2. However, if the result exceeds the maximum positive value, the maximum positive value is stored in reg2; if the result exceeds the maximum negative value, the maximum negative value is stored in reg2. The SAT flag is set to 1. |
| SATADD | imm5, reg2 | II | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Saturated Add. Adds the 5-bit immediate data, sign-extended to word length, to the word data of reg2, and stores the result in reg2. However, if the result exceeds the maximum positive value, the maximum positive value is stored in reg2; if the result exceeds the maximum negative value, the maximum negative value is stored in reg2. The SAT flag is set to 1. |
| SATSUB | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Saturated Subtract. Subtracts the word data of reg1 from the word data of reg2, and stores the result in reg2. However, if the result exceeds the maximum positive value, the maximum positive value is stored in reg2; if the result exceeds the maximum negative value, the maximum negative value is stored in reg2. The SAT flag is set to 1. |
| SATSUBI | imm16, reg1, reg2 | VI | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Saturated Subtract Immediate. Subtracts a 16-bit immediate data, sign-extended to word length, from the word data of reg1, and stores the result in reg2. However, if the result exceeds the maximum positive value, the maximum positive value is stored in reg2; if the result exceeds the maximum negative value, the maximum negative value is stored in reg2. The SAT flag is set to 1. |
| SATSUBR | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | 0/1 | Saturated Subtract Reverse. Subtracts the word data of reg2 from the word data of reg1, and stores the result in reg2. However, if the result exceeds the maximum positive value, the maximum positive value is stored in reg2; if the result exceeds the maximum negative value, the maximum negative value is stored in reg2. The SAT flag is set to 1. |
| SET1 | bit#3, disp16 [reg1] | VIII | – | – | – | 0/1 | – | Set Bit. First, adds a 16-bit displacement, sign-extended to word length, to the data of reg1 to generate a 32-bit address. The bits, specified by the 3-bit bit number, are set at the byte data location specified by the generated address. |

**Table B-1. Instruction Function List (in Alphabetical Order) (8/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| SET1 | reg2, [reg1] | IX | – | – | – | 0/1 | – | Set Bit. First, reads the data of general-purpose register reg1 to generate a 32-bit address. The bit, specified by the data of lower 3 bits of reg2, is set at the byte data location referenced by the generated address. |
| SETF | cccc, reg2 | IX | – | – | – | – | – | Set Flag Condition. The reg2 is set to 1 if a condition specified by condition code "cccc" is satisfied; otherwise, a 0 is stored in reg2. |
| SHL | reg1, reg2 | IX | 0/1 | 0 | 0/1 | 0/1 | – | Shift Logical Left. Logically shifts the word data of reg2 to the left by ‘n’ positions (0 is shifted to the LSB side), where ‘n’ is specified by the lower 5 bits of reg1, and then writes the result in reg2. |
| SHL | imm5, reg2 | II | 0/1 | 0 | 0/1 | 0/1 | – | Shift Logical Left. Logically shifts the word data of reg2 to the left by ‘n’ positions (0 is shifted to the LSB side), where ‘n’ is specified by a 5-bit immediate data, zero-extended to word length, and then writes the result in reg2. |
| SHR | reg1, reg2 | IX | 0/1 | 0 | 0/1 | 0/1 | – | Shift Logical Right. Logically shifts the word data of reg2 to the right by ‘n’ positions (0 is shifted to the MSB side), where ‘n’ is specified by the lower 5 bits of reg1, and then writes the result in reg2. |
| SHR | imm5, reg2 | II | 0/1 | 0 | 0/1 | 0/1 | – | Shift Logical Right. Logically shifts the word data of reg2 to the right by ‘n’ positions (0 is shifted to the MSB side), where ‘n’ is specified by a 5-bit immediate data, zero-extended to word length, and then writes the result in reg2. |
| SLD.B | disp7 [ep], reg2 | IV | – | – | – | – | – | Byte Load. Adds the 7-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Byte data is read from the generated address, sign-extended to word length, and then stored in reg2. |
| SLD.BU | disp4 [ep], reg2 | IV | – | – | – | – | – | Unsigned Byte Load. Adds the 4-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Byte data is read from the generated address, zero-extended to word length, and stored in reg2. |
| SLD.H | disp8 [ep], reg2 | IV | – | – | – | – | – | Halfword Load. Adds the 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Halfword data is read from this 32-bit address with bit 0 masked to 0, sign-extended to word length, and stored in reg2. |

**Table B-1. Instruction Function List (in Alphabetical Order) (9/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| SLD.HU | disp5 [ep], reg2 | IV | – | – | – | – | – | Unsigned Halfword Load. Adds the 5-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Halfword data is read from this 32-bit address with bit 0 masked to 0, zero-extended to word length, and stored in reg2. |
| SLD.W | disp8 [ep], reg2 | IV | – | – | – | – | – | Word Load. Adds the 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address. Word data is read from this 32-bit address with bits 0 and 1 masked to 0, and stored in reg2. |
| SST.B | reg2, disp7 [ep] | IV | – | – | – | – | – | Byte Store. Adds the 7-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address, and stores the data of the lowest byte of reg2 in the generated address. |
| SST.H | reg2, disp8 [ep] | IV | – | – | – | – | – | Halfword Store. Adds the 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address, and stores the lower halfword of reg2 in the generated 32-bit address with bit 0 masked to 0. |
| SST.W | reg2, disp8 [ep] | IV | – | – | – | – | – | Word Store. Adds the 8-bit displacement, zero-extended to word length, to the element pointer to generate a 32-bit address, and stores the word data of reg2 in the generated 32-bit address with bits 0 and 1 masked to 0. |
| ST.B | reg2, disp16 [reg1] | VII | – | – | – | – | – | Byte Store. Adds the 16-bit displacement, sign-extended to word length, to the data of reg1 to generate a 32-bit address, and stores the lowest byte data of reg2 in the generated address. |
| ST.H | reg2, disp16 [reg1] | VII | – | – | – | – | – | Halfword Store. Adds the 16-bit displacement, sign-extended to word length, to the data of reg1 to generate a 32-bit address, and stores the lower halfword of reg2 in the generated 32-bit address with bit 0 masked to 0. |
| ST.W | reg2, disp16 [reg1] | VII | – | – | – | – | – | Word Store. Adds the 16-bit displacement, sign-extended to word length, to the data of reg1 to generate a 32-bit address, and stores the word data of reg2 in the generated 32-bit address with bits 0 and 1 masked to 0. |
| STSR | regID, reg2 | IX | – | – | – | – | – | Store Contents of System Register. Stores the contents of a system register specified by regID in reg2. |

**Table B-1. Instruction Function List (in Alphabetical Order) (10/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| SUB | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | – | Subtract. Subtracts the word data of reg1 from the word data of reg2, and stores the result in reg2. |
| SUBR | reg1, reg2 | I | 0/1 | 0/1 | 0/1 | 0/1 | – | Subtract Reverse. Subtracts the word data of reg2 from the word data of reg1, and stores the result in reg2. |
| SWITCH | reg1 | I | – | – | – | – | – | Jump with Table Look Up. Adds the table entry address (address following SWITCH instruction) and data of reg1 logically shifted to the left by 1 bit, and loads the halfword entry data specified by the table entry address. Next, logically shifts to the left by 1 bit the loaded data, and after sign-extending it to word length, branches to the target address added to the table entry address (instruction following SWITCH instruction). |
| SXB | reg1 | I | – | – | – | – | – | Sign Extend Byte. Sign-extends the lowermost byte of reg1 to word length. |
| SXH | reg1 | I | – | – | – | – | – | Sign Extend Halfword. Sign-extends lower halfword of reg1 to word length. |
| TRAP | vector | X | – | – | – | – | – | Trap. Saves the restored PC and PSW; sets the exception code and the flags of the PSW; jumps to the address of the trap handler corresponding to the trap vector specified by vector, and starts exception processing. |
| TST | reg1, reg2 | I | – | 0 | 0/1 | 0/1 | – | Test. ANDs the word data of reg2 with the word data of reg1. The result is not stored, and only the flags are changed. |
| TST1 | bit#3, disp16 [reg1] | VIII | – | – | – | 0/1 | – | Test Bit. Adds the data of reg1 to a 16-bit displacement, sign-extended to word length, to generate a 32-bit address. Performs the test on the bit, specified by the 3-bit bit number, at the byte data location referenced by the generated address. If the specified bit is 0, the Z flag is set to 1; if the bit is 1, the Z flag is cleared to 0. |
| TST1 | reg2, [reg1] | IX | – | – | – | 0/1 | – | Test Bit. First, reads the data of reg1 to generate a 32-bit address. If the bits indicated by the lower 3 bits of reg2 of the byte data of the generated address are 0, the Z flag is set to 1, and if they are 1, the Z flag is cleared to 0. |
| XOR | reg1, reg2 | I | – | 0 | 0/1 | 0/1 | – | Exclusive Or. Exclusively ORs the word data of reg2 with the word data of reg1, and stores the result in reg2. |

**Table B-1. Instruction Function List (in Alphabetical Order) (11/11)**

| Mnemonic | Operand | Format | Flag | | | | | Instruction Function |
|---|---|---|---|---|---|---|---|---|
| | | | CY | OV | S | Z | SAT | |
| XORI | imm16, reg1, reg2 | VI | – | 0 | 0/1 | 0/1 | – | Exclusive Or Immediate. Exclusively ORs the word data of reg1 with a 16-bit immediate data, zero-extended to word length, and stores the result in reg2. |
| ZXB | reg1 | I | – | – | – | – | – | Zero Extend Byte. Zero-extends to word length the lowest byte of reg1. |
| ZXH | reg1 | I | – | – | – | – | – | Zero Extend Halfword. Zero-extends to word length the lower halfword of reg1. |

**Table B-2. Instruction List (in Format Order) (1/3)**

| Format | Opcode | | Mnemonic | Operand |
|---|---|---|---|---|
| | 15 0 | 31 16 | | |
| I | 0000000000000000 | – | NOP | – |
| | rrrrr000000RRRRR | – | MOV | reg1, reg2 |
| | rrrrr000001RRRRR | – | NOT | reg1, reg2 |
| | rrrrr000010RRRRR | – | DIVH | reg1, reg2 |
| | 0000000000010RRRRR | – | SWITCH | reg1 |
| | 0000000000011RRRRR | – | JMP | [reg1] |
| | rrrrr000100RRRRR | – | SATSUBR | reg1, reg2 |
| | rrrrr000101RRRRR | – | SATSUB | reg1, reg2 |
| | rrrrr000110RRRRR | – | SATADD | reg1, reg2 |
| | rrrrr000111RRRRR | – | MULH | reg1, reg2 |
| | 0000000000100RRRRR | – | ZXB | reg1 |
| | 0000000000101RRRRR | – | SXB | reg1 |
| | 0000000000110RRRRR | – | ZXH | reg1 |
| | 0000000000111RRRRR | – | SXH | reg1 |
| | rrrrr001000RRRRR | – | OR | reg1, reg2 |
| | rrrrr001001RRRRR | – | XOR | reg1, reg2 |
| | rrrrr001010RRRRR | – | AND | reg1, reg2 |
| | rrrrr001011RRRRR | – | TST | reg1, reg2 |
| | rrrrr001100RRRRR | – | SUBR | reg1, reg2 |
| | rrrrr001101RRRRR | – | SUB | reg1, reg2 |
| | rrrrr001110RRRRR | – | ADD | reg1, reg2 |
| | rrrrr001111RRRRR | – | CMP | reg1, reg2 |
| | 1111100001000000 | – | DBTRAP[Note] | – |
| II | rrrrr010000iiiii | – | MOV | imm5, reg2 |
| | rrrrr010001iiiii | – | SATADD | imm5, reg2 |
| | rrrrr010010iiiii | – | ADD | imm5, reg2 |
| | rrrrr010011iiiii | – | CMP | imm5, reg2 |
| | 0000001000iiiiii | – | CALLT | imm6 |
| | rrrrr010100iiiii | – | SHR | imm5, reg2 |
| | rrrrr010101iiiii | – | SAR | imm5, reg2 |
| | rrrrr010110iiiii | – | SHL | imm5, reg2 |
| | rrrrr010111iiiii | – | MULH | imm5, reg2 |
| III | ddddd1011dddCCCC | – | Bcond | disp9 |

★ **Note** Not supported in type C products

**Table B-2. Instruction List (in Format Order) (2/3)**

| Format | Opcode | | Mnemonic | Operand |
|---|---|---|---|---|
| | 15 0 | 31 16 | | |
| IV | rrrrr0000110dddd | – | SLD.BU | disp4 [ep], reg2 |
| | rrrrr0000111dddd | – | SLD.HU | disp5 [ep], reg2 |
| | rrrrr0110ddddddd | – | SLD.B | disp7 [ep], reg2 |
| | rrrrr0111ddddddd | – | SST.B | reg2, disp7 [ep] |
| | rrrrr1000ddddddd | – | SLD.H | disp8 [ep], reg2 |
| | rrrrr1001ddddddd | – | SST.H | reg2, disp8 [ep] |
| | rrrrr1010ddddddd0 | – | SLD.W | disp8 [ep], reg2 |
| | rrrrr1010ddddddd1 | – | SST.W | reg2, disp8 [ep] |
| V | rrrrr11110dddddd | ddddddddddddddd0 | JARL | disp22, reg2 |
| | 0000011110dddddd | ddddddddddddddd0 | JR | disp22 |
| VI | rrrrr110000RRRRR | iiiiiiiiiiiiiiii | ADDI | imm16, reg1, reg2 |
| | rrrrr110001RRRRR | iiiiiiiiiiiiiiii | MOVEA | imm16, reg1, reg2 |
| | rrrrr110010RRRRR | iiiiiiiiiiiiiiii | MOVHI | imm16, reg1, reg2 |
| | rrrrr110011RRRRR | iiiiiiiiiiiiiiii | SATSUBI | imm16, reg1, reg2 |
| | 0000011000 1RRRRR | **Note** | MOV | imm32, reg1 |
| | rrrrr110100RRRRR | iiiiiiiiiiiiiiii | ORI | imm16, reg1, reg2 |
| | rrrrr110101RRRRR | iiiiiiiiiiiiiiii | XORI | imm16, reg1, reg2 |
| | rrrrr110110RRRRR | iiiiiiiiiiiiiiii | ANDI | imm16, reg1, reg2 |
| | rrrrr110111RRRRR | iiiiiiiiiiiiiiii | MULHI | imm16, reg1, reg2 |
| VII | rrrrr111000RRRRR | dddddddddddddddd | LD.B | disp16 [reg1], reg2 |
| | rrrrr111001RRRRR | ddddddddddddddd0 | LD.H | disp16 [reg1], reg2 |
| | rrrrr111001RRRRR | ddddddddddddddd1 | LD.W | disp16 [reg1], reg2 |
| | rrrrr111010RRRRR | dddddddddddddddd | ST.B | reg2, disp16 [reg1] |
| | rrrrr111011RRRRR | ddddddddddddddd0 | ST.H | reg2, disp16 [reg1] |
| | rrrrr111011RRRRR | ddddddddddddddd1 | ST.W | reg2, disp16 [reg1] |
| | rrrrr11110bRRRRR | ddddddddddddddd1 | LD.BU | disp16 [reg1], reg2 |
| | rrrrr111111RRRRR | ddddddddddddddd1 | LD.HU | disp16 [reg1], reg2 |
| VIII | 00bbb111110RRRRR | dddddddddddddddd | SET1 | bit#3, disp16 [reg1] |
| | 01bbb111110RRRRR | dddddddddddddddd | NOT1 | bit#3, disp16 [reg1] |
| | 10bbb111110RRRRR | dddddddddddddddd | CLR1 | bit#3, disp16 [reg1] |
| | 11bbb111110RRRRR | dddddddddddddddd | TST1 | bit#3, disp16 [reg1] |

**Note** 32-bit immediate data. The higher 32 bits (bits 16 to 47) are as follows.

| 31 | 47 |
|---|---|
| iiiiiiiiiiiiiiii | IIIIIIIIIIIIIIII |

**Table B-2. Instruction List (in Format Order) (3/3)**

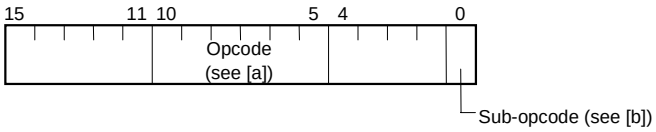
| Format | Opcode | | Mnemonic | Operand |
|---|---|---|---|---|
| | 15 0 | 31 16 | | |
| IX | rrrrr1111110cccc | 0000000000000000 | SETF | cccc, reg2 |
| | rrrrr111111RRRRR | 0000000000100000 | LDSR | reg2, regID |
| | rrrrr111111RRRRR | 0000000001000000 | STSR | regID, reg2 |
| | rrrrr111111RRRRR | 0000000010000000 | SHR | reg1, reg2 |
| | rrrrr111111RRRRR | 0000000010100000 | SAR | reg1, reg2 |
| | rrrrr111111RRRRR | 0000000011000000 | SHL | reg1, reg2 |
| | rrrrr111111RRRRR | 0000000011100000 | SET1 | reg2, [reg1] |
| | rrrrr111111RRRRR | 0000000011100010 | NOT1 | reg2, [reg1] |
| | rrrrr111111RRRRR | 0000000011100100 | CLR1 | reg2, [reg1] |
| | rrrrr111111RRRRR | 0000000011100110 | TST1 | reg2, [reg1] |
| | rrrrr1111110cccc | 0000001000000000 | SASF | cccc, reg2 |
| X | 0000011111iiiii | 0000000100000000 | TRAP | vector |
| | 0000011111100000 | 0000000100100000 | HALT | – |
| | 0000011111100000 | 0000000101000000 | RETI | – |
| | 0000011111100000 | 0000000101000100 | CTRET | – |
| | 0000011111100000 | 0000000101000110 | DBRET[Note] | – |
| | 0000011111100000 | 0000000101100000 | DI | – |
| | 1000011111100000 | 0000000101100000 | EI | – |
| XI | rrrrr111111RRRRR | wwwww01000100000 | MUL | reg1, reg2, reg3 |
| | rrrrr111111RRRRR | wwwww01000100010 | MULU | reg1, reg2, reg3 |
| | rrrrr111111RRRRR | wwwww01010000000 | DIVH | reg1, reg2, reg3 |
| | rrrrr111111RRRRR | wwwww01010000010 | DIVHU | reg1, reg2, reg3 |
| | rrrrr111111RRRRR | wwwww01011000000 | DIV | reg1, reg2, reg3 |
| | rrrrr111111RRRRR | wwwww01011000010 | DIVU | reg1, reg2, reg3 |
| | rrrrr111111RRRRR | wwwww011001cccc0 | CMOV | cccc, reg1, reg2, reg3 |
| XII | rrrrr111111iiiii | wwwww01001IIII00 | MUL | imm9, reg2, reg3 |
| | rrrrr111111iiiii | wwwww01001IIII10 | MULU | imm9, reg2, reg3 |
| | rrrrr111111iiiii | wwwww011000cccc0 | CMOV | cccc, imm5, reg2, reg3 |
| | rrrrr11111100000 | wwwww01101000000 | BSW | reg2, reg3 |
| | rrrrr11111100000 | wwwww01101000010 | BSH | reg2, reg3 |
| | rrrrr11111100000 | wwwww01101000100 | HSW | reg2, reg3 |
| XIII | 0000011001iiiiiL | LLLLLLLLLLLRRRRR | DISPOSE | imm5, list12, [reg1] |
| | 0000011001iiiiiL | LLLLLLLLLLL00000 | DISPOSE | imm5, list12 |
| | 0000011110iiiiiL | LLLLLLLLLLL00001 | PREPARE | list12, imm5 |
| | 0000011110iiiiiL | LLLLLLLLLLLff011 | PREPARE | list12, imm5, sp/imm |

★ **Note** Not supported in type C products

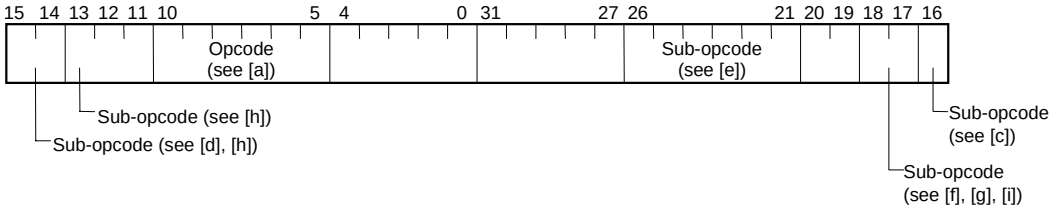
# APPENDIX C INSTRUCTION OPCODE MAP

This chapter shows the opcode map for the instruction code shown below.

### (1) 16-bit format instruction




### (2) 32-bit format instruction




**Remark** Operand convention

| Symbol | Meaning |
|---|---|
| R | reg1: General-purpose register (used as source register) |
| r | reg2: General-purpose register (mainly used as destination register. Some are also used as source registers.) |
| w | reg3: General-purpose register (mainly used as remainder of division results or higher 32 bits of multiply results) |
| bit#3 | 3-bit data for bit number specification |
| imm× | ×-bit immediate data |
| disp× | ×-bit displacement data |
| cccc | 4-bit data condition code specification |

**[a] Opcode**

| Bit 10 | Bit 9 | Bit 8 | Bit 7 | Bits 6, 5 0,0 | Bits 6, 5 0,1 | Bits 6, 5 1,0 | Bits 6, 5 1,1 | Format |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | MOV R, r<br>NOP[Note 1] | NOT | DIVH<br>SWITCH[Note 2]<br>DBTRAP<br>Undefined[Note 3] | JMP[Note 4]<br>SLD.BU[Note 5]<br>SLD.HU[Note 6] | I, IV |
| 0 | 0 | 0 | 1 | SATSUBR<br>ZXB[Note 4] | SATSUB<br>SXB[Note 4] | SATADD R, r<br>ZXH[Note 4] | MULH<br>SXH[Note 4] | I |
| 0 | 0 | 1 | 0 | OR | XOR | AND | TST | |
| 0 | 0 | 1 | 1 | SUBR | SUB | ADD R, r | CMP R, r | |
| 0 | 1 | 0 | 0 | MOV imm5, r<br>CALLT[Note 4] | SATADD imm5, r | ADD imm5, r | CMP imm5, r | II |
| 0 | 1 | 0 | 1 | SHR imm5, r | SAR imm5, r | SHL imm5, r | MULH imm5, r<br>Undefined[Note 4] | |
| 0 | 1 | 1 | 0 | SLD.B | | | | IV |
| 0 | 1 | 1 | 1 | SST.B | | | | |
| 1 | 0 | 0 | 0 | SLD.H | | | | |
| 1 | 0 | 0 | 1 | SST.H | | | | |
| 1 | 0 | 1 | 0 | SLD.W[Note 7]<br>SST.W[Note 7] | | | | |
| 1 | 0 | 1 | 1 | Bcond | | | | III |
| 1 | 1 | 0 | 0 | ADDI | MOVEA<br>MOV imm32, R[Note 4] | MOVHI<br>DISPOSE[Note 4] | SATSUBI | VI, XIII |
| 1 | 1 | 0 | 1 | ORI | XORI | ANDI | MULHI<br>Undefined[Note 4] | VI |
| 1 | 1 | 1 | 0 | LD.B | LD.H[Note 8]<br>LD.W[Note 8] | ST.B | ST.H[Note 8]<br>ST.W[Note 8] | VII |
| 1 | 1 | 1 | 1 | JR<br>JARL<br>LD.BU[Note 10]<br>PREPARE[Note 11] | | Bit manipulation 1[Note 9] | LD.HU[Note 10]<br>Undefined[Note 11]<br>Expansion 1[Note 12] | V, VII, VIII, XIII |

**Notes 1.** If R (reg1) = r0 and r (reg2) = r0 (instruction without reg1 and reg2)

**2.** If R (reg1) ≠ r0 and r (reg2) = r0 (instruction with reg1 and without reg2)

**3.** If R (reg1) = r0 and r (reg2) ≠ r0 (instruction without reg1 and with reg2)

**4.** If R (reg2) = r0 (instruction without reg2)

**5.** If bit 4 = 0 and r (reg2) ≠ r0 (instruction with reg2)

**6.** If bit 4 = 1 and r (reg2) ≠ r0 (instruction with reg2)

**7.** See **[b]**

**8.** See **[c]**

**9.** See **[d]**

**10.** If bit 16 = 1 and r (reg2) ≠ r0 (instruction with reg2)

**11.** If bit 16 = 1 and r (reg2) = r0 (instruction without reg2)

**12.** See **[e]**

★ **Remark** Type C products do not support the DBTRAP instruction.

**[b] Short format load/store instruction (displacement/sub-opcode)**

| Bit 10 | Bit 9 | Bit 8 | Bit 7 | Bit 0 | |
|---|---|---|---|---|---|
| | | | | 0 | 1 |
| 0 | 1 | 1 | 0 | SLD.B | |
| 0 | 1 | 1 | 1 | SST.B | |
| 1 | 0 | 0 | 0 | SLD.H | |
| 1 | 0 | 0 | 1 | SST.H | |
| 1 | 0 | 1 | 0 | SLD.W | SST.W |

**[c] Load/store instruction (displacement/sub-opcode)**

| Bit 6 | Bit 5 | Bit 16 | |
|---|---|---|---|
| | | 0 | 1 |
| 0 | 0 | LD.B | |
| 0 | 1 | LD.H | LD.W |
| 1 | 0 | ST.B | |
| 1 | 1 | ST.H | ST.W |

**[d] Bit manipulation instruction 1 (sub-opcode)**

| Bit 15 | Bit 14 | |
|---|---|---|
| | 0 | 1 |
| 0 | SET1 bit#3, disp16 [R] | NOT1 bit#3, disp16 [R] |
| 1 | CLR1 bit#3, disp16 [R] | TST1 bit#3, disp16 [R] |

### [e] Expansion 1 (sub-opcode)

| Bit 26 | Bit 25 | Bit 24 | Bit 23 | Bits 22, 21 | | | | Format |
|---|---|---|---|---|---|---|---|---|
| | | | | 0,0 | 0,1 | 1,0 | 1,1 | |
| 0 | 0 | 0 | 0 | SETF | LDSR | STSR | Undefined | IX |
| 0 | 0 | 0 | 1 | SHR | SAR | SHL | Bit manipulation 2[Note 1] | |
| 0 | 0 | 1 | 0 | TRAP | HALT | RETI[Note 2]<br>CTRET[Note 2]<br>DBRET[Note 2]<br>Undefined | EI[Note 3]<br>DI[Note 3]<br>Undefined | X |
| 0 | 0 | 1 | 1 | Undefined | | Undefined | | – |
| 0 | 1 | 0 | 0 | SASF | MUL R, r, w<br>MULU R, r, w[Note 4] | MUL imm9, r, w<br>MULU imm9, r, w[Note 4] | | IX, XI, XII |
| 0 | 1 | 0 | 1 | DIVH<br>DIVHU[Note 4] | | DIV<br>DIVU[Note 4] | | XI |
| 0 | 1 | 1 | 0 | CMOV<br>cccc, imm5, r, w | CMOV<br>cccc, R, r, w | BSW[Note 5]<br>BSH[Note 5]<br>HSW[Note 5] | Undefined | XI, XII |
| 0 | 1 | 1 | 1 | Illegal instruction | | | | – |
| 1 | x | x | x | | | | | |

**Notes 1.** See **[f]**

**2.** See **[g]**

**3.** See **[h]**

**4.** If bit 17 = 1

**5.** See **[i]**

★ **Remark** Type C products do not support the DBRET instruction.

### [f] Bit manipulation instruction 2 (sub-opcode)

| Bit 18 | Bit 17 | |
|---|---|---|
| | 0 | 1 |
| 0 | SET1 r, [R] | NOT1 r, [R] |
| 1 | CLR1 r, [R] | TST1 r, [R] |

### [g] Return instruction (sub-opcode)

| Bit 18 | Bit 17 | |
|---|---|---|
| | 0 | 1 |
| 0 | RETI | Undefined |
| 1 | CTRET | DBRET |

### [h] PSW operation instruction (sub-opcode)

| Bit 15 | Bit 14 | Bits 13, 12, 11 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0,0,0 | 0,0,1 | 0,1,0 | 0,1,1 | 1,0,0 | 1,0,1 | 1,1,0 | 1,1,1 |
| 0 | 0 | DI | Undefined | | | | | | |
| 0 | 1 | Undefined | | | | | | | |
| 1 | 0 | EI | Undefined | | | | | | |
| 1 | 1 | Undefined | | | | | | | |

### [i] Endian conversion instruction (sub-opcode)

| Bit 18 | Bit 17 | |
|---|---|---|
| | 0 | 1 |
| 0 | BSW | BSH |
| 1 | HSW | Undefined |

# APPENDIX D DIFFERENCES WITH ARCHITECTURE OF V850 CPU

(1/2)

| Item | | V850E1 CPU | V850 CPU |
|---|---|---|---|
| Instructions (including operand) | BSH reg2, reg3 | Provided | Not provided |
| | BSW reg2, reg3 | | |
| | CALLT imm6 | | |
| | CLR1 reg2, [reg1] | | |
| | CMOV cccc, imm5, reg2, reg3 | | |
| | CMOV cccc, reg1, reg2, reg3 | | |
| | CTRET | | |
| | DBRET[Note] | | |
| | DBTRAP[Note] | | |
| | DISPOSE imm5, list12 | | |
| | DISPOSE imm5, list12 [reg1] | | |
| | DIV reg1, reg2, reg3 | | |
| | DIVH reg1, reg2, reg3 | | |
| | DIVHU reg1, reg2, reg3 | | |
| | DIVU reg1, reg2, reg3 | | |
| | HSW reg2, reg3 | | |
| | LD.BU disp16 [reg1], reg2 | | |
| | LD.HU disp16 [reg1], reg2 | | |
| | MOV imm32, reg1 | | |
| | MUL imm9, reg2, reg3 | | |
| | MUL reg1, reg2, reg3 | | |
| | MULU reg1, reg2, reg3 | | |
| | MULU imm9, reg2, reg3 | | |
| | NOT1 reg2, [reg1] | | |
| | PREPARE list12, imm5 | | |
| | PREPARE list12, imm5, sp/imm | | |
| | SASF cccc, reg2 | | |
| | SET1 reg2, [reg1] | | |
| | SLD.BU disp4 [ep], reg2 | | |
| | SLD.HU disp5 [ep], reg2 | | |
| | SWITCH reg1 | | |
| | SXB reg1 | | |
| | SXH reg1 | | |
| | TST1 reg2, [reg1] | | |
| | ZXB reg1 | | |
| | ZXH reg1 | | |

★ **Note** Not supported in type C products

(2/2)

| Item | | V850E1 CPU | V850 CPU |
|---|---|---|---|
| Instruction format | Format IV | Format is different for some instructions. | |
| | Format XI | Provided | Not provided |
| | Format XII | | |
| | Format XIII | | |
| Instruction execution clocks | | Value differs for some instructions. | |
| Program space | | 64 MB linear | 16 MB linear |
| Valid bits of program counter (PC) | | Lower 26 bits | Lower 24 bits |
| System register | CALLT execution status saving registers (CTPC, CTPSW) | Provided | Not provided |
| | Exception/debug trap status saving registers (DBPC, DBPSW) | | |
| | CALLT base pointer (CTBP) | | |
| | Debug interface register (DIR)[Note 1] | | |
| | Breakpoint control registers 0 and 1 (BPC0, BPC1)[Note 1] | | |
| | Program ID register (ASID)[Note 1] | | |
| | Breakpoint address setting registers 0 and 1 (BPAV0, BPAV1)[Note 1] | | |
| | Breakpoint address mask registers 0 and 1 (BPAM0, BPAM1)[Note 1] | | |
| | Breakpoint data setting registers 0 and 1 (BPDV0, BPDV1)[Note 1] | | |
| | Breakpoint data mask registers 0 and 1 (BPDM0, BPDM1)[Note 1] | | |
| | Exception trap status saving registers | DBPC, DBPSW | EIPC, EIPSW |
| Illegal instruction code | | Instruction code areas differ. | |
| Misaligned access enable/disable setting | | Can be set depending on product | Cannot be set. (misaligned access disabled) |
| ★ Non-maskable interrupt (NMI) | Input | 3 (type A, B, C products) | 1 |
| ★ | | 1 (type D, E, F products) | |
| | Exception code | 0010H, 0020H, 0030H | 0010H |
| | Handler address | 00000010H, 00000020H, 00000030H | 00000010H |
| Debug trap[Note 2] | | Provided | Not provided |
| Pipeline | | At next instruction, pipeline flow differs.<br>• Arithmetic operation instruction<br>• Branch instruction<br>• Bit manipulation instruction<br>• Special instruction (TRAP, RETI) | |

★ **Notes 1.** Used only in type A and B products

★ **2.** Not supported in type C products

# APPENDIX E INSTRUCTIONS ADDED FOR V850E1 CPU COMPARED WITH V850 CPU

Compared with the instruction codes of the V850 CPU, the instruction codes of the V850E1 CPU are upward compatible at the object code level. In the case of the V850E1 CPU, instructions that even if executed have no meaning in the case of the V850 CPU (mainly instructions performing write to the r0 register) are extended as additional instructions.

The following table shows the V850 CPU instructions corresponding to the instruction codes added in the V850E1 CPU. See the table when switching from products that incorporate the V850 CPU to products that incorporate the V850E1 CPU.

**Table E-1. Instructions Added to V850E1 CPU and V850 CPU Instructions with Same Instruction Code (1/2)**

| Instructions Added in V850E1 CPU | V850 CPU Instructions with Same Instruction Code as V850E1 CPU |
|---|---|
| CALLT imm6 | MOV imm5, r0 or SATADD imm5, r0 |
| DISPOSE imm5, list12 | MOVHI imm16, reg1, r0 or SATSUBI imm16, reg1, r0 |
| DISPOSE imm5, list12 [reg1] | MOVHI imm16, reg1, r0 or SATSUBI imm16, reg1, r0 |
| MOV imm32, reg1 | MOVEA imm16, reg1, r0 |
| SWITCH reg1 | DIVH reg1, r0 |
| SXB reg1 | SATSUB reg1, r0 |
| SXH reg1 | MULH reg1, r0 |
| ZXB reg1 | SATSUBR reg1, r0 |
| ZXH reg1 | SATADD reg1, r0 |
| (RFU) | MULH imm5, r0 |
| (RFU) | MULHI imm16, reg1, r0 |
| BSH reg2, reg3 | Illegal instruction |
| BSW reg2, reg3 | |
| CMOV cccc, imm5, reg2, reg3 | |
| CMOV cccc, reg1, reg2, reg3 | |
| CTRET | |
| DIV reg1, reg2, reg3 | |
| DIVH reg1, reg2, reg3 | |
| DIVHU reg1, reg2, reg3 | |
| DIVU reg1, reg2, reg3 | |
| HSW reg2, reg3 | |
| MUL imm9, reg2, reg3 | |
| MUL reg1, reg2, reg3 | |
| MULU reg1, reg2, reg3 | |
| MULU imm9, reg2, reg3 | |
| SASF cccc, reg2 | |

**Table E-1. Instructions Added to V850E1 CPU and V850 CPU Instructions with Same Instruction Code (2/2)**

| Instructions Added in V850E1 CPU | V850 CPU Instructions with Same Instruction Code as V850E1 CPU |
|---|---|
| CLR1 reg2, [reg1] | Undefined |
| DBRET[Note] | |
| DBTRAP[Note] | |
| LD.BU disp16 [reg1], reg2 | |
| LD.HU disp16 [reg1], reg2 | |
| NOT1 reg2, [reg1] | |
| PREPARE list12, imm5 | |
| PREPARE list12, imm5, sp/imm | |
| SET1 reg2, [reg1] | |
| SLD.BU disp4 [ep], reg2 | |
| SLD.HU disp5 [ep], reg2 | |
| TST1 reg2, [reg1] | |

★ **Note** Not supported in type C products

# APPENDIX F INDEX


## [Numeral]

16-bit format instruction ..... 211
16-bit load/store instruction format ..... 44
2-clock branch ..... 169
3-operand instruction format ..... 45
32-bit format instruction ..... 211
32-bit load/store instruction format ..... 45

## [A]

ADD ..... 53
ADDI ..... 54
Additional items related to pipeline ..... 186
Address space ..... 37
Addressing mode ..... 39
Alignment hazard ..... 182
AND ..... 55
ANDI ..... 56
Arithmetic operation instructions ..... 48
Arithmetic operation instructions (pipeline) ..... 173
ASID ..... 30

## [B]

Based addressing ..... 41
Bcond ..... 57
Bit ..... 34, 35
Bit addressing ..... 42
Bit manipulation instruction format ..... 45
Bit manipulation instructions ..... 49
Bit manipulation instructions (pipeline) ..... 176
BPAM0 ..... 31
BPAM1 ..... 31
BPAV0 ..... 31
BPAV1 ..... 31
BPC0 ..... 29
BPC1 ..... 29
BPDM0 ..... 32
BPDM1 ..... 32
BPDV0 ..... 32
BPDV1 ..... 32
BR instruction (pipeline) ..... 175
Branch instructions ..... 49
Branch instructions (pipeline) ..... 174
Breakpoint address mask registers 0 and 1 ..... 31
Breakpoint address setting registers 0 and 1 ..... 31
Breakpoint control registers 0 and 1 ..... 29
Breakpoint data mask registers 0 and 1 ..... 32
Breakpoint data setting registers 0 and 1 ..... 32
BSH ..... 59
BSW ..... 60
Byte ..... 34

## [C]

CALLT ..... 61
CALLT base pointer ..... 25
CALLT caller status saving registers ..... 23
CALLT instruction (pipeline) ..... 176
Cautions when creating programs ..... 185
CLR1 ..... 62
CLR1 instruction (pipeline) ..... 176
CMOV ..... 63
CMP ..... 64
Conditional branch instruction format ..... 44
CTBP ..... 25
CTPC ..... 23
CTPSW ..... 23
CTRET ..... 65
CTRET instruction (pipeline) ..... 177

## [D]

Data alignment ..... 36
Data format ..... 33
Data representation ..... 35
Data type ..... 33
DBPC ..... 24
DBPSW ..... 24
DBRET ..... 66
DBRET instruction (pipeline) ..... 181
DBTRAP ..... 67
DBTRAP instruction (pipeline) ..... 181
Debug function instructions ..... 50
Debug function instructions (pipeline) ..... 181
Debug interface register ..... 26
Debug trap ..... 161
DI ..... 68
DI instruction (pipeline) ..... 177
DIR ..... 26
DISPOSE ..... 69
DISPOSE instruction (pipeline) ..... 178
DIV ..... 71
DIVH ..... 72
DIVHU ..... 74
Divide instructions (pipeline) ..... 173
DIVU ..... 75

## [E]

ECR ..... 20
Efficient pipeline processing ..... 170
EI ..... 76

EIPC ..... 19
EIPSW ..... 19
EI instruction (pipeline) ..... 177
Exception cause register ..... 20
Exception/debug trap status saving registers ..... 24
Exception processing ..... 159
Exception trap ..... 160
Extended instruction format 1 ..... 45
Extended instruction format 2 ..... 46
Extended instruction format 3 ..... 46
Extended instruction format 4 ..... 46

**[F]**

FEPC ..... 20
FEPSW ..... 20
Format I ..... 43
Format II ..... 43
Format III ..... 44
Format IV ..... 44
Format V ..... 44
Format VI ..... 45
Format VII ..... 45
Format VIII ..... 45
Format IX ..... 45
Format X ..... 46
Format XI ..... 46
Format XII ..... 46
Format XIII ..... 46

**[G]**

General-purpose registers ..... 16

**[H]**

Halfword ..... 34
HALT ..... 77
HALT instruction (pipeline) ..... 178
Harvard architecture ..... 186
How to shift to debug mode ..... 189
HSW ..... 78

**[I]**

imm-reg instruction format ..... 43
Immediate addressing ..... 41
Instruction address ..... 39
Instruction format ..... 43
Instruction opcode map ..... 211
Instruction set ..... 51
Integer ..... 35
Internal configuration ..... 15
Interrupt servicing ..... 156
Interrupt status saving registers ..... 19

**[J]**

JARL ..... 79
JMP ..... 80
JMP instruction (pipeline) ..... 175
JR ..... 81
Jump instruction format ..... 44

**[L]**

LD instructions ..... 47
LD instructions (pipeline) ..... 171
LD.B ..... 82
LD.BU ..... 83
LD.H ..... 84
LD.HU ..... 86
LD.W ..... 88
LDSR ..... 90
LDSR instruction (pipeline) ..... 178
Load instructions ..... 47
Load instructions (pipeline) ..... 171
Logical operation instructions ..... 48
Logical operation instructions (pipeline) ..... 174

**[M]**

Maskable interrupt ..... 156
Memory map ..... 38
MOV ..... 91
MOVEA ..... 92
Move word instruction (pipeline) ..... 173
MOVHI ..... 93
MUL ..... 94
MULH ..... 96
MULHI ..... 97
Multiply instructions ..... 47
Multiply instructions (pipeline) ..... 172
MULU ..... 98

**[N]**

NMI status saving registers ..... 20
Non-blocking load/store ..... 168
Non-maskable interrupt ..... 158
NOP ..... 100
NOP instruction (pipeline) ..... 179
NOT ..... 101
NOT1 ..... 102
NOT1 instruction (pipeline) ..... 176

**[O]**

Operand address ..... 41
OR ..... 103
ORI ..... 104

## [P]

PC ..... 17
Pipeline ..... 166
Pipeline disorder ..... 182
Pipeline flow during execution of instructions ..... 171
PREPARE ..... 105
PREPARE instruction (pipeline) ..... 179
Program counter ..... 17
Program ID register ..... 30
Program registers ..... 16
Program status word ..... 21
PSW ..... 21

## [R]

r0 to r31 ..... 16
reg-reg instruction format ..... 43
Register addressing ..... 41
Register addressing (register indirect) ..... 40
Register set ..... 15
Register status after reset ..... 164
Relative addressing (PC relative) ..... 39
Reset ..... 164
Restoring from exception trap and debug trap ..... 163
Restoring from interrupt/exception processing ..... 162
RETI ..... 107
RETI instruction (pipeline) ..... 179

## [S]

SAR ..... 109
SASF ..... 110
SATADD ..... 111
SATSUB ..... 112
SATSUBI ..... 113
SATSUBR ..... 114
Saturated operation instructions ..... 48
Saturated operation instructions (pipeline) ..... 174
SET1 ..... 115
SET1 instruction (pipeline) ..... 176
SETF ..... 116
Shifting to debug mode ..... 189
SHL ..... 118
Short path ..... 187
SHR ..... 119
SLD.B ..... 120
SLD.BU ..... 121
SLD.H ..... 122
SLD.HU ..... 124
SLD.W ..... 126
SLD instructions ..... 47
SLD instructions (pipeline) ..... 171
Software exception ..... 159
Special instructions ..... 49
Special instructions (pipeline) ..... 176
SST.B ..... 128
SST.H ..... 129
SST.W ..... 131
SST instructions ..... 47
ST.B ..... 133
ST.H ..... 134
ST.W ..... 136
ST instructions ..... 47
Stack manipulation instruction format 1 ..... 46
Starting up ..... 165
Store instructions ..... 47
Store instructions (pipeline) ..... 172
STSR ..... 138
STSR instruction (pipeline) ..... 178
SUB ..... 139
SUBR ..... 140
SWITCH ..... 141
SWITCH instruction (pipeline) ..... 180
SXB ..... 142
SXH ..... 143
System registers ..... 18

## [T]

TRAP ..... 144
TRAP instruction (pipeline) ..... 180
TST ..... 145
TST1 ..... 146
TST1 instruction (pipeline) ..... 176

## [U]

Unconditional branch instructions ..... 175
Unsigned integer ..... 35

## [W]

Word ..... 33

## [X]

XOR ..... 147
XORI ..... 148

## [Z]

ZXB ..... 149
ZXH ..... 150

★ 

# APPENDIX G REVISION HISTORY

## G.1 Major Revisions in This Edition

| Page | Description |
|---|---|
| Throughout | Deletion of product names from target devices, addition of product types as target devices |
| p.16 | Modification of description in **2.1 (1) General-purpose registers (r0 to r31)** |
| p.18 | Modification of **Table 2-2 System Register Numbers** |
| p.24 | Modification and addition of description in **2.2.6 Exception/debug trap status saving registers (DBPC, DBPSW)** |
| p.24 | Addition of **Table 2-3 Contents Saved to DBPC** |
| p.26 | Modification of **Figure 2-10 Debug Interface Register (DIR)** |
| p.29 | Modification of **Figure 2-11 Breakpoint Control Registers 0 and 1 (BPC0, BPC1)** |
| p.30 | Addition of description to **2.2.10 Program ID register (ASID)** |
| p.31 | Addition of description to **2.2.11 Breakpoint address setting registers 0 and 1 (BPAV0, BPAV1)** |
| p.31 | Addition of description to **2.2.12 Breakpoint address mask registers 0 and 1 (BPAM0, BPAM1)** |
| p.32 | Addition of description to **2.2.13 Breakpoint data setting registers 0 and 1 (BPDV0, BPDV1)** |
| p.32 | Addition of description to **2.2.14 Breakpoint data mask registers 0 and 1 (BPDM0, BPDM1)** |
| p.36 | Modification of **3.3 Data Alignment** |
| pp.94, 98 | Modification of description and addition of **Caution** to MUL and MULU in **5.3 Instruction Set** |
| p.120 | Addition of **Caution** (2) to **5.3 Instruction Set SLD.B** |
| p.121 | Addition of **Caution** (2) to **5.3 Instruction Set SLD.BU** |
| p.123 | Addition of **Caution** (2) to **5.3 Instruction Set SLD.H** |
| p.125 | Addition of **Caution** (2) to **5.3 Instruction Set SLD.HU** |
| p.127 | Addition of **Caution** (2) to **5.3 Instruction Set SLD.W** |
| p.144 | Correction of operation of TRAP in **5.3 Instruction Set** |
| pp.153, 154 | Modification and addition of **Notes** in **Table 5-6 List of Number of Instruction Execution Clock Cycles** |
| p.160 | Addition of (4) to **6.2.2 Exception trap** |
| p.161 | Addition of description to **6.2.3 Debug trap** |
| p.189 | Addition of **CHAPTER 9 SHIFTING TO DEBUG MODE** |
| p.197 | Addition of **APPENDIX A NOTES** |
| p.224 | Addition of **APPENDIX G REVISION HISTORY** |

## G.2 History of Revisions up to This Edition

A history of the revisions up to this edition is shown below. “Applied to:” indicates the chapters to which the revision was applied.

(1/2)

| Edition | Major Revision from Previous Edition | Applied to: |
|---|---|---|
| 2nd | • Addition of following products (under development) to target products<br>NB85ET, NU85E, NU85ET, μPD703108, 703114, 70F3114, 703116<br>• Deletion of following product from target products<br>μPD703117<br>• Change of following products from “under development” to “developed”<br>μPD703106, 703107, 70F3107 | Throughout |
| | Change of **Note** in **Figure 2-1 Registers** | **CHAPTER 2 REGISTER SET** |
| | Change of **Table 2-2 System Register Numbers** | |
| | Addition of **Note** to **Figure 2-6 Program Status Word (PSW)** | |
| | Addition of **Note** to **2.2.6 Exception/debug trap status saving registers (DBPC, DBPSW)** | |
| | Change of **Caution** in **2.2.8 Debug interface register (DIR)** | |
| | Change of **Caution** in **2.2.9 Breakpoint control registers 0 and 1 (BPC0, BPC1)** | |
| | Change of **Figure 2-11 Breakpoint Control Registers 0 and 1 (BPC0, BPC1)** | |
| | Change of **Caution** in **2.2.10 Program ID register (ASID)** | |
| | Change of **Caution** in **2.2.11 Breakpoint address setting registers 0 and 1 (BPAV0, BPAV1)** | |
| | Change of **Caution** in **2.2.12 Breakpoint address mask registers 0 and 1 (BPAM0, BPAM1)** | |
| | Change of **Caution** in **2.2.13 Breakpoint data setting registers 0 and 1 (BPDV0, BPDV1)** | |
| | Change of **Caution** in **2.2.14 Breakpoint data mask registers 0 and 1 (BPDM0, BPDM1)** | |
| | Addition of **Caution** to **5.2 (10) Debug function instructions** | **CHAPTER 5 INSTRUCTION** |
| | Addition of **Caution** to **DBRET** in **5.3 Instruction Set** | |
| | Addition of **Caution** to **DBTRAP** in **5.3 Instruction Set** | |
| | Change and addition of **Note** in **Table 5-6 List of Number of Instruction Execution Clock Cycles (NB85E, NB85ET, NU85E, and NU85ET)** | |
| | Change of **Note** in **Table 5-7 List of Number of Instruction Execution Clock Cycles (V850E/MA1, V850E/MA2, V850E/IA1, and V850E/IA2)** | |
| | Addition of **Note** to **Table 6-1 Interrupt/Exception Codes** | **CHAPTER 6 INTERRUPT AND EXCEPTION** |
| | Addition of **Caution** to **6.2.3 Debug trap** | |
| | Addition of **Remark** and **Example** to **8.1.2 2-clock branch** | **CHAPTER 8 PIPELINE** |
| | Addition of **Caution** to **8.1.3 Efficient pipeline processing** | |
| | Correction of description in **8.2 (2) V850E/MA1, V850E/MA2, V850E/IA1, V850E/IA2** | |
| | Correction of description in **8.2.1 (2) SLD instructions** | |
| | Correction of description in **8.2.3 Multiply instructions** | |
| | Addition of **Remark** to **8.2.4 (3) Divide instructions** | |
| | Correction of description in **8.2.8 (2) TST1 instruction** | |
| | Addition of **Remark** to **8.2.9 (3) DI, EI instructions** | |
| | Addition of **Caution** to **8.2.9 (7) NOP instruction** | |

(2/2)

| Edition | Major Revision from Previous Edition | Applied to: |
|---|---|---|
| 2nd | Addition of **8.3 Pipeline Disorder** | **CHAPTER 8 PIPELINE** |
| | Addition of **8.4 Additional Items Related to Pipeline** | |
| | Addition of **Note** to **Table A-1 Instruction Function List (in Alphabetical Order)** | **APPENDIX A INSTRUCTION LIST** |
| | Addition of **Note** to **Table A-2 Instruction List (in Format Order)** | |
| | Correction of **Figure** in **Appendix B (2) 32-bit format instruction** | **APPENDIX B INSTRUCTION OPCODE MAP** |
| | Addition of **Remark** to **Appendix B [a] Opcode** | |
| | Addition of **Remark** to **Appendix B [e] Expansion 1 (sub-opcode)** | |
| | Addition of **Note** to **Appendix C DIFFERENCES WITH ARCHITECTURE OF V850 CPU** | **APPENDIX C DIFFERENCES WITH ARCHITECTURE OF V850 CPU** |
| | Addition of **Note** to **Table D-1 Instructions Added to V850E1 CPU and V850 CPU Instructions with Same Instruction Code** | **APPENDIX D INSTRUCTIONS ADDED FOR V850E1 CPU COMPARED WITH V850 CPU** |