Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of non-flammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.
<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Mar 01,2004</td>
<td>First edition issued</td>
<td>-</td>
</tr>
</tbody>
</table>
This page is blank for reasons of layout.
# Table of contents

## CHAPTER 1 CPU PROGRAMMING MODEL

1.1 Processor Modes ........................................................................................................... 1-2  
   1.1.1 Privileged Instructions .......................................................................................... 1-2  
1.2 CPU Registers ............................................................................................................... 1-2  
1.3 General-purpose Registers ......................................................................................... 1-2  
1.4 Control Registers ...................................................................................................... 1-3  
   1.4.1 Processor Status Word Register: PSW (CR0) .................................................... 1-4  
   1.4.2 Condition Bit Register: CBR (CR1) ...................................................................... 1-5  
   1.4.3 Stack Pointer for Interrupt: SPI (CR2) and Stack Pointer for User: SPU (CR3) . 1-5  
   1.4.4 EIT Vector Base Register: EVB (CR5) ............................................................... 1-6  
   1.4.5 Backup PC: BPC (CR6) ..................................................................................... 1-6  
1.5 Accumulators ............................................................................................................. 1-7  
1.6 Program Counter (PC) ............................................................................................... 1-7  
1.7 Data Formats ............................................................................................................. 1-8  
   1.7.1 Bi-endian Function ............................................................................................ 1-8  
   1.7.2 Data Types ...................................................................................................... 1-8  
   1.7.3 Data Formats .................................................................................................. 1-9  
1.8 Addressing Modes ................................................................................................... 1-11

## CHAPTER 2 INSTRUCTION SET

2.1 Outline of the Instruction Set ....................................................................................... 2-2  
2.2 Instruction Set .......................................................................................................... 2-2  
   2.2.1 Load and Store Instructions (10 instructions) ................................................... 2-2  
   2.2.2 Transfer Instructions (6 instructions) ................................................................. 2-4  
   2.2.3 Arithmetic/Logical Instructions (46 instructions) ............................................. 2-4  
   2.2.4 Branch Instructions (21 instructions) ............................................................... 2-6  
   2.2.5 Bit Manipulating Instructions (5 instructions) ................................................. 2-6  
   2.2.6 EIT Related Instructions (2 instructions) ......................................................... 2-8  
   2.2.7 DSP Function Instructions (22 instructions) ................................................. 2-9  
   2.2.8 Coprocessor Support Instructions (3 instructions) ......................................... 2-9  
2.3 List of OPSP Extended Instruction Set ...................................................................... 2-15  
   2.3.1 New Extended Instructions of the OPSP-CPU .............................................. 2-15  
   2.3.2 Function-Extended Instructions of the OPSP-CPU ......................................... 2-16  
2.4 Instruction Formats .................................................................................................. 2-17  
2.5 Parallel Instruction Execution .................................................................................... 2-18  
   2.5.1 Instruction Formats .......................................................................................... 2-18  
   2.5.2 Parallel Instruction Execution in the OPSP .................................................... 2-19  
   2.5.3 16-Bit Instruction List by Category ................................................................. 2-19  
   2.5.4 Positions of Parallel Executed Instructions ............................................... 2-21  
   2.5.5 Operand Interferences ..................................................................................... 2-22
CHAPTER 3 INSTRUCTIONS

3.1 Guide to Detailed Instruction Description .................................................................................. 3-2
3.2 Detailed Description of Instructions .......................................................................................... 3-6
3.3 Notes about the BCL and BNCL Instructions ............................................................................. 3-127
3.4 Exception and Trap Handling during Parallel Instruction Execution ........................................ 3-128

APPENDICES

Appendix 1 Mechanism of Pipelined Instruction Processing .............................................................. A-2
  Appendix 1.1 Outline of Pipelined Instruction Processing ................................................................. A-2
  Appendix 1.2 Flow of Instruction Processing in the O and S Pipes .................................................. A-5
  Appendix 1.3 Instructions and Pipelined Processing ...................................................................... A-6
  Appendix 1.4 Pipelined Processing of Parallel Instructions ............................................................ A-7
  Appendix 1.5 Basic Pipeline Operation ......................................................................................... A-8
Appendix 2 Instruction Processing Time ............................................................................................ A-12
1.1 Processor Modes

The OPSP-CPU core (hereafter abbreviated “OPSP-CPU”) provides two processor modes: Supervisor Mode and User Mode. A hierarchical resource protection mechanism can be realized by using these processor modes. Each processor mode has designated rights with respect to memory access and executable instructions, which are higher for supervisor mode than for user mode.

When an EIT event occurs, the CPU goes to supervisor mode. The processor mode in which the CPU was immediately before the EIT event occurred is stored in the backup PM (BPM) bit of the Processor Status Word Register (PSW). When the RTE instruction is executed, the CPU returns to the previous processor mode that is stored in the BPM bit.

1.1.1 Privileged Instructions

Privileged instructions are those that can only be executed in supervisor mode. If a privileged instruction is executed in user mode, a privileged instruction exception occurs. The privileged instructions include RTE, MVTC, SETPSW, and CLRPSW.

1.2 CPU Registers

The OPSP-CPU has 16 general-purpose registers, 6 control registers, 2 accumulators, and a program counter. The accumulators are configured with 64 bits, while all other registers are configured with 32 bits.

1.3 General-purpose Registers

The general-purpose registers are 32 bits wide, and there are 16 of them (R0 to R15). These registers are used to hold data and base addresses. Of these, R14 and R15 are used as a link register and a stack pointer (SPI or SPU), respectively. The link register is used to hold the return address when executing a subroutine call instruction. The stack pointer is switched between a stack pointer for interrupt (SPI) and a stack pointer for user (SPU) depending on the value of the stack mode (SM) bit in the Processor Status Word Register (PSW).

Note: The stack pointer is switched between a stack pointer for interrupt (SPI) and a stack pointer for user (SPU) depending on the value of the SM bit in the PSW.

Figure 1.3.1 General-purpose Registers
1.4 Control Registers

There are six control registers: Processor Status Word Register (PSW), Condition Bit Register (C), Stack Pointer for Interrupt (SPI), Stack Pointer for User (SPU), EIT Vector Base Register (EVB), and Backup PC (BPC).

Dedicated MVTC and MVFC instructions are used to set and read these control registers. Furthermore, SETPSW and CLRPSW instructions can be used for the PSW.

MVTC, SETPSW, and CLRPSW are the privileged instructions that can only be executed when the CPU is operating in supervisor mode. Which processor mode is active is determined by the processor mode (PM) bit in the Processor Status Word Register (PSW).

Note 1: CRn (n = 0–3, 5, 6) denotes a control register number.

Note 2: Dedicated MVTC and MVFC instructions are used to set and read the control registers.

Figure 1.4.1 Control Registers
1.4.1 Processor Status Word Register: PSW (CR0)

The Processor Status Word Register (PSW) indicates the status of the OPSP-CPU. It consists of two bit fields: the PSW field that is normally used, and the BPSW field in which the PSW field is saved when an EIT occurs.

The PSW field further consists of the stack mode bit (SM), interrupt enable bit (IE), processor mode bit (PM), coprocessor interrupt enable bit (CE), and condition bit (C). Similarly, the BPSW field consists of the backup SM bit (BSM), backup IE bit (BIE), backup PM bit (BPM), backup CE bit (BCE), and backup C bit (BC).

After reset, the BSM, BIE, BPM, BCE, and BC are indeterminate. All other bits are 0.

To switch the processor mode, set BPM = 1 using the MVTC instruction and then execute the RTE instruction to branch to the user space (H'0000 0000 – H'7FFF FFFF). If the PM bit needs to be altered directly with the MVTC instruction, always be sure to alter it in the user space.

---

<table>
<thead>
<tr>
<th>b</th>
<th>Bit Name</th>
<th>Function</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–15</td>
<td>No functions assigned. Fix these bits to 0.</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>BSM</td>
<td>Backup SM bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>17</td>
<td>BIE</td>
<td>Backup IE bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>18–19</td>
<td>No functions assigned. Fix these bits to 0.</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>BPM</td>
<td>Backup PM bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>21</td>
<td>BCE</td>
<td>Backup CE bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>22</td>
<td>No functions assigned. Fix these bits to 0.</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>BC</td>
<td>Backup C bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>SM</td>
<td>Stack mode bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>25</td>
<td>IE</td>
<td>Interrupt enable bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>26–27</td>
<td>No functions assigned. Fix these bits to 0.</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>PM</td>
<td>Processor mode bit</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>29</td>
<td>CE</td>
<td>Coprocessor interrupt enable bit</td>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

After reset: \( \text{B'0000 0000 0000 0000 ??00 ??0? 0000 0000} \)
1.4.2 Condition Bit Register: CBR (CR1)

The Condition Bit Register (CBR) is derived from the condition bit (C) of the PSW to serve as a separate register. The value written to the condition bit in the PSW is reflected in this register. This register can only be read. (Writing to this register with the MVTC instruction is ignored.)

After reset, the CBR is H'0000 0000.

1.4.3 Stack Pointer for Interrupt: SPI (CR2) and Stack Pointer for User: SPU (CR3)

The Stack Pointer for Interrupt (SPI) and the Stack Pointer for User (SPU) hold the address of the current stack pointer. These registers can be accessed as the general-purpose register R15. Whether R15 is used as the SPI or as the SPU is determined by the stack mode bit (SM) in the PSW.
1.4.4 EIT Vector Base Register: EVB (CR5)

The EIT Vector Base Register (EVB) holds the EIT vector entry start address. The 16 high-order bits of the EIT vector entry start address comprise the value of the 16 high-order bits in this register.

![EVB Register Diagram](image)

<table>
<thead>
<tr>
<th>b</th>
<th>Bit Name</th>
<th>Function</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AT</td>
<td>Address translation mode</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Address translation mode bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1–15</td>
<td>EVB</td>
<td>Set A1–A15 of EIT vector entry in these bits.</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>Vector base bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16–31</td>
<td>No functions assigned. Fix these bits to 0.</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(1) AT (address translation mode) bit (b0)

This bit is a copy of the address translation mode bit (AT) in the MATM register, and is a read-only bit.

(2) EVB (EIT vector base) bits (b1–b15)

These bits set the EIT vector entry start address A1–A15. However, the reset interrupt (RI) vector is located at the address H'0000 0000 no matter how the EIT vector base bits are set.

Note: The EVB register can be set only once immediately after reset. Write to the EVB register should be performed at the beginning of a reset handler.

1.4.5 Backup PC: BPC (CR6)

The Backup PC (BPC) is used to save the value of the program counter (PC) when an EIT occurs. Bit 31 is fixed to 0.

When an EIT occurs, the PC value at which the EIT occurred or the PC value for the next instruction is set in the BPC depending on the type of the EIT that occurred. The value of the BPC is returned to the PC when the RTE instruction is executed.

![BPC Register Diagram](image)
1.5 Accumulators

The accumulator is a 56-bit register used in the instructions for the DSP function. There are two of such accumulators, ACC0 and ACC1. During read or write, the accumulator is handled as a 64-bit register. In this case, bits 0–7 in the accumulator are sign-extended with the value of bit 8 during read, and are ignored during write. The accumulator is also used in the multiplication instruction “MUL.” Be aware that when this instruction is executed, the value of the accumulator, whether ACC0 or ACC1, is destroyed.

Use the MVTACHI and MVTACLO instructions to write to the accumulator. The MVTACHI and MVTACLO instructions write data to the 32 high-order bits (bits 0–31) and the 32 low-order bits (bits 32–63) in the accumulator, respectively.

Use the MVFACHI, MVFACLO, and MVFACMI instructions to read the accumulator. The MVFACHI, MVFACLO, and MVFACMI instructions read data from the 32 high-order bits (bits 0–31), the 32 low-order bits (bits 32–63), and the 32 middle bits (bits 16–47) in the accumulator, respectively.

After reset, ACC0 and ACC1 are indeterminate.

1.6 Program Counter (PC)

The Program Counter (PC) is a 32-bit counter that holds the address of the currently executed instruction. Since the instructions in the OPSP-CPU begin from even addresses, the LSB (bit 31) in the PC is always 0.

After reset, the PC is H’0000 0000.
1.7 Data Formats

1.7.1 Bi-endian Function

The OPSP-CPU supports the bi-endian function that allows either data format, big endian or little endian, to be adopted.

This manual is written for operation in big endian mode.

1.7.2 Data Types

The data types that the instruction set of the OPSP-CPU can handle are signed or unsigned 8, 16, and 32-bit integers. Signed integer values are represented by the 2’s complement.

![Data Types Diagram]

Figure 1.7.1 Data Types
1.7.3 Data Formats

(1) Data formats in the OPSP-CPU registers

The data size in the OPSP-CPU registers is always the word (32 bits). When byte (8-bit) or halfword (16-bit) data in memory is loaded into a register, the data is sign-extended (LDB, LDH instructions) or zero-extended (LDUB, LDUH instructions) to the word (32-bit) quantity before being stored in the register.

When data in an OPSP-CPU register is stored into memory, the ST, STH, or STB instruction is used. The ST, STH, and STB instructions store the full 32-bit data, the lower 16-bit data, or the least significant 8-bit data of the register in memory, respectively.

![Diagram of data formats in registers](image-url)
(2) **Data formats in memory**

The data in memory has one of three data sizes: byte (8 bits), halfword (16 bits), or word (32 bits). Although byte data can be located at any address, halfword and word data must be located at halfword-aligned addresses (least significant address bit = 0) and word-aligned addresses (two least significant address bits = 00), respectively. If access to misaligned memory data is attempted, an address exception occurs.

![Figure 1.7.3 Data Formats in Memory](image-url)
1.8 Addressing Modes

The OPSP-CPU has the following addressing modes:

(1) Register direct [expressed as R or CR or A\textsuperscript{Note}]
A general-purpose or control register or an accumulator is specified directly as the target to be operated on.

(2) Register indirect [expressed as @R]
The address is indicated indirectly by a register value. (This addressing mode can be specified in all load and store instructions.)

(3) Register relative indirect (expressed as @(disp,R])
The address is indicated indirectly by (register value) + (16-bit displacement which is sign-extended to 32 bits).

(4) Register indirect + register update
• Register value incremented by 1
  The address is indicated by a preupdate register value (specifiable in only STB instruction)
• Register value incremented by 2
  The address is indicated by a preupdate register value (specifiable in only STH instruction)
• Register value incremented by 4
  The address is indicated by a preupdate register value (specifiable in only LD instruction)
• Register value incremented by 8
  The address is indicated by an updated register value (specifiable in only ST instruction)
• Register value decremented by 4
  The address is indicated by an updated register value (specifiable in only ST instruction)

(5) Immediate [expressed as #imm]
1, 4, 5, 8, 16, or 24-bit immediate value. (For details on how the value is handled, refer to the detailed description of each instruction in the latter part of this manual.)

(6) PC relative [expressed as pcdisp]
The address is indicated by (PC value) + (8, 16, or 24-bit displacement which is sign-extended to 32 bits and then shifted left 2 bits).

Note: The accumulators ACC0 and ACC1 are mnemonically expressed as A0 and A1, respectively.
This page is blank for reasons of layout.
CHAPTER 2
INSTRUCTION SET
2.1 Outline of the Instruction Set

The OPSP-CPU has 115 distinct instructions. A RISC architecture is adopted for the OPSP-CPU, so that memory access basically is accomplished by using load and store instructions. Arithmetic/logical operations are executed by register-to-register operation. Furthermore, compound instructions such as Load & Address Update and Store & Address Update are supported.

2.2 Instruction Set

The instruction set of the OPSP-CPU is shown below.

New instructions that have been added in the OPSP-CPU from the M32R family instruction set are marked by double asterisks (**), and function-extended instructions are marked by a single asterisk (*).

2.2.1 Load and Store Instructions (10 instructions)

These instructions perform data transfer between memory and a register.

- LD Load
- LDB Load byte
- LDUB Load unsigned byte
- LDH Load halfword
- LDUH Load unsigned halfword
- LOCK Load locked
- ST Store
- * STB Store byte
- * STH Store halfword
- UNLOCK Store unlocked
Following three addressing modes can be specified in the load and store instructions.

(1) Register indirect
   The address is indicated indirectly by a register value. (This addressing mode can be specified in all load and store instructions.)

(2) Register relative indirect
   The address is indicated indirectly by (register value) + (16-bit displacement which is sign-extended to 32 bits). (This addressing mode can be specified in all load and store instructions other than LOCK and UNLOCK.)

(3) Register indirect + register update
   - Register value incremented by 1
     The address is indicated by a preupdate register value (specifiable in only STB instruction)
   - Register value incremented by 2
     The address is indicated by a preupdate register value (specifiable in only STH instruction)
   - Register value incremented by 4
     The address is indicated by a preupdate register value (specifiable in only LD instruction)
   - Register value incremented by 4
     The address is indicated by an updated register value (specifiable in only ST instruction)
   - Register value decremented by 4
     The address is indicated by an updated register value (specifiable in only ST instruction)

Whichever addressing mode is used, rules for the data formats in memory must be observed. To access halfword or word data, a halfword aligned or word aligned address must be specified, respectively. (The two least significant bits of the accessed address must be "00" or "10" for halfword data, or "00" for word data.) If a misaligned address is specified, an address exception occurs.

If byte or halfword data is accessed in a load instruction, the data has its high order bits sign or zero-extended to become 32-bit data before being stored in a register.
2.2.2 Transfer Instructions (6 instructions)

These instructions perform a register to register transfer or a register to immediate transfer.

- **LD24** Load 24-bit immediate
- **LDI** Load immediate
- **MV** Move register
- **MVFC** Move from control register
- **MVTC** Move to control register
- **SETH** Set high-order 16-bit

2.2.3 Arithmetic/Logical Instructions (46 instructions)

These instructions perform register to register comparison, arithmetic/logical operation, multiplication/division, or shift operation.

- **Comparison** (7 instructions)
  - **CMP** Compare
  - **CMPEQ** Compare equal to
  - **CMPI** Compare immediate
  - **CMPU** Compare unsigned
  - **CMPIU** Compare unsigned immediate
  - **CMPZ** Compare equal to zero
  - **PCMPBZ** Parallel compare byte to zero

- **Arithmetic operation** (10 instructions)
  - **ADD** Add
  - **ADD3** Add 3-operand
  - **ADDI** Add immediate
  - **ADDV** Add with overflow
  - **ADDV3** Add 3-operand with overflow
  - **ADDX** Add with carry
  - **NEG** Negate
  - **SUB** Subtract
  - **SUBV** Subtract with overflow
  - **SUBX** Subtract with borrow

- **Logical operation** (7 instructions)
  - **AND** AND
  - **AND3** AND 3-operand
  - **NOT** Logical NOT
  - **OR** OR
  - **OR3** OR 3-operand
  - **XOR** Exclusive OR
  - **XOR3** Exclusive OR 3-operand
### Multiplication/division (13 instructions)

- **DIV** Divide
- **DIVB** Divide byte
- **DIVH** Divide halfword
- **DIVU** Divide unsigned
- **DIVUB** Divide unsigned byte
- **DIVUH** Divide unsigned halfword
- **MUL** Multiply
- **REM** Reminder
- **REMB** Reminder byte
- **REMH** Reminder halfword
- **REMU** Reminder unsigned
- **REMUH** Reminder unsigned halfword

### Shift (9 instructions)

- **SLL** Shift left logical
- **SLL3** Shift left logical 3-operand
- **SLLI** Shift left logical immediate
- **SRA** Shift right arithmetic
- **SRA3** Shift right arithmetic 3-operand
- **SRAI** Shift right arithmetic immediate
- **SRL** Shift right logical
- **SRL3** Shift right logical 3-operand
- **SRLI** Shift right logical immediate
### 2.2.4 Branch Instructions (21 instructions)

These instructions are used to change the program flow.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>Branch on C-bit</td>
</tr>
<tr>
<td>** BCL</td>
<td>Branch and link on C-bit</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch on equal to</td>
</tr>
<tr>
<td>BEQZ</td>
<td>Branch on equal to zero</td>
</tr>
<tr>
<td>BGEZ</td>
<td>Branch on greater than or equal to zero</td>
</tr>
<tr>
<td>BGTZ</td>
<td>Branch on greater than zero</td>
</tr>
<tr>
<td>BL</td>
<td>Branch and link</td>
</tr>
<tr>
<td>BLEZ</td>
<td>Branch on less than or equal to zero</td>
</tr>
<tr>
<td>BLTZ</td>
<td>Branch on less than zero</td>
</tr>
<tr>
<td>BNC</td>
<td>Branch on not C-bit</td>
</tr>
<tr>
<td>** BNCL</td>
<td>Branch and link on not C-bit</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on not equal to</td>
</tr>
<tr>
<td>BNEZ</td>
<td>Branch on not equal to zero</td>
</tr>
<tr>
<td>BRA</td>
<td>Branch</td>
</tr>
<tr>
<td>** JC</td>
<td>Jump on C-bit</td>
</tr>
<tr>
<td>JL</td>
<td>Jump and link</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>** JNC</td>
<td>Jump on not C-bit</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>** SC</td>
<td>Skip on C-bit</td>
</tr>
<tr>
<td>** SNC</td>
<td>Skip on not C-bit</td>
</tr>
</tbody>
</table>

Only the word aligned addresses (those aligned with word boundaries) can be specified as the jump address.
For the BRA, BL, BC, BNC, BCL and BNCL instructions, an 8-bit or 24-bit immediate value can be specified in addressing mode. For the BEQ, BNE, BEQZ, BNEZ, BGTZ, BLTZ, BGEZ and BLEZ instructions, a 16-bit immediate value should be specified in addressing mode.

For the JMP, JL, JC and JNC instructions, the jump address is specified by a register value. However, the two least significant address bits are ignored.

For the SC and SNC instructions, the jump address is indicated by (PC value of the branch instruction) + 4.

For other branch instructions, the jump address is indicated by (PC value of branch instruction) + (sign-extended immediate value that is shifted two bits left). However, the two least significant bits of the PC value are cleared to 0 when an addition is performed. In Figure 2.2.1, for example, assume that instruction A or instruction B is the branch instruction, and that the program is to jump to instruction G. Then the immediate value, in either case, is 4.

For the JL, BL, BCL and BNCL instructions that are used for subroutine calls, the PC value for the return address is stored in R14 at the same time the program branches off. The value stored in R14 is (PC value of branch instruction + 4), with the two least significant bits of the PC value cleared to 0. In Figure 2.2.1, for example, assume that instruction A or instruction B is the JL, BL, BCL, or BNCL instruction. Then the return address, in either case, is the “instruction C.”

![Figure 2.2.1 Jump Address of a Branch Instruction](image)
2.2.5 Bit Manipulating Instructions (5 instructions)

These instructions set or clear the bits in memory or registers and those in the Processor Status Word Register (PSW).

** BCLR Bit clear
** BSET Bit set
** BTST Bit test
** CLRPSW Clear PSW
** SETPSW Set PSW

2.2.6 EIT Related Instructions (2 instructions)

These instructions are provided for EIT events (Exception, Interrupt and Trap). These include an instruction to invoke a trap and an instruction to return from EIT handling.

* TRAP Trap
* RTE Return from EIT
2.2.7 DSP Function Instructions (22 instructions)

In the OPSP-CPU, the DPS function instructions of the M32R family instruction set have been extended as follows:
- There are two accumulators, compared to one in the past.
- Multiply-accumulate operations are enhanced.
- New general-purpose register rounding instructions are added.

The DPS function instructions of the OPSP-CPU are shown below. New instructions that have been added in the OPSP-CPU from the M32R family instruction set are marked by double asterisks (**), and function-extended instructions are marked by a single asterisk (*).

These instructions include those that perform 32 bit × 16 bit or 16 bit × 16 bit multiply or multiply-accumulate operations. Also included are those that round the data in an accumulator or general-purpose register or perform data transfer between an accumulator and general-purpose register.

* MACHI Multiply-accumulate high-order halfwords
** MACLH1 Multiply-accumulate low-order halfword and high-order halfword using accumulator1
* MACLO Multiply-accumulate low-order halfwords
MACWHI Multiply-accumulate word and high-order halfword
MACWLO Multiply-accumulate word and low-order halfword
** MACWU1 Multiply-accumulate word and unsigned low-order halfword using accumulator1
** MSBLO Multiply low-order halfwords and subtract
* MULH1 Multiply high-order halfwords
* MULLO Multiply low-order halfwords
MULWHI Multiply word and high-order halfword
MULWLO Multiply word and low-order halfword
** MULWU1 Multiply word and unsigned low-order halfword using accumulator1
* MVFACHI Move high-order word from accumulator
* MVFACLO Move low-order word from accumulator
* MVFACMI Move middle-order word from accumulator
* MVTACHI Move high-order word to accumulator
* MVTACLO Move low-order word to accumulator
* RAC Round accumulator
* RACH Round accumulator halfword
** SADD Add accumulators
** SATB Saturate word into byte
** SATH Saturate word into halfword

Operation of these instructions are schematically shown in the next pages.
Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, “Instructions”.

Figure 2.2.2 Operation of the DSP Function Instructions 1 (Multiplication)
Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, “Instructions”.

Figure 2.2.3 Operation of the DSP Function Instructions 2 (Multiply-Accumulate Operation)
Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, "Instructions."

Figure 2.2.4 Operation of the DSP Function Instructions 3 (Multiply-Accumulate Operation)

Note: In the actual operation of the DSP function instructions, the result is adjusted for the storage location or sign-extended. For details, refer to Chapter 3, "Instructions."

Figure 2.2.5 Operation of the DSP Function Instructions 4 (Addition)
<Rounding the accumulator data to a word size>

\[ \text{ACC0,ACC1} \]

\[
\begin{array}{ccc}
\text{Sign} & \text{Data} & 0 \\
\end{array}
\]

<Rounding the accumulator data to a halfword size>

\[ \text{ACC0,ACC1} \]

\[
\begin{array}{ccc}
\text{Sign} & \text{Data} & 0 \\
\end{array}
\]

<Rounding the data in a general-purpose register to a byte size>

\[
\begin{array}{ccc}
\text{Rsrc} & \text{Sign} & \text{Data} \\
\end{array}
\]

<Rounding the data in a general-purpose register to a halfword size>

\[
\begin{array}{ccc}
\text{Rsrc} & \text{Sign} & \text{Data} \\
\end{array}
\]

Figure 2.2.6 Operation of the DSP Function Instructions 5 (Rounding)

MVFACMI instruction

\[ \text{ACC0,ACC1} \]

\[ \begin{array}{ccc}
\text{MVFACHI} & \text{MVFACLO} \\
\end{array} \]

\[
\begin{array}{ccc}
\text{Rdest} & \text{Sign} & \text{Data} \\
\end{array}
\]

Figure 2.2.7 Operation of the DSP Function Instructions 6 (Transfer between Accumulator and Register)
2.2.8 Coprocessor Support Instructions (3 instructions)

These instructions are used for interfacing with a coprocessor, as shown below.

- MVTCP: Move to Coprocessor register
- MVFCP: Move from Coprocessor register
- OPECP: Operate Coprocessor
2.3 List of OPSP Extended Instruction Set

The instruction set of the OPSP-CPU has 27 new instructions that have been added as extensions from the M32R family instruction set and 21 conventional instructions which have had their functionality extended.

### 2.3.1 New Extended Instructions of the OPSP-CPU

Table 2.3.1 List of new extended instructions

<table>
<thead>
<tr>
<th>Classification</th>
<th>Mnemonic</th>
<th>Functional outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison instructions</td>
<td>CMPEQ</td>
<td>Compare between registers</td>
</tr>
<tr>
<td></td>
<td>CMPZ</td>
<td>Compare register and immediate 0</td>
</tr>
<tr>
<td></td>
<td>PCMPBZ</td>
<td>Compare register and immediate 0 bytewise</td>
</tr>
<tr>
<td>Multiplication/division</td>
<td>DIVB</td>
<td>Divide 8-bit signed integer</td>
</tr>
<tr>
<td></td>
<td>DIVH</td>
<td>Divide 16-bit signed integer</td>
</tr>
<tr>
<td></td>
<td>DIVUB</td>
<td>Divide 8-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>DIVUH</td>
<td>Divide 16-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>REMB</td>
<td>Remainder of 8-bit signed integer</td>
</tr>
<tr>
<td></td>
<td>REMH</td>
<td>Remainder of 16-bit signed integer</td>
</tr>
<tr>
<td></td>
<td>REMUB</td>
<td>Remainder of 8-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>REMUH</td>
<td>Remainder of 16-bit unsigned integer</td>
</tr>
<tr>
<td>Branch instructions</td>
<td>BCL</td>
<td>Branch if condition bit (C) = 1 and store return address in R14</td>
</tr>
<tr>
<td></td>
<td>BNCL</td>
<td>Branch if condition bit (C) = 0 and store return address in R14</td>
</tr>
<tr>
<td></td>
<td>JC</td>
<td>Branch if condition bit (C) = 1</td>
</tr>
<tr>
<td></td>
<td>JNC</td>
<td>Branch if condition bit (C) = 0</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>Skip parallel execution pair if condition bit (C) = 1</td>
</tr>
<tr>
<td></td>
<td>SNC</td>
<td>Skip parallel execution pair if condition bit (C) = 0</td>
</tr>
<tr>
<td>DSP function instructions</td>
<td>MACLH1</td>
<td>Multiply-accumulate operation (register × register + accumulator A1 → accumulator A1)</td>
</tr>
<tr>
<td></td>
<td>MACWU1</td>
<td>Multiply-accumulate operation (register × register + accumulator A1 → accumulator A1)</td>
</tr>
<tr>
<td></td>
<td>MSBLO</td>
<td>Multiply-accumulate operation (accumulator A0 – register × register → accumulator A0)</td>
</tr>
<tr>
<td></td>
<td>MULWU1</td>
<td>Multiplication (register × register → accumulator A1)</td>
</tr>
<tr>
<td></td>
<td>SADD</td>
<td>Addition (accumulator A0 + accumulator A1 → accumulator A0)</td>
</tr>
<tr>
<td></td>
<td>SATB</td>
<td>Round register data to byte size</td>
</tr>
<tr>
<td></td>
<td>SATH</td>
<td>Round register data to halfword size</td>
</tr>
<tr>
<td>Coprocessor support instructions</td>
<td>MVTCP</td>
<td>Move to coprocessor register</td>
</tr>
<tr>
<td></td>
<td>MVFCP</td>
<td>Move from coprocessor register</td>
</tr>
<tr>
<td></td>
<td>OPEC</td>
<td>Coprocessor operation</td>
</tr>
</tbody>
</table>

Note: In the table, the accumulators ACC0 and ACC1 are mnemonically expressed as A0 and A1, respectively.
### 2.3.2 Function-Extended Instructions of the OPSP-CPU

<table>
<thead>
<tr>
<th>Classification</th>
<th>Mnemonic</th>
<th>Function-extended content</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP function</td>
<td>MACHI</td>
<td>Accumulator A0 or A1 can be specified in the operand description.</td>
</tr>
<tr>
<td>instructions</td>
<td>MACLO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MULHI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MULLO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVFACHI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVFACLO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVFACMI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVTACHI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MVTACLO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAC</td>
<td>Accumulator A0 or A1 can be specified in the operand description. In addition, the result deriving after left-shifting the accumulator bit specified by an immediate (imm1) is rounded.</td>
</tr>
<tr>
<td></td>
<td>RACH</td>
<td></td>
</tr>
<tr>
<td>Arithmetic/logical</td>
<td>SLL</td>
<td>The parallel-executed instruction category has been changed from the left-side instruction (O–) to the both-side instruction (OS). (For details about the instruction category, refer to Section 2.5.3, “16-Bit Instruction List by Category.”)</td>
</tr>
<tr>
<td>instructions</td>
<td>SLLI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRAI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRLI</td>
<td></td>
</tr>
<tr>
<td>Load/store instructions</td>
<td>STB</td>
<td>Register update has been added to addressing modes.</td>
</tr>
<tr>
<td></td>
<td>STH</td>
<td></td>
</tr>
<tr>
<td>EIT related instructions</td>
<td>TRAP</td>
<td>The run-time BPC value has been changed from $BPC = PC + 4$ to $BPC = PC$ of the next instruction.</td>
</tr>
<tr>
<td></td>
<td>RTE</td>
<td>Return to the halfword boundary is possible.</td>
</tr>
</tbody>
</table>

Note: In the table, the accumulators ACC0 and ACC1 are mnemonically expressed as A0 and A1, respectively.
2

INSTRUCTION SET

2.4 Instruction Formats

The OPSP-CPU has two instruction formats: a 16-bit instruction, two of which are stored in pairs within the 32-bit word boundary, and a 32-bit instruction. (See Figure 2.4.1.)

The basic instruction formats of the OPSP-CPU are shown in Figure 2.4.2.

![Figure 2.4.1 16-Bit and 32-Bit Instructions](image)

- **16-bit instruction**
  - **<Instruction format>**
    - \( op1 \ R1 \ op2 \ R2 \)
    - \( op1 \ R1 \ c \)
    - \( op1 \ cond \ c \)
  - **<Operation of the instruction>**
    - \( R1 = R1 \ op R2 \) AND \( Rdest , Rsrc \)
    - \( R1 = R1 \ op c \) ADD \( Rdest , #imm8 \)
    - Branch (Short Displacement) BC pcdisp8

- **32-bit instruction**
  - **<Instruction format>**
    - \( op1 \ R1 \ op2 \ R2 \ c \)
    - \( op1 \ R1 \ op2 \ R2 \ c \)
    - \( op1 \ R1 \ c \)
    - \( op1 \ cond \ c \)
  - **<Operation of the instruction>**
    - \( R1 = R2 \ op c \) SRL3 \( Rdest , Rsrc , #imm16 \)
    - Compare and Branch BEQ \( Rsrc1 , Rsrc2 , \)
    - \( R1 = R1 \ op c \) LD24 \( Rdest , #imm24 \)
    - Branch BC pcdisp24

![Figure 2.4.2 Basic Instruction Formats](image)
2.5 Parallel Instruction Execution

2.5.1 Instruction Formats

The OPSP-CPU instruction set architecture supports parallel instruction execution for two 16-bit instructions that are stored in pairs within the word boundary. Whether two instructions are executed in parallel is determined by the value of the most significant bit (MSB) of each 16-bit instruction. (The MSB of each instruction only determines the method of instruction execution and does not affect the functionality of the instruction.)

The MSB of any 16-bit instruction that exists in the upper halfword location is always 0. If the MSB of the instruction that follows is also 0, then the two instructions are executed sequentially; if the MSB = 1, the two instructions are executed in parallel.

If the MSB of instruction B in Figure 2.5.2 is 0, then instruction A and instruction B are executed sequentially. If the MSB of instruction B is 1, then instruction A and instruction B are executed in parallel. If instruction B needs to be executed in parallel, it is automatically altered to an instruction whose MSB is set to 1 by the assembler. For the same reason, NOP instructions used to adjust the word alignment have always their MSB set to 1 by the assembler.

The MSB of all 32-bit instructions is always 1, so that they are not executed in parallel.

---

**Figure 2.5.1 Instruction Processing**

- **Instruction A → instruction B sequential**
  - 0 16-bit instruction A
  - 0 16-bit instruction B
- **Instruction A & instruction B in parallel**
  - 0 16-bit instruction A
  - 1 16-bit instruction B
  - 16-bit instruction B whose MSB is set to 1
- **Instruction A & NOP instruction in parallel**
  - 0 16-bit instruction A
  - 1111 0000 0000 0000
  - NOP instruction
  - NOP instruction whose MSB is set to 1

**Note:** The instruction located in the lower 16-bit part of the word boundary that is to be executed in parallel with the preceding instruction has its MSB automatically set to 1 by the assembler.
2.5.2 Parallel Instruction Execution in the OPSP

The OPSP-CPU has two pipelines: O pipe and S pipe. Two 16-bit instructions are executed in parallel using these two pipelines.

The 16-bit instructions executed in the S pipe include DSP function instructions and multiplication, arithmetic operation, logical operation, shift, comparison, transfer and NOP instructions. In the O pipe, on the other hand, all 16-bit instructions except DSP function and multiplication instructions can be executed.

Note that 32-bit instructions are not executed in parallel, and that all of them are executed in the O pipe.

![Diagram of Parallel Instruction Execution Mechanism of the OPSP-CPU](image)

2.5.3 16-Bit Instruction List by Category

The 16-bit instructions that can be executed in parallel are classified into three categories by the executable pipeline.

- Instructions that can be executed in only the O pipe (left-side instruction: O–)
- Instructions that can be executed in only the S pipe (right-side instruction: –S)
- Instructions that can be executed in both O and S pipes (both-side instruction: OS)

The 16-bit instructions classified by category are listed in Table 2.5.1.
### Table 2.5.1 16-Bit Instruction List by Category

<table>
<thead>
<tr>
<th>O– (left-side instructions)</th>
<th>OS (both-side instructions)</th>
<th>–S (right-side instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>ADD</td>
<td>MACHI</td>
</tr>
<tr>
<td>BCL</td>
<td>ADDI</td>
<td>MACLH1</td>
</tr>
<tr>
<td>BL</td>
<td>ADDV</td>
<td>MACLO</td>
</tr>
<tr>
<td>BNC</td>
<td>ADDX</td>
<td>MACWHI</td>
</tr>
<tr>
<td>BNCL</td>
<td>AND</td>
<td>MACQLO</td>
</tr>
<tr>
<td>BRA</td>
<td>CMP</td>
<td>MACWU1</td>
</tr>
<tr>
<td>BTST</td>
<td>CMPEQ</td>
<td>MSBLO</td>
</tr>
<tr>
<td>CLRPSW</td>
<td>CMPU</td>
<td>MUL</td>
</tr>
<tr>
<td>JC</td>
<td>CMPZ</td>
<td>MULHI</td>
</tr>
<tr>
<td>JL</td>
<td>LDI</td>
<td>MULLO</td>
</tr>
<tr>
<td>JMP</td>
<td>MV</td>
<td>MULWHI</td>
</tr>
<tr>
<td>JNC</td>
<td>NEG</td>
<td>NULWLO</td>
</tr>
<tr>
<td>LD</td>
<td>NOP</td>
<td>NULWU1</td>
</tr>
<tr>
<td>LDB</td>
<td>NOT</td>
<td>MVFACHI</td>
</tr>
<tr>
<td>LDH</td>
<td>OR</td>
<td>MVFACLO</td>
</tr>
<tr>
<td>LDUB</td>
<td>PCMPBZ</td>
<td>MVFACMI</td>
</tr>
<tr>
<td>LDUH</td>
<td>SLL</td>
<td>MVTACHI</td>
</tr>
<tr>
<td>LOCK</td>
<td>SLLI</td>
<td>MVTACLO</td>
</tr>
<tr>
<td>MVFC</td>
<td>SRA</td>
<td>RAC</td>
</tr>
<tr>
<td>MVTC</td>
<td>SRAI</td>
<td>RACH</td>
</tr>
<tr>
<td>RTE</td>
<td>SRL</td>
<td>SADD</td>
</tr>
<tr>
<td>SETPSW</td>
<td>SRLI</td>
<td></td>
</tr>
<tr>
<td>SC</td>
<td>SUB</td>
<td></td>
</tr>
<tr>
<td>SNC</td>
<td>SUBV</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>SUBX</td>
<td></td>
</tr>
<tr>
<td>STB</td>
<td>XOR</td>
<td></td>
</tr>
<tr>
<td>STH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNLOCK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.5.4 Positions of Parallel Executed Instructions

The 16-bit instruction pairs that can be executed in parallel are limited to the following four combinations of instruction categories.

- Left-side instruction and right-side instruction (O– and –S)
- Left-side instruction and both-side instruction (O– and OS)
- Both-side instruction and right-side instruction (OS and –S)
- Both-side instruction and both-side instruction (OS and OS)

The locations of instruction categories when 16-bit instruction pairs are executed in parallel are shown below.

- Left-side instruction (O–) located in the upper 16-bit part and the right-side instruction (–S) located in the lower 16-bit part
- Left-side instruction (O–) located in the upper 16-bit part and the both-side instruction (OS) located in the lower 16-bit part
- Both-side instruction (OS) located in the upper 16-bit part and the right-side instruction (–S) located in the lower 16-bit part
- Both-side instruction (OS) located in the upper 16-bit part and the other both-side instruction located in the lower 16-bit part

However, if a NOP instruction is located in the lower 16-bit part for the purpose of word alignment, a right-side instruction (–S) may be located in the upper 16-bit part as an instruction pair to be executed in parallel.

![Figure 2.5.3 Locations of Parallel Executable Instruction Categories](image-url)

Note: The instruction located in the lower 16-bit part of the word boundary that is to be executed in parallel with the preceding instruction has its MSB automatically set to 1 by the assembler.
2.5.5 Operand Interferences

Two parallel executed 16-bit instructions are executed independently of each other, and not sequenced in time. When executed in parallel, the two instructions are handled as having no mutual dependency with regard to the operand, so that they are not subject to interlock processing. Please be aware of this point when writing a program.

The value of the source operand referenced by a parallel executed instruction pair is one that was stored in the operand immediately before the CPU started executing the instructions in parallel. For example, if in a parallel executed instruction pair, one instruction writes to a register and the other instruction references it, the register value that is referenced is one that was stored in the register immediately before the CPU started executing the instruction pair in parallel. Furthermore, after the instruction pair was executed, the result is written to the register.

Note, however, that if two instructions are executed in parallel that write to the same register (collision of writes to a register), program operation cannot be guaranteed.

(1) Examples of operand interferences in general-purpose registers

The following shows typical examples of operand interferences in general-purpose registers attributable to two transfer instructions (MV instructions).

Example 1: MV R1,R0 || MV R2,R1

Example 2: MV R1,R0 || MV R1,R2

Note: the symbol || denotes that two instructions are executed in parallel.

In example 1, one of the two instructions in pairs writes to a register (R1) and the other references it. In this case, R1 is assigned the value of R0. Similarly, R2 is assigned the value of R1 before assignment to R1 (i.e., the value of R1 before the instruction “MV R1,R0” is executed).

Example 2 is an example where two instructions in the instruction pair write to the same register (collision of writes to a register). In this case, the registers accessed for write by two MV instructions both are R1, so that the value of R1 after instruction execution is indeterminate.

(2) Examples of operand interferences in control registers

In addition to general-purpose registers, operand interferences will occur in control registers such as the PSW and CBR that include the condition bit (C).

Example 3: When two instructions are executed successively

```
CMP R1,R0
BC _label
```

Example 4: When two instructions are executed in parallel

```
CMP R1,R0 || BC _label
```

Note: the symbol || denotes that two instructions are executed in parallel.

In example 3, the comparison instruction (CMP) is executed before the conditional branch instruction (BC) is executed. In this case, the condition bit (C) is updated as a result of the CMP instruction executed, and the BC instruction references this updated condition bit (C) to determine whether or not to branch.

In example 4, the CMP and the BC instructions are executed in parallel. The BC instruction references the condition bit (C) before the CMP instruction is executed, to determine whether or not to branch. Be aware that the condition bit (C) is referenced before it is operated on by execution of the CMP instruction. The result of the CMP instruction executed is reflected in the condition bit (C) after parallel instruction execution.
Furthermore, if two instructions are executed in parallel that will change the condition bit (C), the value of the condition bit (C) after instruction execution becomes indeterminate as in the case of a collision of writes to a register that occurs in general-purpose registers. Shown below are examples where the condition bit (C) becomes indeterminate after an instruction pair is executed in parallel.

Example 5: CMP R1,R2 || ADDX R3,R4
Example 6: MVTC R1,PSW || ADDX R1,R2
Example 7: TRAP #1 || CMP R3,R4
Example 8: RTE || ADDX R3,R4

Note: the symbol || denotes that two instructions are executed in parallel.
This page is blank for reasons of layout.
3.1 Guide to Detailed Instruction Description

The following outlines each item that is described in the detailed description of instructions in the pages to follow.

[Mnemonic]

The mnemonics of the OPSP-CPU consist of an instruction and the operand description that follows. The operand is the target to be operated on by the instruction.

Table 3.1.1 List of Operand description

<table>
<thead>
<tr>
<th>Operand description</th>
<th>Addressing mode</th>
<th>Target to be operated on by instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Register direct</td>
<td>General-purpose register of the OPSP-CPU (R0–R15)</td>
</tr>
<tr>
<td>CR</td>
<td>Control register</td>
<td>Control register of the OPSP-CPU (CR0=PSW, CR1=CBR, CR2=SPI, CR3=SPU, CR5=EVB, CR6=BPC)</td>
</tr>
<tr>
<td>CPR</td>
<td>Coprocessor register</td>
<td>Register of the coprocessor connected to the OPSP-CPU</td>
</tr>
<tr>
<td>A</td>
<td>Accumulator</td>
<td>Content of the OPSP-CPU accumulator (A0, A1)</td>
</tr>
<tr>
<td>@(disp, Rn)</td>
<td>Register relative indirect</td>
<td>Memory content whose address is indicated by (register value) + (16-bit constant that is sign-extended to 32 bits)</td>
</tr>
<tr>
<td>@Rn+</td>
<td>Register indirect + register update</td>
<td>Register value incremented by 4, 2, or 1 (Memory content whose address is indicated by a preupdate register value)</td>
</tr>
<tr>
<td>@+Rn</td>
<td>Register indirect + register update</td>
<td>Register value incremented by 4 (Memory content whose address is indicated by an updated register value)</td>
</tr>
<tr>
<td>@- Rn</td>
<td>Register indirect + register update</td>
<td>Register value decremented by 4 (Memory content whose address is indicated by an updated register value)</td>
</tr>
<tr>
<td>#imm</td>
<td>Immediate</td>
<td>Immediate value (For details on how the value is handled, refer to the detailed description of each instruction.)</td>
</tr>
<tr>
<td>pcdisp</td>
<td>PC relative</td>
<td>Memory content whose address is indicated by (PC value) + (8, 16, or 24-bit displacement which is sign-extended to 32 bits and then shifted left 2 bits).</td>
</tr>
</tbody>
</table>

Note: In operand descriptions “Rsrc” and “Rdest,” src and dest each represent a general-purpose register number (0–15). In operand descriptions “CRsrc” and “CRdest,” src and dest each represent a control register number (0–3, 5, or 6). In operand descriptions “Asrc” and “Adest,” src and dest each represent an accumulator number (0 or 1).
3

INSTRUCTIONS

3.1 Guide to Detailed Instruction Description

[Function]

Operation of each instruction is described by first outlining what the instruction does and then showing a C language based description of the operation. The description of instruction operation is outlined below.

Table 3.1.2 Description of Operations (Operators)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition (binary operator)</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction (binary operator)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication (binary operator)</td>
</tr>
<tr>
<td>/</td>
<td>Division (binary operator)</td>
</tr>
<tr>
<td>%</td>
<td>Remainder calculation (binary operator)</td>
</tr>
<tr>
<td>++</td>
<td>Increment (unary operator)</td>
</tr>
<tr>
<td>--</td>
<td>Decrement (unary operator)</td>
</tr>
<tr>
<td>-</td>
<td>Sign inversion (unary operator)</td>
</tr>
<tr>
<td>=</td>
<td>Assign right side to left side (assignment operator)</td>
</tr>
<tr>
<td>+=</td>
<td>Add left and right side variables and assign the result to left side (assignment operator)</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract right side variable from left side variable and assign the result to left side (assignment operator)</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than (relational operator)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Smaller than (relational operator)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal (relational operator)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Smaller than or equal (relational operator)</td>
</tr>
<tr>
<td>==</td>
<td>Equal (relational operator)</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal (relational operator)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND (logical operator)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Create conditional expression (conditional operator)</td>
</tr>
</tbody>
</table>
Table 3.1.3 Description of Operations (Bitwise Operators)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>Shift the bit left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift the bit right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive-OR (EXOR)</td>
</tr>
<tr>
<td>~</td>
<td>Bit inversion</td>
</tr>
</tbody>
</table>

Table 3.1.4 Data Types

<table>
<thead>
<tr>
<th>Representation</th>
<th>Type</th>
<th>Signed or unsigned</th>
<th>Bit length</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Integer</td>
<td>Signed</td>
<td>8</td>
<td>-128 to +127</td>
</tr>
<tr>
<td>short</td>
<td>Integer</td>
<td>Signed</td>
<td>16</td>
<td>-32,768 to +32,767</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>Signed</td>
<td>32</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>unsigned char</td>
<td>Integer</td>
<td>Unsigned</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Integer</td>
<td>Unsigned</td>
<td>16</td>
<td>0 to 655,535</td>
</tr>
<tr>
<td>unsigned int</td>
<td>Integer</td>
<td>Unsigned</td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>signed64bit</td>
<td>Integer</td>
<td>Signed</td>
<td>64</td>
<td>Signed 64-bit integer</td>
</tr>
</tbody>
</table>

(when operating on accumulators)
[Description]
The function of each instruction is detailed here. Furthermore, changes of the condition bit (C) in the PSW register that occur as a result of execution of the instruction are described.

[EIT occurrence]
A generated EIT means an EIT event (exception, interrupt, or trap) that may occur as a result of execution of the instruction. The EIT events that are likely to occur as a result of instruction execution include an address exception, trap and a privileged instruction exception.

[Encoding]
A 16-bit or 32-bit instruction bit pattern is shown. In the instruction format, src and dest each represent the corresponding register number, while imm and disp represent immediate and displacement values, respectively. (The magnitude of the numeric value that is assigned to each bit field is determined by the field width.) For details about the instruction format, refer to Section 2.3, “Instruction Formats,” in Chapter 2.
3.2 Detailed Description of Instructions

Each instruction of the OPSP-CPU is described in detail beginning with the next page. Instructions are listed in alphabetical order. Note that each page consists of the items described below.

- **Instruction Name**: (instruction type and full name are in center)
- **Instruction Mnemonic**: [Mnemonic]
- **Instruction Function**: (expression corresponds to C language method)
- **Instruction Description and Effect on Condition Bit (C)**
- **EIT Events that May Occur when this Instruction is Executed**
- **16- or 32-bit Instruction Format**
ADD

arithmetic operation instruction
Add

[Mnemonic]
ADD Rdest, Rsrc

[Function]
Add
Rdest = Rdest + Rsrc

[Description]
ADD adds Rs to Rdest and puts the result in Rdest.
The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

0000 dest 1010 src ADD Rdest, Rsrc
### ADD3

**arithmetic operation instruction**
Add 3-operand

#### Mnemonic

```
ADD3 Rdest, Rsrc, #imm16
```

#### Function

Add

```
Rdest = Rsrc + (signed short) imm16;
```

#### Description

ADD3 adds the 16-bit immediate value to Rsrc and puts the result in Rdest. The immediate value is sign-extended to 32 bits before the operation.

The condition bit (C) dose not changed.

#### EIT occurrence

None

#### Encoding

```
1000 dest 1010 src imm16
```

```
ADD3 Rdest, Rsrc, #imm16
```
ADDI

arithmetic operation instruction
Add immediate

[Mnemonic]
ADDI Rdest,#imm8

[Function]
Add
Rdest = Rdest + (signed char) imm8;

[Description]
ADDI adds the 8-bit immediate value to Rdest and puts the result in Rdest. The immediate value is sign-extended to 32 bits before the operation.
The condition bit (C) does not changed.

[EIT occurrence]
None

[Encoding]

0100 dest imm8 ADDI Rdest,#imm8
ADDV

arithmetic operation instruction
Add with overflow checking

[Mnemonic]
ADDV Rdest, Rsrc

[Function]
Add
Rdest = (signed) Rdest + (signed) Rsrc;
C = overflow ? 1 : 0

[Description]
ADDV adds Rsrc to Rdest and puts the result in Rdest.
The condition bit (C) is set when the addition results in overflow; otherwise it is cleared.

[EIT occurrence]
None

[Encoding]

0000 dest 1000 src ADDV Rdest, Rsrc
ADDV3

arithmetic operation instruction
Add 3-operand with overflow checking

[Mnemonic]
ADDV3 Rdest,Rsrc,#imm16

[Function]
Add
Rdest = ( signed ) Rsrc+ ( signed ) ( (signed short)imm16);
C = overflow ? 1 : 0

[Description]
ADDV3 adds the 16-bit immediate value to Rsrc and puts the result in Rdest. The immediate value is
sign-extended to 32 bits before it is added to Rsrc.
The condition bit (C) is set when the addition results in overflow; otherwise it is cleared.

[EIT occurrence]
None

[Encoding]

```
<table>
<thead>
<tr>
<th>0000</th>
<th>dest</th>
<th>0000</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

ADDV3 Rdest,Rsrc,#imm16
### ADDX

**Mnemonic**

ADDX Rdest,Rsrc

**Function**

Add

\[ \text{Rdest} = (\text{unsigned}) \text{Rdest} + (\text{unsigned}) \text{Rsrc} + C; \]

\[ C = \text{carry}_{\text{out}} ? 1 : 0; \]

**Description**

ADDX adds Rsrc and C to Rdest, and puts the result in Rdest. The condition bit (C) is set when the addition result cannot be represented by a 32-bit unsigned integer; otherwise it is cleared.

**EIT occurrence**

None

**Encoding**

```
 0000  dest  1001  src
```

ADDX Rdest,Rsrc
AND

logic operation instruction
AND

[Mnemonic]

AND Rdest, Rsnc

[Function]

Logical AND
Rdest = Rdest & Rsnc;

[Description]

AND computes the logical AND of the corresponding bits of Rdest and Rsnc and puts the result in Rdest.
The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

```
0000 dest 1100 src AND Rdest, Rsnc
```
**AND3**

logic operation instruction
AND 3-operand

[Mnemonic]

\[
\text{AND3 } \text{Rdest}, \text{Rsrc}, \#\text{imm16}
\]

[Function]

Logical AND

\[
\text{Rdest} = \text{Rsrc} \& (\text{unsigned short}) \text{imm16};
\]

[Description]

AND3 computes the logical AND of the corresponding bits of Rsrc and the 16-bit immediate value, which is zero-extended to 32 bits, and puts the result in Rdest.

The condition bit (C) does not changed.

[EIT occurrence]

None

[Encoding]

\[
\begin{array}{c|c|c|c}
1000 & \text{dest} & 1100 & \text{src} \\
\end{array} \quad \begin{array}{c|c}
\text{imm16} \\
\end{array}
\]

\[
\text{AND3 } \text{Rdest}, \text{Rsrc}, \#\text{imm16}
\]
Branch instruction
Branch on C-bit

**BC**

### [Mnemonic]

1. BC **pcdisp8**
2. BC **pcdisp24**

### [Function]

Branch

1. if \( C = 1 \) \( PC = ( PC \& 0xfffffffc ) + ( ( \text{signed char} ) \text{pcdisp8} ) \ll 2 \);
2. if \( C = 1 \) \( PC = ( PC \& 0xfffffffc ) + ( \text{sign}_\text{extend} ( \text{pcdisp24} ) \ll 2 ) \);

where

\[
\text{#define sign}_\text{extend}(x) ( ( ( \text{signed} ) ( (x)\ll 8 ) )\gg 8 )
\]

### [Description]

BC causes a branch to the specified label when the condition bit (C) is 1.

There are two instruction formats; which allows software, such as an assembler, to decide on the better format. The condition bit (C) dose not changed.

### [EIT occurrence]

None

### [Encoding]

<table>
<thead>
<tr>
<th>0111 1100</th>
<th><strong>pcdisp8</strong></th>
<th>BC <strong>pcdisp8</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1100</td>
<td><strong>pcdisp24</strong></td>
<td>BC <strong>pcdisp24</strong></td>
</tr>
</tbody>
</table>
Branch instruction
Branch and link on C-bit

[Mnemonic]

(1) BCL pcdisp8
(2) BCL pcdisp24

[Function]

Branch

(1) if ( C == 1 ) {
    R14 = ( PC & 0xfffffffc ) + 4 ;
    PC = ( PC & 0xfffffffc ) + ( (( signed char ) pcdisp8 ) << 2 ) ;
}

(2) if ( C == 1 ) {
    R14 = ( PC & 0xfffffffc ) + 4 ;
    PC = ( PC & 0xfffffffc ) + ( sign_extend ( pcdisp24 ) << 2 ) ;
}

where

#define sign_extend(x)     ( (( signed ) (( x ) << 8 ) ) >> 8 )

[Description]

When the condition bit (C) = 1, BCL causes a branch to the specified label and store the return address in R14.

There are two instruction formats; this allows software, such as an assembler, to decide on the better format.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

```
0111 1000 pcdisp8  BCL pcdisp8

1111 1000 pcdisp24  BCL pcdisp24
```
BCLR  

bit operation instruction
Bit clear

[Mnemonic]

BCLR #bitpos,@(disp16, Rsrc)

[Function]

Bit operation for memory contents. Set a specified bit to 0.

*(char *)(Rsrc+(signed short)disp16) &= ~(1 << (7 - bitpos));

[Description]

BCLR reads byte data in memory from the address specified by Rsrc and a 16-bit displacement and stores the read value after changing its bit specified by bitpos to 0.

The displacement is sign-extended before address calculation. bitpos is specified for bits 0–7 where MSB = 0 and LSB = 7. Memory is accessed in bytes.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

1010 0:bitpos 0111 src disp16

BCLR #bitpos,@(disp16, Rsrc)
**BEQ** branch instruction

**Mnemonic**

```
BEQ Rs1c, Rs1c, pcdisp16
```

**Function**

Branch

```
if ( Rs1c = Rs1c ) PC = ( PC & 0xffffffff ) + ( ( signed short ) pcdisp16 ) << 2);
```

**Description**

BEQ causes a branch to the specified label when Rs1c is equal to Rs1c.

The condition bit (C) does not change.

**EIT occurrence**

None

**Encoding**

```
 1011 src1 0000 src2 pcdisp16
```

BEQ Rs1c, Rs1c, pcdisp16
BEQZ
branch instruction
Branch on equal to zero

[Mnemonic]
BEQZ Rsrc,pcdisp16

[Function]
Branch
if ( Rsrc == 0 ) PC = ( PC & 0xfffffffc ) + ( ( ( signed short ) pcdisp16 ) << 2);

[Description]
BEQZ causes a branch to the specified label when Rsrc is equal to zero.
The condition bit (C) does not changed.

[EIT occurrence]
None

[Encoding]

```
1011 0000 1000 src pcdisp16
```

BEQZ Rsrc,pcdisp16
BGEZ
branch instruction
Branch on greater than or equal to zero

[Mnemonic]
BGEZ Rsr[1], pcdisp16

[Function]
Branch
if ((signed) Rsr >= 0) PC = (PC & 0xfffffffc) + ((signed short) pcdisp16) << 2;

[Description]
BGEZ causes a branch to the specified label when Rsr treated as a signed 32-bit value is greater than or equal to zero.
The condition bit (C) does not changed.

[EIT occurrence]
None

[Encoding]

1011 0000 1011 src pcdisp16

BGEZ Rsr[1], pcdisp16
**BGTZ**

branch instruction
Branch on greater than zero

[Mnemonic]

\[ \text{BGTZ } \text{Rsrc,pcdisp16} \]

[Function]

Branch

\[ \text{if ( (signed) Rsrc > 0 ) PC = ( PC & 0xfffffffc ) + ( ( ( signed short ) pcdisp16 ) \ll 2);} \]

[Description]

BGTZ causes a branch to the specified label when Rsrc treated as a signed 32-bit value is greater than zero. The condition bit (C) does not changed.

[EIT occurrence]

None

[Encoding]

\[
\begin{array}{c|c|c|c}
1011 & 0000 & 1101 & \text{src} \\
\end{array}
\]

\[ \text{BGTZ } \text{Rsrc,pcdisp16} \]
3.2 Detailed Description of Instructions

BL

branch instruction
Branch and link

[Mnemonic]
(1) BL pcdisp8
(2) BL pcdisp24

[Function]
branch
(1) R14 = ( PC & 0xffffffff ) + 4;
    PC = ( PC & 0xffffffff ) + ( ( signed char ) pcdisp8 ) << 2 );
(2) R14 = ( PC & 0xffffffff ) + 4;
    PC = ( PC & 0xffffffff ) + ( sign_extend ( pcdisp24 ) << 2 );
where
#define sign_extend(x)   ( ( ( signed ) ( (x)<< 8 ) ) >>8 )

[Description]
BL causes an unconditional branch to the address specified by the label and puts the return address in R14.
There are two instruction formats; this allows software, such as an assembler, to decide on the better format.
The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

0111 1110  pcdisp8  BL pcdisp8

1111 1110  pcdisp24  BL pcdisp24
BLEZ
branch instruction
Branch on less than or equal to zero

[Mnemonic]
BLEZ Rsfc,pcdisp16

[Function]
Branch
if ( (signed) Rsfc <= 0 ) PC = ( PC & 0xfffffffc ) + ( ( ( signed short ) pcdisp16 ) << 2);

[Description]
BLEZ causes a branch to the specified label when the contents of Rsfc treated as a signed 32 bit value, is less than or equal to zero.
The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

<table>
<thead>
<tr>
<th>1111</th>
<th>0000</th>
<th>1100</th>
<th>src</th>
<th>pcdisp16</th>
</tr>
</thead>
</table>

BLEZ Rsfc,pcdisp16
BLTZ
branch instruction
Branch on less than zero

[Mnemonic]
BLTZ Rsrc,pcdisp16

[Function]
Branch
if ((signed) Rsrc < 0 ) PC = ( PC & 0xfffffffc ) + ( ( ( signed short ) pcdisp16 ) << 2);

[Description]
BLTZ causes a branch to the specified label when Rsrc treated as a signed 32-bit value is less than zero.
The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

1011 0000 1010 src pcdisp16

BLTZ Rsrc,pcdisp16
BNC
branch instruction
Branch on not C-bit

[Mnemonic]

1. BNC pcdisp8
2. BNC pcdisp24

[Function]

Branch
(1) if (C==0) PC = (PC & 0xfffffffc) + ((signed char) pcdisp8) << 2;
(2) if (C==0) PC = (PC & 0xfffffffc) + (sign_extend(pcdisp24) << 2);
where
#define sign_extend(x) ( ((signed) ((x)<< 8 )) >>8 )

[Description]

BNC causes a branch to the specified label when the condition bit (C) is 0. There are two instruction formats; this allows software, such as an assembler, to decide on the better format.

The condition bit (C) does not changed.

[EIT occurrence]

None

[Encoding]

0111 1101 pcdisp8
BNC pcdisp8

1111 1101 pcdisp24
BNC pcdisp24
BNCL

branch instruction
Branch and link on not C-bit

[Mnemonic]

(1) BNCL pcdisp8
(2) BNCL pcdisp24

[Function]

Branch
(1) if ( C == 0 ) {
    R14 = ( PC & 0xfffffffc ) + 4 ;
    PC = ( PC & 0xfffffffc ) + ((( signed char ) pcdisp8 ) << 2 ) ;
}
(2) if ( C == 0 ) {
    R14 = ( PC & 0xfffffffc ) + 4 ;
    PC = ( PC & 0xfffffffc ) + ( sign_extend ( pcdisp24 ) << 2 ) ;
}

where
#define sign_extend(x)  ((( signed ) (( x ) << 8 ) ) >> 8 )

[Description]

When the condition bit (C) = 0, BNCL causes a branch to the specified label and stores the return address in R14. There are two instruction formats; this allows software, such as an assembler, to decide on the better format. The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111 1001</td>
<td>pcdisp8</td>
</tr>
<tr>
<td>1111 1001</td>
<td>pcdisp24</td>
</tr>
</tbody>
</table>

BNCL pcdisp8
BNCL pcdisp24
### BNE

#### branch instruction
Branch on not equal to

#### [Mnemonic]

\[
\text{BNE } \text{Rsrc1, Rsrc2, pcdisp16}
\]

#### [Function]

Branch

\[
\text{if ( Rsrc1 \neq Rsrc2 ) } \text{PC} = ( \text{PC} \& \text{0xfffffffc} ) + \left( \left( \text{signed short} \right) \text{pcdisp16} \right) << 2);
\]

#### [Description]

BNE causes a branch to the specified label when Rsrc1 is not equal to Rsrc2. The condition bit (C) dose not changed.

#### [EIT occurrence]

None

#### [Encoding]

\[
\text{1011 src1 0001 src2 pcdisp16}
\]

\[
\text{BNE Rsrc1, Rsrc2, pcdisp16}
\]
### BNEZ

**branch instruction**

Branch on not equal to zero

---

**[Mnemonic]**

BNEZ Rsrc,pcdisp16

**[Function]**

Branch

if ( Rsrc != 0 ) PC = ( PC & 0xffffffff ) + ( ( ( signed short ) pcdisp16 ) << 2);

**[Description]**

BNEZ causes a branch to the specified label when Rsrc is not equal to zero.
The condition bit (C) dose not changed.

**[EIT occurrence]**

None

**[Encoding]**

```
   1011  0000  1001  src  pcdisp16

BNEZ Rsrc,pcdisp16
```
3.2 Detailed Description of Instructions

**BRA**

branch instruction

Branch

[Mnemonic]

1. **BRA pcdisp8**
2. **BRA pcdisp24**

[Function]

Branch

1. \( PC = ( PC \& 0xfffffffc) + ( (\ ( \text{signed char} \ ) \ pcdisp8) \ << 2); \)
2. \( PC = ( PC \& 0xfffffffc) + ( \text{sign}\_\text{extend} \ ( pcdisp24) \ << 2); \)

where

\[
\text{#define sign}\_\text{extend}(x) \left( \left( (\ \text{signed} \ (x) \ << 8 ) \right) >> 8 \right)
\]

[Description]

BRA causes an unconditional branch to the address specified by the label.

There are two instruction formats; this allows software, such as an assembler, to decide on the better format.

[EIT occurrence]

None

[Encoding]

```
0111 1111  pcdisp8
```

BRA pcdisp8

```
1111 1111
```

BRA pcdisp24
3.2 Detailed Description of Instructions

BSET

bit operation instruction
Bit set

[Mnemonic]

BSET #bitpos, @(disp16,Rsrc)

[Function]

Bit operation on memory content. Set a specified bit to 1.
*(char *)(Rsrc+(signed short)disp16) |= (1 << (7 - bitpos));

[Description]

BSET reads byte data in memory from the address specified by Rsrc and a 16-bit displacement and stores the
read value after changing its bit specified by bitpos to 1.
The displacement is sign-extended before address calculation. bitpos is specified for bits 0–7 where MSB = 0 and
LSB = 7. Memory is accessed in bytes.

[EIT occurrence]

None

[Encoding]

```
0110 0 bitpos 0110 src disp16

BSET #bitpos, @(disp16,Rsrc)
```
**INSTRUCTIONS**

### 3.2 Detailed Description of Instructions

**BTST**

#### [Mnemonic]

```
BTST #bitpos, Rsrc
```

#### [Function]

Bit operation to extract a specified register bit.

```
C = (Rsrc >> (7 - bitpos)) & 1;
```

#### [Description]

BTST extracts a bit specified by `bitpos` from the 8 low-order bits of `Rsrc` and sets it in the condition bit (C). "bitpos" is specified for bits 0–7 where LSB = 7.

#### [EIT occurrence]

None

#### [Encoding]

```
0000 0 : bitpos 1111 src BTST #bitpos, Rsrc
```
CLRPSW  

bit operation instruction

Clear PSW

[Mnemonic]

CLRPSW  #imm8

[Function]

Set SM, IE, PM, CE or C bit in the PSW to 0.

PSW &=(unsigned char)imm8 | 0x0000ff00

[Description]

Logically AND the inverse of the 8-bit value specified by imm8 with the 8 low-order bits in the PSW (bits 24–31) bitwise and write the result to the 8 low-order bits in the PSW bit by bit.

[EIT occurrence]

Privilege instruction exception(PIE)

[Encoding]

0111 0010  imm8  CLRPSW  #imm8
**CMP**

**compare instruction**

**Compare**

**[Mnemonic]**

CMP Rsr1, Rsr2

**[Function]**

Compare

C = ((signed) Rsr1 < (signed) Rsr2) ? 1:0;

**[Description]**

The condition bit (C) is set to 1 when Rsr1 is less than Rsr2. The operands are treated as signed 32-bit values.

**[EIT occurrence]**

None

**[Encoding]**

```
0000 src1 0100 src2 CMP Rsr1, Rsr2
```
**CMPEQ**

**[Mnemonic]**

\[ \text{CMPEQ } R\text{src1}, R\text{src2} \]

**[Function]**

Compare

\[ C = ( R\text{src1} == R\text{src2} ) ? 1 : 0 ; \]

**[Description]**

When Rsrc1 and Rsrc2 are equal, the condition bit (C) is set to 1.

**[EIT occurrence]**

None

**[Encoding]**

```
0000 src1 0110 src2 CMPEQ Rsrc1,Rsrc2
```
### CMPI

#### [Mnemonic]

CMPI Rsrc, #imm16

#### [Function]

Compare

\[ C = (\text{(signed) Rsrc} < \text{(signed)} (\text{(signed short) imm16})) \ ? \ 1:0; \]

#### [Description]

The condition bit (C) is set when Rsrc is less than 16-bit immediate value. The operands are treated as signed 32-bit values. The immediate value is sign-extended to 32-bit before the operation.

#### [EIT occurrence]

None

#### [Encoding]

\[
\begin{array}{cccc}
1000 & 0000 & 0100 & \text{src} \\
\end{array}
\]

CMPI Rsrc, #imm16
**CMPU**

**compare instruction**

**Compare unsigned**

---

**Mnemonic**

`CMPU Rsrl,Rsrc2`

---

**Function**

Compare

\[ C = ((\text{unsigned}) \ Rsrc1 < (\text{unsigned}) \ Rsrc2) \ ? \ 1:0; \]

---

**Description**

The condition bit (C) is set when \( Rsrc1 \) is less than \( Rsrc2 \). The operands are treated as unsigned 32-bit values.

---

**EIT occurrence**

None

---

**Encoding**

```
0000 src1 0101 src2 CMPU Rsrc1,Rsrc2
```
CMPUI

compare instruction
Compare unsigned immediate

[Mnemonic]

CMPUI Rs, #imm16

[Function]

Compare

\[ C = \left( (\text{unsigned}) \, \text{Rs} < (\text{unsigned}) \, (\text{signed short}) \, \text{imm16} \right) \, ? \, 1:0; \]

[Description]

The condition bit (C) is set when Rs is less than the 16-bit immediate value. The operands are treated as unsigned 32-bit values. The immediate value is sign-extended to 32-bit before the operation.

[EIT occurrence]

None

[Encoding]

```
  1000 0000 0101 src  imm16
  CMPUI Rs, #imm16
```
**CMPZ**

**compare instruction**
*Compare equal to zero*

**[Mnemonic]**

```plaintext
CMPZ Rsrc
```

**[Function]**

Compare

```plaintext
C = ( Rsrc == 0 ) ? 1 : 0;
```

**[Description]**

The condition bit (C) is set when `Rsrc` is zero.

**[EIT occurrence]**

None

**[Encoding]**

```plaintext
0000 0000 0111 src CMPZ Rsrc
```
3.2 Detailed Description of Instructions

DIV

multiply and divide instruction

Divide

[Mnemonic]

DIV Rdest, Rsrc

[Function]

Signed division

\[ \text{Rdest} = (\text{signed}) \frac{\text{Rdest}}{(\text{signed}) \text{Rsrc}}; \]

[Description]

DIV divides Rdest by Rsrc and puts the quotient in Rdest. The operands are treated as signed 32-bit values and the result is rounded toward zero.

The condition bit (C) does not change.

When Rsrc is zero, Rdest does not change.

[EIT occurrence]

None

[Encoding]

```
1001  dest  0000  src  0000  0000  0000  0000
```

DIV Rdest, Rsrc
DIVB
multiply and divide instruction
Divide byte

[Mnemonic]
DIVB Rdest,Rsrc

[Function]
Signed division
Rdest = (signed char)Rdest / (signed)Rsrc ;

[Description]
DIVB divides Rdest by Rsrc and store the quotient in Rdest. Of the operands of this instruction, the dividend is handled as a signed 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as a signed 32-bit value, and the quotient is rounded toward zero.
The condition bit (C) does not change.
When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]
None

[Encoding]

```
 1001 | dest | 0000 | src | 0000 | 0000 | 0001 | 1000
```

DIVB Rdest,Rsrc
DIVH multiply and divide instruction
Divide Half-word

[Mnemonic]

DIVH Rdest, Rsrc

[Function]

Signed division
Rdest = ( signed short ) Rdest / ( signed ) Rsrc;

[Description]

DIVH divides Rdest by Rsrc and store the quotient in Rdest. Of the operands of this instruction, the dividend is handled as a signed 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as a signed 32-bit value, and the quotient is rounded toward zero.
The condition bit (C) does not change.
When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]

None

[Encoding]

```
1001  dest  0000  src  0000  0000  0001  0000
```

DIVH Rdest, Rsrc
DIVU

multiply and divide instruction
Divide unsigned

[Mnemonic]
DIVU Rdest,Rsrc

[Function]
Unsigned division
\[ Rdest = \text{(unsigned)} \ Rdest / \text{(unsigned)} \ Rsrc; \]

[Description]
DIVU divides Rdest by Rsrc and puts the quotient in Rdest. The operands are treated as unsigned 32-bit values and the result is rounded toward zero.
The condition bit (C) dose not changed.
When Rsrc is zero, Rdest dose not changed.

[EIT occurrence]
None

[Encoding]

```
1001 dest 0001 src 0000 0000 0000 0000
```

DIVU Rdest,Rsrc
DIVUB multiply and divide instruction
Divide unsigned byte

[Mnemonic]
DIVUB Rdest,Rsrc

[Function]
Unsigned division
Rdest = (unsigned char)Rdest / (unsigned)Rsrc;

[Description]
DIVUB divides Rdest by Rsrc and stores the quotient in Rdest.
Of the operands of this instruction, the dividend is handled as an unsigned 8-bit value, with the 24 high-order bits
(bits 0–23) ignored. The divisor is handled as an unsigned 32-bit value, and the quotient is rounded toward zero.
The condition bit (C) does not change.
When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]
None

[Encoding]

```
1001 dest 0001 src 0000 0000 0001 1000
DIVUB Rdest,Rsrc
```
DIVUH multiply and divide instruction
Divide unsigned halfword

[Mnemonic]
DIVUH Rdest,Rsrc

[Function]
Unsigned division
Rdest = (unsigned short)Rdest / (unsigned)Rsrc;

[Description]
DIVUH divides Rdest by Rsrc and stores the quotient in Rdest.
Of the operands of this instruction, the dividend is handled as an unsigned 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as an unsigned 32-bit value, and the quotient is rounded toward zero.
The condition bit (C) does not change.
When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]
None

[Encoding]

```
1001 dest 0001 src 0000 0000 0001 0000
```

DIVUH Rdest,Rsrc
JC

branch instruction
Jump on C-bit

[Mnemonic]

JC Rsfc

[Function]

Jump
if (C == 1) PC = Rsfc & 0xffffffff;

[Description]

JC causes a jump to the address specified by Rsfc when the condition bit (C) = 1.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

```
0001 1100 1100 src
                JC Rsfc
```
**JL**

branch instruction
Jump and link

[Mnemonic]

```
JL Rsrc
```

[Function]

Subroutine call (register direct)
```
R14 = ( PC & 0xffffffff ) + 4;
PC = Rsrc & 0xffffffff;
```

[Description]

JL causes an unconditional jump to the address specified by Rsrc and puts the return address in R14.
The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

```
0001 1110 1100 src JL Rsrc
```
JMP

[Branch]

JMP Rsrc

[Function]

Jump

PC = Rsr & 0xffffffff;

[Description]

JMP causes an unconditional jump to the address specified by Rsr.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

0001 1111 1100 src JMP Rsrc
JNC

branch instruction
Jump on not C-bit

[Mnemonic]

JNC Rssrc

[Function]

Jump
if (C==0)PC =Rssrc & 0xffffffff;

[Description]

JNC causes a jump to the address specified by Rssrc when the condition bit (C) = 0. The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

0001 1101 1100 src JNC Rssrc
Load/store instruction

**LD**

[Mnemonic]

1. \( \text{LD} \ R_{\text{dest}}, R_{\text{src}} \)
2. \( \text{LD} \ R_{\text{dest}}, R_{\text{src}} + \)
3. \( \text{LD} \ R_{\text{dest}}, (\text{disp16}, R_{\text{src}}) \)

[Function]

Load

1. \( R_{\text{dest}} = *\left(\text{signed int}\right) R_{\text{src}}; \)
2. \( R_{\text{dest}} = *\left(\text{signed int}\right) R_{\text{src}}, R_{\text{src}} += 4; \)
3. \( R_{\text{dest}} = *\left(\text{signed int}\right) \left( R_{\text{src}} + (\text{signed short}) \text{disp16} \right); \)

[Description]

1. The contents of the memory at the address specified by \( R_{\text{src}} \) are loaded into \( R_{\text{dest}} \).
2. The contents of the memory at the address specified by \( R_{\text{src}} \) are loaded into \( R_{\text{dest}} \). \( R_{\text{src}} \) is post incremented by 4.
3. The contents of the memory at the address specified by \( R_{\text{src}} \) combined with the 16-bit displacement are loaded into \( R_{\text{dest}} \). The displacement value is sign-extended to 32 bits before the address calculation.

The condition bit (C) does not changed.

[EIT occurrence]

Address exception (AE)

[Encoding]

\[
\begin{align*}
0010 & \text{ dest } 1100 \text{ src } & \text{LD} \ R_{\text{dest}}, \ R_{\text{src}} \\
0010 & \text{ dest } 1110 \text{ src } & \text{LD} \ R_{\text{dest}}, \ R_{\text{src}} + \\
1010 & \text{ dest } 1100 \text{ src } & \text{disp16} \\
\end{align*}
\]

\( \text{LD} \ R_{\text{dest}}, \ (\text{disp16}, R_{\text{src}}) \)
LD24

transfer instruction
Load 24-bit immediate

[Mnemonic]
LD24 Rdest,#imm24

[Function]
Load
Rdest = imm24 & 0x00ffffff;

[Description]
LD24 loads the 24-bit immediate value into Rdest. The immediate value is zero-extended to 32 bits.
The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

1110 dest imm24

LD24 Rdest,#imm24
3.2 Detailed Description of Instructions

LDB

load/store instruction
Load byte

[Mnemonic]

(1) LDB  Rdest,@Rsrc
(2) LDB  Rdest,@(disp16,Rsrc)

[Function]

Load
(1)  Rdest = *( signed char *) Rsrc;
(2)  Rdest = *( signed char *) ( Rsrc + ( signed short ) disp16 );

>Description

(1)  LDB sign-extends the byte data of the memory at the address specified by Rsrc and loads it into Rdest.
(2)  LDB sign-extends the byte data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest.
The displacement value is sign-extended to 32 bits before the address calculation.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

<table>
<thead>
<tr>
<th>0010 dest 1000 src</th>
<th>LDB  Rdest,@Rsrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010 dest 1000 src</td>
<td>displ16</td>
</tr>
</tbody>
</table>

LDB  Rdest,@(disp16,Rsrc)
LDH load/store instruction
Load halfword

[Mnemonic]
(1) LDH Rdest,@Rsrc
(2) LDH Rdest,@(disp16,Rsrc)

[Function]
Load
(1) Rdest = *( signed short *) Rsrc;
(2) Rdest = *( signed short *) ( Rsrc + ( signed short ) disp16 );

[Description]
(1) LDH sign-extends the halfword data of the memory at the address specified by Rsrc and
(2) LDH sign-extends the halfword data of the memory at the address specified by Rsrc combined with the 16-bit
displacement, and loads it into Rdest. The displacement value is sign-extended to 32 bits before the address
calculation.

The condition bit (C) does not changed.

[EIT occurrence]
Address exception (AE)

[Encoding]

<table>
<thead>
<tr>
<th>0010</th>
<th>dest</th>
<th>1010</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDH Rdest,@Rsrc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1010</th>
<th>dest</th>
<th>1010</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>disp16</td>
</tr>
<tr>
<td>LDH Rdest,@(disp16,Rsrc)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.2 Detailed Description of Instructions

**LDI**
transfer instruction
Load immediate

[Mnemonic]

1. **LDI** Rdest,#imm8
2. **LDI** Rdest,#imm16

[Function]
Load

1. Rdest = ( signed char ) imm8;
2. Rdest = ( signed short ) imm16;

[Description]

1. LDI loads the 8-bit immediate value into Rdest. The immediate value is sign-extended to 32 bits.
2. LDI loads the 16-bit immediate value into Rdest. The immediate value is sign-extended to 32 bits.

The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

<table>
<thead>
<tr>
<th>0110</th>
<th>dest</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI</td>
<td>Rdest,#imm8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1001</th>
<th>dest</th>
<th>111</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI</td>
<td>Rdest,#imm16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**INSTRUCTIONS**

### 3.2 Detailed Description of Instructions

#### LDUB

**load/store instruction**

**Load unsigned byte**

**[Mnemonic]**

1. LDUB Rdest,@Rsrc
2. LDUB Rdest,@(disp16,Rsrc)

**[Function]**

Load

1. \( Rdest = *(\text{unsigned char} \ ) \ Rsrc; \)
2. \( Rdest = *(\text{unsigned char} \ ) ( Rsrc + (\text{signed short} \ ) \ \text{disp16}); \)

**[Description]**

1. LDUB zero-extends the byte data from the memory at the address specified by Rsrc and loads it into Rdest.
2. LDUB zero-extends the byte data of the memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest. The displacement value is sign-extended to 32 bits before address calculation.

The condition bit (C) does not changed.

**[EIT occurrence]**

None

**[Encoding]**

<table>
<thead>
<tr>
<th>0010</th>
<th>dest</th>
<th>1001</th>
<th>src</th>
<th>LDUB Rdest,@Rsrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>dest</td>
<td>1001</td>
<td>src</td>
<td>displ16</td>
</tr>
</tbody>
</table>

LDUB Rdest,@(disp16,Rsrc)
**LDUH**

**load/store instruction**

Load unsigned halfword

[Mnemonic]

1. LDUH Rdest,@Rsrc
2. LDUH Rdest,@(disp16,Rsrc)

[Function]

Load

1. \( Rdest = *(\text{unsigned short}) \ Rsrc; \)
2. \( Rdest = *(\text{unsigned short}) (Rsrc + (\text{signed short} \ disp16)); \)

[Description]

1. LDUH zero-extends the halfword data from the memory at the address specified by Rsrc and loads it into Rdest.
2. LDUH zero-extends the halfword data in memory at the address specified by Rsrc combined with the 16-bit displacement, and loads it into Rdest. The displacement value is sign-extended to 32 bits before the address calculation.

The condition bit (C) dose not changed.

[EIT occurrence]

Address exception (AE)

[Encoding]

<table>
<thead>
<tr>
<th>dest</th>
<th>1011</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>LDUH Rdest,@Rsrc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dest</th>
<th>1011</th>
<th>src</th>
<th>disp16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>LDUH Rdest,@(disp16,Rsrc)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.2 Detailed Description of Instructions

**LOCK**

Load/Store Instruction

Load locked

[Mnemonic]

LOCK Rdest,@Rsrc

[Function]

Load locked

LOCK = 1, Rdest = *( signed int *) Rsrc;

[Description]

The contents of the word at the memory location specified by Rsrc are loaded into Rdest.

The condition bit (C) dose not changed.

This instruction sets the LOCK bit in addition to simple loading. When the LOCK bit is 1, DMA transfer request or HOLD request is not accepted.

The LOCK bit is cleared by executing the UNLOCK instruction.

The LOCK bit is internal to the CPU and cannot be accessed directly except by using the LOCK or UNLOCK instructions.

[EIT occurrence]

Address exception (AE)

[Encoding]

```
0010 dest 1101 src  LOCK Rdest,@Rsrc
```
MACHI

DSP function instruction
Multiply-accumulate high-order halfword

[Mnemonic]

MACHI Rsrl, Rsro, Adest

[Function]

Multiply and add

Adest += ( ( signed ) ( Rsrl & 0xffff0000 ) * ( signed short ) ( Rsro >> 16 ) ;

[Description]

MACHI multiplies the 16 high-order bits of Rsrl and the 16 high-order bits of Rsro together and adds the result of multiplication to the 56 low-order bits of accumulator Adest.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 8–15 of Adest are sign-extended before being added. The result of addition is stored in Adest. The 16 high-order bits of Rsrl and the 16 high-order bits of Rsro are handled as signed integers. The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

0011 src1 100 src2 MACHI Rsrl, Rsro, Adest

When accumulator A0 is specified : 0
When accumulator A1 is specified : 1
MACLH1

DSP function instruction
Multiply-accumulate low-order halfword and
high-order halfword using accumulator 1

[Mnemonic]
MACLH1 Rsrc1,Rsrc2

[Function]
Multiply and Add
A1 += ((signed) (Rsrc1 << 16) * (signed short) (Rsrc2 >> 16));

[Description]
MACLH1 multiplies the 16 low-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator A1.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of A1 and those that correspond to bits 8–15 of A1 are sign-extended before being added. The result of addition is stored in A1. The 16 low-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 are handled as signed integers.

A0 does not change as a result of execution of this instruction.
The condition bit (C) does not change.

[Encoding]

0101 src1 1100 src2 MACLH1 Rsrc1,Rsrc2
MACLO

DSP function instruction
Multiply-accumulate low-order halfword

[Mnemonic]

MACLO Rsrc1, Rsrc2, Adest

[Function]

Multiply and Add

Adest += ((signed)(Rsrc1 << 16) * (signed short)Rsrc2);

[Description]

MACLO multiplies the 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator Adest. However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 8–15 of Adest are sign-extended before being added. The result of addition is stored in Adest. The 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 are handled as signed integers. The condition bit (C) does not change.

[Encoding]

0001 src1 101 src2 MACLO Rsrc1, Rsrc2, Adest

When accumulator A0 is specified: 0
When accumulator A1 is specified: 1
MACWHI

DSP function instruction
Multiply-accumulate word and high-order halfword

[Mnemonic]
MACWHI Rscl,Rsrc2

[Function]
Multiply and Add
A0 += (( signed ) Rscl * ( signed short ) ( Rsrc2 >> 16 ));

[Description]
MACWHI multiplies the 32 bits of Rsrc1 and the high-order 16 bits of Rsrc2, then adds the result to the low-order 56 bits in the accumulator. The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 8 through 15 of the accumulator is sign extended before addition. The result of addition is stored in the accumulator. The 32 bits of Rsrc1 and the high-order 16 bits of Rsrc2 are treated as signed values. A1 does not change as a result of execution of this instruction. The condition bit (C) does not changed.

[Encoding]

```
0011 src1 0110 src2 MACWHI Rscl,Rsrc2
```
MACWLO

DSP function instruction
Multiply-accumulate word and low-order halfword

MACWLO

[Mnemonic]

MACWLO Rsrl,Rsrc2

[Function]

Multiply and Add

\[ A0 += ((\text{signed}) \ Rsrc1 \times (\text{signed short}) \ Rsrc2); \]

[Description]

MACWLO multiplies the 32 bits of Rsrl and the low-order 16 bits of Rsrc2, then adds the result to the low-order 56 bits in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 8 through 15 of the accumulator is sign-extended before the addition. The result of the addition is stored in the accumulator. The 32 bits Rsrl and the low-order 16 bits of Rsrc2 are treated as signed values.

A1 does not change as a result of execution of this instruction.

The condition bit (C) dose not changed.

[Diagram]

[Code]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

[Encoding]

0011 src1 0111 src2 MACWLO Rsrl,Rsrc2

None
MACWU1
DSP function instruction
Multiply-accumulate word and unsigned low-order halfword using accumulator 1

[Mnemonic]
MACWU1 Rsrl,Rsrc2

[Function]
Multiply and Add
A1 += ((signed) Rsrc1 * (unsigned short) Rsrc2);

[Description]
MACWU1 multiplies the entire content (32 bits) of Rsrc1 and the 16 low-order bits of Rsrc2 together and adds the result of multiplication to the 56 low-order bits of accumulator A1.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at the least significant bit of A1 and those that correspond to bits 8–15 of A1 are sign-extended before being added. The result of addition is stored in A1. The 32 bits of Rsrc1 are handled as a signed integer and the 16 low-order bits of Rsrc2 are handled as an unsigned integer.

The condition bit (C) does not change.

[Encoding]
0101 src1 1011 src2 MACWU1 Rsrl,Rsrc2
**MSBLO**

DSP function instruction
Multiply low-order halfwords and subtract

**[Mnemonic]**

MSBLO Rsrc1, Rsrc2

**[Function]**

Multiply and Add

A0 -= ((signed) (Rsrc1 << 16) * (signed short) Rsrc2);

**[Description]**

Multiply the 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 together and subtract the result of multiplication from the 56 low-order bits of accumulator A0.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of A0 and those that correspond to bits 8–15 of A0 are sign-extended before subtraction. The result of subtraction is stored in A0. The 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 are handled as signed integers.

A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.

**[EIT occurrence]**

None

**[Encoding]**

<table>
<thead>
<tr>
<th>0101</th>
<th>src1</th>
<th>1101</th>
<th>src2</th>
</tr>
</thead>
</table>

MSBLO Rsrc1, Rsrc2
MUL

multiply and divide instruction

Multiply

[Mnemonic]
MUL Rdest, Rsrc

[Function]

Multiply

{ signed64bit tmp;
  tmp = ( signed64bit ) Rdest * ( signed64bit ) Rsrc;
  Rdest = ( signed int ) tmp;
}

[Description]

MUL multiplies Rdest by Rsrc and puts the result in Rdest. The operands are treated as signed values. The condition bit (C) does not change. The contents of the accumulator are destroyed by this instruction.

[EIT occurrence]

None

[Encoding]

0001 dest 0110 src MUL Rdest, Rsrc
MULHI

DSP function instruction
Multiply high-order halfwords

[Mnemonic]

MULHI Rsrc1, Rsr2, Adest

[Function]

Multiply

Adest = ((signed) (Rsrc1 & 0xffff0000) * (signed short) (Rsrc2 >> 16));

[Description]

MULHI multiplies the 16 high-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 together and stores the result in accumulator Adest.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 0–15 of Adest are sign-extended. Furthermore, the bits 48–63 of Adest are cleared to 0. The 16 high-order bits of Rsrc1 and the 16 high-order bits of Rsrc2 are handled as signed integers.

The condition bit (C) does not change.

[Encoding]

0011 src1 000 src2 MULHI Rsrc1, Rsr2, Adest

When accumulator A0 is specified : 0
When accumulator A1 is specified : 1
MULLO

DSP function instruction
Multiply low-order halfwords

[Mnemonic]

MULLO Rsrl,Rsrc2,Adest

[Function]

Multiply

\[ \text{Adest} = \left( \text{signed} \ right) ( \text{Rsrc1} \ll 16 ) \times \left( \text{signed short} \right) \text{Rsrc2} ; \]

[Description]

MULLO multiplies the 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 together and stores the result in accumulator Adest.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at bit 47 of Adest and those that correspond to bits 0–15 of Adest are sign-extended. Furthermore, the bits 48–63 of Adest are cleared to 0. The 16 low-order bits of Rsrc1 and the 16 low-order bits of Rsrc2 are handled as signed integers.

The condition bit (C) does not change.

[Encoding]

0011 src1 001 src2

MULLO Rsrl,Rsrc2,Adest

When accumulator A0 is specified : 0
When accumulator A1 is specified : 1
MULWHI

DSP function instruction
Multiply word and high-order halfword

[Mnemonic]
MULWHI Rsrc1,Rsrc2

[Function]
Multiply
\[
A0 = \left( (\text{signed}) Rsrc1 \times (\text{signed short}) (Rsrc2 >> 16) \right);
\]

[Description]
MULWHI multiplies the 32 bits of Rsrc1 and the high-order 16 bits of Rsrc2, and stores the result in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 0 through 15 of the accumulator is sign-extended. The 32 bits of Rsrc1 and high-order 16 bits of Rsrc2 are treated as signed values.

A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

```
0011  src1  0010  src2  MULWHI  Rsrc1,Rsrc2
```
MULWLO
DSP function instruction
Multiply word and low-order halfword

[Mnemonic]
MULWLO Rsrc1, Rsrc2

[Function]
Multiply
A0 = ( (signed) Rsrc1 * (signed short) Rsrc2);

[Description]
MULWLO multiplies the 32 bits of Rsrc1 and the low-order 16 bits of Rsrc2, and stores the result in the accumulator.

The LSB of the multiplication result is aligned with the LSB of the accumulator, and the portion corresponding to bits 0 through 15 of the accumulator is sign extended. The 32 bits of Rsrc1 and low-order 16 bits of Rsrc2 are treated as signed values.

A1 does not change as a result of execution of this instruction.
The condition bit (C) does not change.

[Encoding]

0011 src1 0011 src2 MULWLO Rsrc1, Rsrc2

[EIT occurrence]
None
INSTRUCTIONS
3.2 Detailed Description of Instructions

MULWU1
DSP function instruction
Multiply word and unsigned low-order halfword
unsigned accumulator 1

[Mnemonic]
MULWU1 Rsrl,Rsrc2

[Function]
Multiply
A1 = ((signed) Rsrl * (unsigned short) Rsrc2);

[Description]
MULWU1 multiplies the entire content (32 bits) of Rsrl and the 16 low-order bits of Rsrc2 together and stores the result in accumulator A1.

However, the bit position of the multiplication result is adjusted so that its least significant bit is at the least significant bit of A1 and those that correspond to bits 0–15 of A1 are sign-extended. The 32 bits of Rsrl are handled as a signed integer and the 16 low-order bits of Rsrc2 are handled as an unsigned integer.

The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

0101 src1 1010 src2 MULWU1 Rsrl,Rsrc2
**MV**  
**transfer instruction**  
**Move register**  

**[Mnemonic]**  
`MV Rdest, Rsrc`

**[Function]**  
Transfer  
`Rdest = Rsrc;`

**[Description]**  
MV moves Rsrc to Rdest.  
The condition bit (C) dose not changed.

**[EIT occurrence]**  
None

**[Encoding]**  
```
0001  dest 1000  src   MV Rdest, Rsrc
```
**MVFACHI**

DSP function instruction
Move from accumulator high-order word

**[Mnemonic]**

MVFACHI Rdest,Asrc

**[Function]**

Transfer from accumulator to register
Rdest = (signed) (Asrc >> 32);

**[Description]**

MVFACHI moves the high-order 32 bits of the accumulator Asrc to Rdest.
The condition bit (C) dose not changed.

**[EIT occurrence]**

None

**[Encoding]**

```
0101  dest  1111  00  MVFACHI  Rdest,Asrc
```

<table>
<thead>
<tr>
<th>dest</th>
<th>1111</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asrc</td>
<td>When accumulator A0 is specified : 00</td>
<td>When accumulator A1 is specified : 01</td>
</tr>
</tbody>
</table>

---

Rev.0.01  Feb 05,2004  3-71
REJ09B0135-0001Z
### MVFACLO

**DSP function instruction**

**Move from accumulator low-order word**

#### Mnemonic

\[
\text{MVFACLO } R\text{dest}, A\text{src}
\]

#### Function

Transfer from accumulator to register

\[
R\text{dest} = \text{(signed) } A\text{src};
\]

#### Description

MVFACLO moves the low-order 32 bits of the accumulator A\text{src} to R\text{dest}.

The condition bit (C) does not change.

#### EIT occurrence

None

#### Encoding

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
\text{dest} & 1 & 1 & 1 \\
0 & 1
\end{array}
\]

When accumulator A0 is specified: 00
When accumulator A1 is specified: 01
MVFACMI

DSP function instruction
Move middle-order word from accumulator

[Mnemonic]

MVFACMI Rdest,Asrc

[Function]

Transfer from accumulator to register
Rdest = (signed) (Asrc >> 16);

[Description]

MVFACMI moves bits 16 through 47 of the accumulator Asrc to Rdest.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

0101 dest 1111 10 MVFACMI Rdest,Asrc

When accumulator A0 is specified : 00
When accumulator A1 is specified : 01
**MVFC**

transfer instruction
Move from control register

**[Mnemonic]**

MVFC Rdest, CRsrc

**[Function]**

Transfer from control register to register

Rdest = CRsrc;

**[Description]**

MVFC moves CRsrc to Rdest.

The condition bit (C) does not change.

**[EIT occurrence]**

None

**[Encoding]**

<table>
<thead>
<tr>
<th>0001</th>
<th>dest</th>
<th>1001</th>
<th>src</th>
</tr>
</thead>
</table>

MVFC Rdest, CRsrc
MVFCP  Coprocessor support instruction
Move from Coprocessor register

[Mnemonic]

MVFCP  Rdest,CPRsrc,inst,cpid

[Function]

Transfer
Rdest = CPRsrc(num);

[Description]

MVFCP moves the content of CPRsrc register of the coprocessor specified by the coprocessor ID(cpidd) to Rdest.
The bit field “inst” is provided for operation bits passed to the coprocessor. If additional processing needs to be
performed while transferring data to the coprocessor, the necessary direction can be given to the coprocessor by
setting this bit field.
The condition bit (C) does not change.

[EIT occurrence]

Coprocessor interrupt (CPI) or coprocessor disable exception (CDE)

[Encoding]

| 1101 | dest | 0101 | src | cpid | 0000 | inst |

MVFCP  Rdest,CPRsrc,inst,cpid
MVTACHI

DSP function instruction
Move high-order word to accumulator

[Mnemonic]
MVTACHI Rs, A

[Function]
Transfer between accumulator and register
Adest[0:31] = Rs;

[Description]
MVTACHI moves the content of Rs to the 32 high-order bits (bits 0–31) of accumulator Adest.
The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

```
0101 src 0111 00 MVTACHI Rs, A
```

When accumulator A0 is specified: 00
When accumulator A1 is specified: 01
MVTACLO

DSP function instruction
Move low-order word to accumulator

[Mnemonic]
MVTACLO Rsrc, Adest

[Function]
Transfer between accumulator and register
Adest [32 : 63] = Rsrc;

[Description]
MVTACLO moves the content of Rsrc to the 32 low-order bits (bits 32–63) of accumulator Adest.
The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

```
0101 src 0111 01 MVTACLO Rsrc, Adest
```

Adest
When accumulator A0 is specified : 00
 When accumulator A1 is specified : 01
**MVTC**

transfer instruction
Move to control register

**[Mnemonic]**

```
MVTC Rsrcl, CRdest
```

**[Function]**

Transfer from register to control register

```
CRdest = Rsrscl;
```

**[Description]**

MVTC moves Rsrcl to CRdest.

If PSW(CR0) is specified as CRdest, the condition bit (C) is changed; otherwise it dose not changed.

**[EIT occurrence]**

Privilege instruction exception (PIE)

**[Encoding]**

```
  0001  dest  1010  src
```

`MVTC Rsrcl, CRdest`
MVTCP

Coprocessor support instruction
Move to Coprocessor register

[Mnemonic]

MVTCP Rsrc, CPRdest, inst, cpid

[Function]

Transfer
CPRdest(num) = Rsrc;

[Description]

MVTCP moves the content of Rsrc register to CPRdest register of the coprocessor specified by the coprocessor ID(cpid).

The bit field “inst” is provided for operation bits passed to the coprocessor. If additional processing needs to be performed while transferring data to the coprocessor, the necessary direction can be given to the coprocessor by setting this bit field.

The condition bit (C) does not change.

[EIT occurrence]

Coprocessor interrupt (CPI) or coprocessor disable exception (CDE)

[Encoding]

1101 src 0110 dest cpid 0000 inst

MVTCP Rsrc, CPRdest, inst, cpid
3.2 Detailed Description of Instructions

**NEG**

**arithmetic operation instruction**

Negate

**NEG**

[Mnemonic]

NEG Rdest,Rsrc

[Function]

Negate

\[ Rdest = 0 - (\text{signed}) \ Rsrc ; \]

[Description]

NEG negates (changes the sign of) Rsrc treated as a signed 32-bit value, and puts the result in Rdest.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

```
0000 dest 0011 src NEG Rdest,Rsrc
```
**NOP**

branch instruction
No operation

**[Mnemonic]**

NOP

**[Function]**

No operation

/* */

**[Description]**

NOP performs no operation. The subsequent instruction then processed.
The condition bit (C) dose not changed.

**[EIT occurrence]**

None

**[Encoding]**

```
0111 0000 0000 0000 NOP
```
**NOT**

logic operation instruction
Logical NOT

**[Mnemonic]**

\[ \text{NOT \ Rdest, Rsrc} \]

**[Function]**

Logical NOT
\[ \text{Rdest} = \sim \text{Rsrc} ; \]

**[Description]**

NOT inverts each of the bits of Rsrc and puts the result in Rdest.
The condition bit (C) does not change.

**[EIT occurrence]**

None

**[Encoding]**

\[
\begin{array}{c|c|c}
\text{0000} & \text{dest} & \text{1011} & \text{src} \\
\end{array}
\]

\[ \text{NOT \ Rdest, Rsrc} \]
OR

logic operation instruction

[Logic: OR]

OR Rdest, Rsrd

[Function]
Logical OR
Rdest = Rdest | Rsrd;

[Description]
OR computes the logical OR of the corresponding bits of Rdest and Rsrd, and puts the result in Rdest.
The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

0000 dest 1110 src OR Rdest, Rsrd
3.2 Detailed Description of Instructions

**OR3**

logic operation instruction
OR 3-operand

**Mnemonic**

OR3  Rdest,Rsrc,#imm16

**Function**

Logical OR

Rdest = Rsrc | ( unsigned short ) imm16;

**Description**

OR3 computes the logical OR of the corresponding bits of Rsrc and the 16-bit immediate value, which is zero-extended to 32 bits, and puts the result in Rdest.

The condition bit (C) dose not changed.

**EIT occurrence**

None

**Encoding**

<table>
<thead>
<tr>
<th>1000</th>
<th>dest</th>
<th>1110</th>
<th>src</th>
<th>imm16</th>
</tr>
</thead>
</table>

OR3  Rdest,Rsrc,#imm16
OPECP

Coprocessor support instruction
Operate Coprocessor

[Mnemonic]

OPECP CPRdest, CPRsrc, inst, cpid

[Function]

Coprocessor operation
CPRdest = inst(CPRdest, CPRsrc)(cpid);

[Description]

OPECP executes the coprocessor instruction specified by "inst" to the coprocessor is specified by coprocessor ID(cpid) and moves the result to CPRdest.
The condition bit (C) does not change.

[EIT occurrence]

Coprocessor interrupt (CPI) or coprocessor disable exception (CDE)

[Encoding]

```
1101 CPdest 0111 CPsrc cpid 0000 inst
```

OPECP CPRdest, CPRsrc, inst, cpid
3.2 Detailed Description of Instructions

**PCMPBZ**

Parallel compare byte to zero

[Mnemonic]

PCMPBZ Rs

[Function]

Compare

\[ C = ((\text{Rs}[0:7] == 0) \lor (\text{Rs}[8:15] == 0) \lor (\text{Rs}[16:23] == 0) \lor (\text{Rs}[24:31] == 0)) ? 1 : 0 \]

[Description]

Rs is assumed to be consisting of four packed 8-bit data. When one of these four 8-bit data \( = 0 \), the condition bit \( C \) is set to 1.

[EIT occurrence]

None

[Encoding]

```
0000 0011 0111 src PCMPBZ Rs
```
### RAC

**DSP function instruction**
Round accumulator

#### [Mnemonic]

RAC Adest,Asrc,#imm1

#### [Function]

Round

```cpp
( signed64bit tmp;
  tmp = ( signed64bit )Asrc << imm1;
  tmp = tmp + 0x0000 0000 0000 8000;
  if ( tmp > ( signed64bit ) 0x0000 7fff ffff 0000 )
    Adest = 0x0000 7fff ffff 0000;
  else if ( tmp < ( signed64bit ) 0xffff 8000 0000 0000 )
    Adest = 0xffff 8000 0000 0000;
  else
    Adest = tmp & 0xffff ffff ffff 0000;
}
( imm1 = 1, 2; )
```

#### [Description]

RAC rounds the contents in the accumulator to word size and stores the result in the accumulator.
The condition bit (C) does not change.

#### [EIT occurrence]

None

#### [Encoding]

```
0101  00 1001  0   RAC Adest,Asrc,#imm1

imm1
  When value 1 is specified : 0
  When value 2 is specified : 1

Asrc
  When accumulator A0 is specified : 00
  When accumulator A1 is specified : 01

Adest
  When accumulator A0 is specified : 00
  When accumulator A1 is specified : 01
```
[Supplementary explanation]

The RAC instruction is executed following a procedure similar to the one described below.

- **Procedure 1**
  - Sign-extended virtual bits 0–7
  - Shifted 1 or 2 bits to the left

- **Procedure 2**
  - The accumulator value changes according to a 64-bit value consisting of virtual bits 0–7 in which the 1 or 2-bit shift is reflected plus the left-shifted bits 8–63.
RACH

DSP function instruction
Round accumulator halfword

[Mnemonic]

RACH Adest, Asrc, #imm1

[Function]

Round

( signed64bit tmp:
    tmp = ( signed64bit )Asrc << imm1;
    tmp = tmp + 0x0000 0000 8000 0000;
    if( tmp > ( signed64bit ) 0x0000 7fff 0000 0000 )
        Adest = 0x0000 7fff 0000 0000;
    else if ( tmp < ( signed64bit ) 0xffff 8000 0000 0000 )
        Adest = 0xffff 8000 0000 0000;
    else
        Adest = tmp & 0xffff ffff 0000 0000;
)

( imm1 = 1, 2; )

[Description]

RAC rounds the accumulator value to a halfword size and store the result in the accumulator.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

<table>
<thead>
<tr>
<th>0101</th>
<th>00</th>
<th>1000</th>
<th>01</th>
</tr>
</thead>
</table>

imm1

When value 1 is specified : 0
When value 2 is specified : 1

Asrc

When accumulator A0 is specified : 00
When accumulator A1 is specified : 01

Adest

When accumulator A0 is specified : 00
When accumulator A1 is specified : 01
[Supplementary explanation]

The RACH instruction is executed following a procedure similar to the one described below.

- **Procedure 1**

  Sign-extended virtual bits 0–7

  Shifted 1 or 2 bits to the left

- **Procedure 2**

  The accumulator value changes according to a 64-bit value consisting of the left-shifted bits 8–63 plus virtual bits 0–7 in which the value of 1 or 2 shifted out bits is reflected.
REM multiply and divide instruction

[Description]

REM divides Rdest by Rsrc and stores the remainder in Rdest. The operands are treated as signed 32-bit values. The quotient is rounded toward zero and the remainder takes the same sign as the dividend. The condition bit (C) does not change. When Rsrc is zero, the value of Rdest does not change.

[Encoding]

```
1001 dest 0010 src 0000 0000 0000 0000
REM Rdest,Rsrc
```
**REMB**

multiply and divide instruction

Remainder byte

**[Mnemonic]**

REMB Rdest,Rsrc

**[Function]**

Signed remainder

Rdest = (signed char)Rdest % (signed)Rsrc ;

**[Description]**

REMB divides Rdest by Rsr and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as a signed 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as a signed 32-bit value. The quotient is rounded toward 0, and the remainder takes the same sign as the divisor.

The condition bit (C) does not change.

When Rsr is zero, the value of Rdest does not change.

**[EIT occurrence]**

None

**[Encoding]**

```
1001 dest 0010 src 0000 0000 0001 1000

REMB Rdest,Rsrc
```
**REMH**

Multiply and divide instruction  
Remainder halfword

[Mnemonic]
REMH Rdest,Rsrc

[Function]
Signed remainder  
Rdest = (signed short)Rdest \%(signed)Rsrc ;

[Description]
REMH divides Rdest by Rsrc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as a signed 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as a signed 32-bit value. The quotient is rounded toward 0, and the remainder takes the same sign as the divisor.

The condition bit (C) does not change.
When Rsrc is zero, the value of Rdest does not change.

[EIT occurrence]
None

[Encoding]

<table>
<thead>
<tr>
<th>1001</th>
<th>dest</th>
<th>0010</th>
<th>src</th>
<th>0000</th>
<th>0000</th>
<th>0001</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

REMH Rdest, Rsrc
INSTRUCTIONS
3.2 Detailed Description of Instructions

REMU multiply and divide instruction
Remainder unsigned

[Mnemonic]
REMU Rdest, Rsrc

[Function]
Unsigned remainder
Rdest = (unsigned) Rdest % (unsigned) Rsrc;

>Description]
REMU divides Rdest by Rsrc and stores the remainder in Rdest. The operands are treated as unsigned 32-bit values.
The condition bit (C) does not changed.
When Rsrc is zero, Rdest does not changed.

[EIT occurrence]
None

[Encoding]

<table>
<thead>
<tr>
<th>1001</th>
<th>dest</th>
<th>0011</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

REMU Rdest, Rsrc
**REMUB** multiply and divide instruction

Remainder unsigned byte

**[Mnemonic]**

REMUB Rdest, Rs rc

**[Function]**

Unsigned remainder

\[ \text{Rdest} = (\text{unsigned char})\text{Rdest} \mod (\text{unsigned})\text{Rsrc} ; \]

**[Description]**

REMUB divides Rdest by Rs rc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as an unsigned 8-bit value, with the 24 high-order bits (bits 0–23) ignored. The divisor is handled as an unsigned 32-bit value.

The condition bit (C) does not change.

When Rs rc is zero, the value of Rdest does not change.

**[EIT occurrence]**

None

**[Encoding]**

<table>
<thead>
<tr>
<th>1001</th>
<th>dest</th>
<th>0011</th>
<th>src</th>
<th>0000</th>
<th>0000</th>
<th>0001</th>
<th>1000</th>
</tr>
</thead>
</table>

REMUB Rdest, Rs rc
**REMUH**

**multiply and divide instruction**

**Remainder unsigned halfword**

**[Mnemonic]**

\[ \text{REMUH } \text{Rdest,Rsrc} \]

**[Function]**

Unsigned remainder

\[ \text{Rdest } = \text{(unsigned short)} \text{Rdest } \% \text{(unsigned)} \text{Rsrc} ; \]

**[Description]**

REMUH divides Rdest by Rsrc and stores the remainder in Rdest. Of the operands of this instruction, the dividend is handled as an unsigned 16-bit value, with the 16 high-order bits (bits 0–15) ignored. The divisor is handled as an unsigned 32-bit value.

- The condition bit (C) does not change.
- When Rsrc is zero, the value of Rdest does not change.

**[EIT occurrence]**

None

**[Encoding]**

\[
\begin{array}{cccccccc}
1001 & \text{dest} & 0011 & \text{src} & 0000 & 0000 & 0001 & 0000 \\
\end{array}
\]

**REMUH Rdest,Rsrc**
3.2 Detailed Description of Instructions

### RTE

#### Mnemonic

RTE

#### Function

Return from EIT handler

- SM = BSM ;
- IE = BIE ;
- PM = BPM
- CE = BCE
- C = BC ;
- PC = BPC & 0xffffffff ;

#### Description

Restore the SM, IE, PM, CE and C bits of the PSW register from the respective backup bits BSM, BIE, BPM, BCE and C, and branch to the address indicated by BPC.

#### EIT occurrence

Privilege instruction exception (PIE)

#### Encoding

```
0001  0000  1101  0110  RTE
```
**SADD**  
DSP function instruction  
Add accumulators

**Mnemonic**

SADD

**Function**

Add

\[ A0 = ((\text{signed}) A0 + (\text{signed}) ((\text{signed}) A1 >> 16)); \]

**Description**

SADD adds accumulator A0 and accumulator A1 that have been arithmetically shifted 16 bits right and stores the result in A0.

The values of A0 and A1 that have been shifted 16 bits right are handled as signed integers.

The accumulator A1 does not change as a result of execution of this instruction.

The condition bit (C) does not change.

**EIT occurrence**

None

**Encoding**

0101 0000 1110 0100  SADD
SATB
DSP function instruction
Saturate word into Byte

[Mnemonic]
SATB Rdest, Rscc

[Function]
Saturation processing
{
if ((signed char) 0x7f <= (signed) Rscc)
   Rdest = 0x0000007f;
else if ((signed) Rscc <= (signed char) 0x80;)
   Rdest = 0xffffffff80;
else
   Rdest = Rscc;
};

[Description]
SATB rounds the value of Rscc to a byte size (saturation processing) and stores the result in Rdest.
The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

```
1000 dest 0110 src 0000 0011 0000 0000
SATB Rdest,Rsrc
```
[Supplementary explanation]

The value of Rdest changes according to the value of Rsrc.

![Diagram showing the relationship between Rsrc and Rdest values]

- Positive value of Rsrc:
  - \( 0000 \text{ to } 007F \)
  - \( 0000 \text{ to } 007E \)
  - \( 0000 \text{ to } 0000 \)
  - \( FFFF \text{ to } FF81 \)
  - \( FFFF \text{ to } FF80 \)
  - \( 8000 \text{ to } 0000 \)

- Negative value of Rsrc:
  - \( 0000 \text{ to } 0000 \)
  - \( FF \text{ to } FF \)
  - \( FF \text{ to } FF \)
  - \( 80 \text{ to } 80 \)
SATH

DSP function instruction
Saturate word into Half-word

[Mnemonic]
SATH Rdest, RsrC

[Function]
Saturation processing
{
if ((signed short) 0x7fff <= (signed) RsrC)
Rdest = 0x00007fff;
else if ((signed) RsrC <= (signed short) 0x8000)
Rdest = 0xffff8000;
else
Rdest = RsrC;
}

[Description]
SATH rounds the value of RsrC to a halfword size (saturation processing) and stores the result in Rdest.
The condition bit (C) does not change.

[EIT occurrence]
None

[Encoding]

| 1000 dest 0110 src | 0000 0010 0000 0000 |

SATH Rdest, RsrC
[Supplementary explanation]

The value of Rdest changes according to the value of Rsrc.
SC
branch
Skip on C-bit

[Mnemonic]
SC

[Function]
Conditional skip
if (C == 1) Cancel parallel execution of the next 16-bit instruction ;

[Description]
When the condition bit (C) = 1, cancel the 16-bit instruction to be executed at the same time and skip to the next instruction.

This instruction is used for conditional execution of another instruction to be executed in parallel with it (executed when C = 0), and is effective for only parallel instruction execution.

However, a combination of 16-bit instructions that can be executed simultaneously with the SC instruction is both-side instructions (OS) and right-side instructions (-S).

[EIT occurrence]
None

[Encoding]

0111 0100 0000 0001 SC
SETPSW

[Mnemonic]

SETPSW #imm8

[Function]

Set SM, IE, PM, CE or C bit in the PSW to 1.

PSW |= (unsigned char ) imm8;

[Description]

Logically OR the 8-bit value specified by imm8 with the 8 low-order bits in the PSW (bits 24–31) bit wise and write the result to the 8 low-order bits in the PSW bit by bit.

Make sure that all of the unsupported PWS bits in the microcomputer used are set to 0.

[EIT occurrence]

Privilege instruction exception(PIE)

[Encoding]

0111 0001 imm8

SETPSW #imm8
**SETH**

**Transfer Instruction**
Set high-order 16-bit

**Mnemonic**

\[
\text{SETH} \quad \text{Rdest, #imm16}
\]

**Function**

Transfer instruction

\[
\text{Rdest} = (\text{signed short}) \text{imm16} \ll 16;
\]

**Description**

SETH loads the immediate value into the 16 most significant bits of Rdest. The 16 least significant bits become zero. The condition bit (C) does not change.

**EIT occurrence**

None

**Encoding**

<table>
<thead>
<tr>
<th>1101</th>
<th>dest</th>
<th>1100</th>
<th>0000</th>
<th>imm16</th>
</tr>
</thead>
</table>

SETH Rdest, #imm16
SLL

**shift instruction**
Shift left logical

**Mnemonic**

SLL Rdest, Rsrc

**Function**

Logical left shift

Rdest = Rdest << ( Rsrc & 31 ) ;

**Description**

SLL left logical-shifts the contents of Rdest by the number specified by Rsrc, shifting zeroes into the least significant bits.

Only the five least significant bits of Rsrc are used.

The condition bit (C) dose not changed.

**EIT occurrence**

None

**Encoding**

```
0001  dest  0100  src  SLL Rdest, Rsrc
```
SLL3

shift instruction
Shift left logical 3-operand

[Mnemonic]
SLL3  Rdest,Rsrc,#imm16

[Function]
Logical left shift
Rdest = Rsrc << ( imm16 & 31 );

[Description]
SLL3 left logical-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, shifting zeroes into the least significant bits.
Only the five least significant bits of the 16-bit immediate value are used.
The condition bit (C) dose not changed.

[EIT occurrence]
None

[Encoding]

\[
\begin{array}{ccc}
1001 & \text{dest} & 1100 & \text{src} \\
\end{array}
\]

SLL3  Rdest,Rsrc,#imm16
### SLLI

**shift instruction**  
Shift left logical immediate

#### [Mnemonic]

SLLI Rdest,#imm5

#### [Function]

Logical left shift  
Rdest = Rdest << imm5;

#### [Description]

SLLI left logical-shifts the contents of Rdest by the number specified by the 5-bit immediate value, shifting zeroes into the least significant bits.  
The condition bit (C) dose not changed.

#### [EIT occurrence]

None

#### [Encoding]

```
0101 dest 010 imm5  SLLI Rdest,#imm5
```
**SNC**

[Mnemonic]

SNC

[Function]

Conditional skip

if (C == 0) Cancel parallel execution of the next 16-bit instruction;

[Description]

When the condition bit (C) = 0, cancel the 16-bit instruction to be executed at the same time and skip to the next instruction.

This instruction is used for conditional execution of another instruction to be executed in parallel with it (executed when C = 1), and is effective for only parallel instruction execution.

However, A combination of 16-bit instructions that can be executed simultaneously with the SNC instruction is both-side instructions (OS) and right-side instructions (-S).

[EIT occurrence]

None

[Encoding]

```
0111 0101 0000 0001 SNC
```
3.2 Detailed Description of Instructions

**SRA**

shift instruction
Shift right arithmetic

[Mnemonic]

SRA  Rdest, Rsrc

[Function]

Arithmetic right shift
Rdest = (signed) Rdest >> (Rsrc & 31);

[Description]

SRA right arithmetic-shifts the contents of Rdest by the number specified by Rsrc, replicates the sign bit in the MSB of Rdest and puts the result in Rdest.
Only the five least significant bits are used.
The condition bit (C) dose not changed.

[EIT occurrence]

None

[Encoding]

```
0001 dest 0010 src SRA Rdest, Rsrc
```
SRA3
shift instruction
Shift right arithmetic 3-operand

[Mnemonic]

SRA3 Rdest, Rsrc, #imm16

[Function]

Arithmetic right shift

Rdest = (signed) Rsrc >> (imm16 & 31);

[Description]

SRA3 right arithmetic-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, replicates the sign bit in Rsrc and puts the result in Rdest.

Only the five least significant bits are used.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

```
1001 dest 1010 src imm16
```

SRA3 Rdest, Rsrc, #imm16
SRAI

shift instruction
Shift right arithmetic immediate

[Mnemonic]

SRAI Rdest,#imm5

[Function]

Logical right shift

Rdest = (signed) Rdest >> imm5;

[Description]

SRAI right arithmetic-shifts the contents of Rdest by the number specified by the 5-bit immediate value, replicates the sign bit in MSB of Rdest and puts the result in Rdest.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

\[
\begin{array}{ccc}
0101 & \text{dest} & 001 & \text{imm5} \\
\end{array}
\]

SRAI Rdest,#imm5
SRL

[Mnemonic]

SRA Rdest, Rsra

[Function]

Logical right shift

Rdest = (unsigned) Rdest >> (Rsrc & 31);

[Description]

SRL right logical-shifts the contents of Rdest by the number specified by Rsrc, shifts zeroes into the most significant bits and puts the result in Rdest.

Only the five least significant bits of Rsrc are used.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

<table>
<thead>
<tr>
<th>0001</th>
<th>dest</th>
<th>0000</th>
<th>src</th>
</tr>
</thead>
</table>

SRL Rdest, Rsrc
SRL3  
Shift right logical 3-operand

[Mnemonic]

SRL3 Rdest,Rsrc,#imm16

[Function]

Logical right shift

Rdest = (unsigned) Rsrc >> (imm16 & 31);

[Description]

SRL3 right logical-shifts the contents of Rsrc into Rdest by the number specified by the 16-bit immediate value, shifts zeroes into the most significant bits. Only the five least significant bits of the immediate value are valid.
The condition bit (C) does not changed.

[EIT occurrence]

None

[Encoding]

```
  1001    dest 1000    src  imm16
```

SRL3 Rdest,Rsrc,#imm16
**SRLI**

shift instruction
Shift right logical immediate

[Mnemonic]

\[ \text{SRLI } \text{Rdest}, \#\text{imm5} \]

[Function]

Logical right shift
\[
\text{Rdest} = (\text{unsigned}) \text{Rdest} \gg (\text{imm5} \& 31);
\]

[Description]

SRLI right arithmetic-shifts Rdest by the number specified by the 5-bit immediate value, shifting zeroes into the most significant bits.

The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

\[
\begin{array}{c|c|c|c}
0101 & \text{dest} & 000 & \text{imm5} \\
\end{array}
\]

SRLI Rdest, #imm5
**ST**

load/store instruction

**ST**

[Mnemonic]

(1) \( ST \ Rsrc1, @Rsrc2 \)
(2) \( ST \ Rsrc1, @+Rsrc2 \)
(3) \( ST \ Rsrc1, @-Rsrc2 \)
(4) \( ST \ Rsrc1, @(disp16, Rsrc2) \)

[Function]

Store

(1) \( * (\text{signed int} \*) \ Rsrc2 = Rsrc1; \)
(2) \( \ Rsrc2 += 4, * (\text{signed int} \*) \ Rsrc2 = Rsrc1; \)
(3) \( \ Rsrc2 -= 4, * (\text{signed int} \*) \ Rsrc2 = Rsrc1; \)
(4) \( * (\text{signed int} \*) \ (\ Rsrc2 + (\text{signed short}) \ disp16 ) = Rsrc1; \)

[Description]

(1) \( ST \) stores \( Rsrc1 \) in the memory at the address specified by \( Rsrc2 \).
(2) \( ST \) increments \( Rsrc2 \) by 4 and stores \( Rsrc1 \) in the memory at the address specified by the resultant \( Rsrc2 \).
(3) \( ST \) decrements \( Rsrc2 \) by 4 and stores the contents of \( Rsrc1 \) in the memory at the address specified by the resultant \( Rsrc2 \).
(4) \( ST \) stores \( Rsrc1 \) in the memory at the address specified by \( Rsrc \) combined with the 16-bit displacement. The displacement value is sign-extended before the address calculation.

The condition bit (C) dose not changed.

[EIT occurrence]

Address exception (AE)
### Encoding

<table>
<thead>
<tr>
<th></th>
<th>src1</th>
<th>src2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0100</td>
<td>src2</td>
<td>ST Rscl, @Rscl2</td>
</tr>
<tr>
<td>0010</td>
<td>0110</td>
<td>src2</td>
<td>ST Rscl, @+Rscl2</td>
</tr>
<tr>
<td>0010</td>
<td>0111</td>
<td>src2</td>
<td>ST Rscl, @-Rscl2</td>
</tr>
<tr>
<td>1010</td>
<td>0100</td>
<td>src2</td>
<td>ST Rscl, @(disp16, Rscl2)</td>
</tr>
</tbody>
</table>

ST Rscl, @(disp16, Rscl2)
3.2 Detailed Description of Instructions

**STB**

load/store instruction

Store byte

**[Mnemonic]**

1. STB Rsrl, @Rsrc2
2. STB Rsrl, @Rsrc2 +
3. STB Rsrl, @(disp16, Rsrc2)

**[Function]**

Store

1. *(signed char *)Rsrc2 = Rsrc1;
2. *(signed char *)Rsrc2 = Rsrc1, Rsrc2 += 1;
3. *(signed char *)(Rsrc2 + (signed short)disp16) = Rsrc1;

**[Description]**

1. STB stores the byte data on the LSB side of Rsrc in a memory location whose address is specified by Rdest.
2. STB stores the byte data on the LSB side of Rsrc in a memory location whose address is specified by Rdest, and then increment Rdest by 1.
3. STB stores the byte data on the LSB side of Rsrc1 in a memory location whose address is specified by Rdest and a 16-bit displacement. The displacement is sign-extended before address calculation.

The condition bit (C) does not change.

**[EIT occurrence]**

None

**[Encoding]**

<table>
<thead>
<tr>
<th>0010</th>
<th>src1</th>
<th>0000</th>
<th>src2</th>
<th>STB Rsrl, @Rsrc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>src1</td>
<td>0001</td>
<td>src2</td>
<td>STB Rsrl, @Rsrc2 +</td>
</tr>
<tr>
<td>1010</td>
<td>src1</td>
<td>0000</td>
<td>src2</td>
<td>disp16</td>
</tr>
</tbody>
</table>

STB Rsrl, @(disp16, Rsrc2)
3.2 Detailed Description of Instructions

**STH**

**Mnemonic**

1. `STH Rsrc1, @Rsrc2`
2. `STH Rsrc1, @Rsrc2+`
3. `STH Rsrc1, @(disp16, Rsrc2)`

**Function**

- **Store**
  - (1) `*(signed short *)Rsrc2 = Rsrc1;`
  - (2) `*(signed short *)Rsrc2 = Rsrc1, Rsrc2 += 2;`
  - (3) `*(signed short *)(Rsrc2 + (signed short)disp16) = Rsrc1;`

**Description**

1. STH stores the halfword data on the LSB side of Rsrc in a memory location whose address is specified by Rdest.
2. STH stores the halfword data on the LSB side of Rsrc in a memory location whose address is specified by Rdest, and then increment Rdest by 2.
3. STH stores the halfword data on the LSB side of Rsrc in a memory location whose address is specified by Rdest and a 16-bit displacement. The displacement is sign-extended before address calculation.

   The condition bit (C) does not change.

**EIT occurrence**

Address exception (AE)

**Encoding**

```
0010 src1 0010 src2   STH Rsrc1, @Rsrc2
0010 src1 0011 src2   STH Rsrc1, @Rsrc2+
1010 src1 0010 src2   disp16   STH Rsrc1, @(disp16, Rsrc2)
```
3.2 Detailed Description of Instructions

**SUB**

*arithmetic operation instruction*

Subtract

**[Mnemonic]**

```
SUB Rdest, Rsrc
```

**[Function]**

Subtract

```
Rdest = Rdest - Rsrc;
```

**[Description]**

SUB subtracts Rsrc from Rdest and puts the result in Rdest.

The condition bit (C) does not change.

**[EIT occurrence]**

None

**[Encoding]**

```
0000 dest 0010 src SUB Rdest, Rsrc
```
3.2 Detailed Description of Instructions

**SUBV**

**arithmetic operation instruction**

**Subtract with overflow checking**

[**Mnemonic**]

```plaintext
SUBV Rdest, Rsrc
```

[**Function**]

```
Subtract
Rdest = ( signed ) Rdest - ( signed ) Rsrc;
C = overflow ? 1 : 0;
```

[**Description**]

SUBV subtracts Rsrc from Rdest and puts the result in Rdest.

The condition bit (C) is set when the subtraction results in overflow; otherwise, it is cleared.

[**EIT occurrence**]

None

[**Encoding**]

```
0000  dest  0000  src
```

SUBV Rdest, Rsrc
SUBX

arithmetic operation instruction
Subtract with borrow

[Mnemonic]

SUBX Rdest,Rsrc

[Function]

Subtract
Rdest = (unsigned) Rdest - (unsigned) Rsrc - C;
C = borrow ? 1 : 0;

[Description]

SUBX subtracts Rsrc and C from Rdest and puts the result in Rdest.
The condition bit (C) is set when the subtraction result cannot be represented by a 32-bit unsigned integer;
otherwise it is cleared.

[EIT occurrence]

None

[Encoding]

0000 dest 0001 src SUBX Rdest,Rsrc
3.2 Detailed Description of Instructions

**TRAP**

**EIT-related instruction**

**Trap**

**[Mnemonic]**

TRAP #imm4

**[Function]**

Generate TRAP

- BPC = NextPC; (NextPC denotes the PC of the next instruction)
- BSM = SM;
- BIE = IE;
- BPM = PM
- BCE = CE
- BC = C;
- IE = 0;
- PM = 0;
- CE = 0
- C = 0;
- call_trap_handler(imm4);

**[Description]**

Generate a trap of the specified number.

The values of the SM, IE, PM, CE and C bits in the PSW register are saved to the respective backup bits BSM, BIE, BPM, BCE and BC, and the IE, PM, CE and C bits each are updated to 0.

**[EIT occurrence]**

Trap (TRAP)

**[Encoding]**

```
0001 0000 1111 imm4 TRAP #imm4
```
UNLOCK

load/store instruction
Store unlocked

[Mnemonic]
UNLOCK Rsrc1,@Rsrc2

[Function]
Store unlocked
if ( LOCK == 1 ) { *( signed int *) Rsrc2 = Rsrc1; }
LOCK = 0;

[Description]
When the LOCK bit is 1, the contents of Rsrc1 are stored at the memory location specified by Rsrc2. When the
LOCK bit is 0, store operation is not executed. The condition bit (C) dose not changed. This instruction clears the
LOCK bit to 0 in addition to the simple storage operation.
The LOCK bit is internal to the CPU and cannot be accessed accepts by using the LOCK and UNLOCK
instructions.

[EIT occurrence]
Address exception (AE)

[Encoding]

0010 src1 0101 src2 UNLOCK Rsrc1,@Rsrc2
XOR

logic operation instruction
Exclusive OR

[Mnemonic]

XOR Rdest, Rsrc

[Function]

Exclusive OR
Rdest = Rdest ^ Rsrc;

[Description]

XOR computes the logical XOR of the corresponding bits of Rdest and Rsrc, and puts the result in Rdest.
The condition bit (C) does not change.

[EIT occurrence]

None

[Encoding]

<table>
<thead>
<tr>
<th>0000</th>
<th>dest</th>
<th>1101</th>
<th>src</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR Rdest, Rsrc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**XOR3**

logic operation instruction
Exclusive OR 3-operand

---

**[Mnemonic]**

XOR3 Rdest, Rsrc, #imm16

**[Function]**

Exclusive OR

\[ Rdest = Rsrc \oplus (\text{unsigned short} ) \text{imm16}; \]

**[Description]**

XOR3 computes the logical XOR of the corresponding bits of Rsrc and the 16-bit immediate value, which is zero-extended to 32 bits, and puts the result in Rdest.

The condition bit (C) does not changed.

**[EIT occurrence]**

None

**[Encoding]**

```
 1000  dest 1101  src   imm16
```

XOR3 Rdest, Rsrc, #imm16
3.3 Notes about the BCL and BNCL Instructions

If the BCL or BNCL instruction is located at a word boundary and the 16-bit instruction in the latter half of the word boundary is a sequentially executed instruction, it depends on the value of the C bit whether the latter half 16-bit instruction is executed. Therefore, be especially careful when the BCL or BNCL instruction is located at a word boundary and the instruction is followed by a 16-bit instruction in the latter half of the word boundary.

If instruction codes are located in the manner shown below and BCL (or BNCL) branches off upon C bit = 1, the latter half 16-bit instruction is not executed. This is because the jump addresses in the OPSP-CPU should always be aligned with word boundaries (except when the RTE instruction is executed). If C bit = 0 and BCL (or BNCL) does not branch, the latter half 16-bit instruction is executed.

```
0  BCL or BNCL  0  xxxx instruction
```

It depends on the value of the C bit whether the xxxx instruction is executed.

On the other hand, if the latter half 16-bit instruction (yyyy instruction) is a parallel executed instruction, the yyyy instruction is executed at the same time BCL (or BNCL) is executed.

```
0  BCL or BNCL  1  yyyy instruction
```

The yyyy instruction is executed in parallel with BCL (or BNCL).
3.4 Exception and Trap Handling during Parallel Instruction Execution

During parallel instruction execution, exceptions and traps are handled in the manner shown below.

Table 3.4.1 Exception and trap handling during parallel instruction execution

<table>
<thead>
<tr>
<th>O pipe (left side) instruction</th>
<th>S pipe (right side) instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIE</td>
<td>Any instruction</td>
<td>RIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>RIE</td>
<td>RIE</td>
<td>RIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>Any instruction</td>
<td>RIE</td>
<td>RIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>PIE</td>
<td>Any instruction</td>
<td>PIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>PIE</td>
<td>RIE</td>
<td>RIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>AE</td>
<td>Any instruction</td>
<td>AE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>AE</td>
<td>RIE</td>
<td>RIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
<tr>
<td>TRAP</td>
<td>Any instruction</td>
<td>TRAP occurs. Instructions on both the left and right sides are executed.</td>
</tr>
<tr>
<td>TRAP</td>
<td>RIE</td>
<td>RIE occurs. Instruction on neither the left side nor the right side is executed.</td>
</tr>
</tbody>
</table>
This page is blank for reasons of layout.
Appendix 1 Mechanism of Pipelined Instruction Processing

Appendix 1.1 Outline of Pipelined Instruction Processing

The OPSP CPU core has two pipelines (O pipe and S pipe). These two pipelines each consist of five pipeline stages. Parallel instruction execution by the OPSP CPU core is accomplished by using these two pipelines at the same time. (For details about combinatorial instructions that can be executed in parallel at the same time, refer to Section 2.5, “Parallel Instruction Execution.”)

- Operation of the O pipeline and outline of each stage

(1) IF stage (instruction fetch stage)
   This is the stage in which the CPU fetches instructions. Instructions are fetched from memory (cache).
   The OPSP CPU has an instruction queue, so that it continues fetching instructions until the instruction queue is filled, regardless of whether decode processing in the D (decode) stage has finished.

(2) D stage (decode stage)
   In the D stage, the CPU decodes instructions (DEC). At this time, the CPU reads out a register (RF) and if the result of the immediately preceding instruction needs to be referenced, it performs bypass processing (BYP). However, the bypass processing is performed only when the immediately preceding instruction is a register-to-register transfer or arithmetic operation instruction or a DSP function instruction.

(3) E stage (execution stage)
   In this stage, the CPU performs arithmetic operation or address calculation (OP).

(4) MEM stage (memory access stage)
   In this stage, the CPU accesses the operand (OA). This stage is used only when executing load/store instructions.

(5) WB stage (write-back stage)
   In this stage, the CPU writes the operation result or fetched data to a register.

Appendix Figure 1.1 shows a structure of the O pipeline and the operation performed in it.
Pipeline stages:

- IF stage
- D stage
- E stage
- MEM stage
- WB stage

Executed processing:

- IF
- DEC
- OP
- OA
- RF/BYP
- WB

1 cycle

IF : Instruction fetch processing
DEC : Instruction decode processing
RF : Register fetch processing
OP : Operation
BYP : Bypass processing
OA : Operand access
WB : Write-back processing

Appendix Figure 1.1 Structure of the O Pipeline and the Operation Performed in It
Operation of the S pipeline and outline of each stage

(1) IF stage (instruction fetch stage)
This is the stage in which the CPU fetches instructions. Instructions are fetched from memory (cache). The OPSP CPU has an instruction queue, so that it continues fetching instructions until the instruction queue is filled, regardless of whether decode processing in the D (decode) stage has finished.

(2) D stage (decode stage)
In the D stage, the CPU decodes instructions (DEC). At this time, the CPU reads out a register (RF) and if the result of the immediately preceding instruction needs to be referenced, it performs bypass processing (BYP). However, the bypass processing is performed only when the immediately preceding instruction is a register-to-register transfer or arithmetic operation instruction or a DSP function instruction.

(3) E1 stage (execution stage 1)
In this stage, the CPU performs arithmetic operation or transfers data to registers or accumulators (OP1).

(4) E2 stage (execution stage 2)
This stage is used for DSP function instructions that write data to accumulators (OP2). In this case, the CPU requires two execution cycles because two execution stages (E1 and E2) are used. This stage is not used for other instructions that do not write operation results to accumulators. These instructions are sent directly to the next WB stage.

(5) WB stage (write-back stage)
Operation results are written to registers or accumulators.

Appendix Figure 1.2 shows a structure of the S pipeline and the operation performed in it.

---

**Pipeline stages**

<table>
<thead>
<tr>
<th>IF stage</th>
<th>D stage</th>
<th>E1 stage</th>
<th>E2 stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DEC</td>
<td>OP1</td>
<td>OP2</td>
<td>WB</td>
</tr>
</tbody>
</table>

- **IF**: Instruction fetch processing
- **DEC**: Instruction decode processing
- **RF**: Register fetch processing
- **OP1**: Operation 1
- **OP2**: Operation 2
- **BYP**: Bypass processing
- **WB**: Write-back processing

Appendix Figure 1.2 Structure of the S Pipeline and the Operation Performed in It
Appendix 1.2 Flow of Instruction Processing in the O and S Pipes

The IF and D stages in the O and S pipes are shared between the two pipelines, so that all instructions are processed in common at up to the D stage. Which pipeline each particular instruction should be forwarded to, is determined in the D stage.

Instructions to be executed in parallel are forwarded to both the O and S pipes. Other instructions are forwarded to either the O or the S pipe and processed separately at the E and subsequent stages.

A flow of instruction processing in the O and S pipes is shown below.

Appendix Figure 1.3 Flow of Instruction Processing in the O and S Pipes
Appendix 1.3 Instructions and Pipelined Processing

The pipelines incorporated in the OPSP CPU each consist of five stages. Since the MEM stage is used for only load/store instructions, and the E2 stage is used for only DSP function instructions that write data to accumulators, all other instructions are processed at five pipeline stages.

For load/store instructions

5 stages
Pipeline

* If memory access is performed with zero wait states, the MEM stage is executed in one cycle. Otherwise, the MEM stage is executed in two or more cycles.

For DSP function instructions

(Instructions that write data to the accumulator)

5 stages
Pipeline

For other instructions

4 stages
Pipeline

* For multi-cycle instructions such as multiply or divide instructions, the E stage is executed in two or more cycles.
Appendix 1.4 Pipelined Processing of Parallel Instructions

The OPSP CPU uses two pipelines (O and S pipes) at the same time to accomplish parallel instruction processing. A pair of 16-bit instructions to be executed in parallel are forwarded through pipeline stages at the same time from the IF to the D stage. After being forwarded to the E stage, they are processed independently in the O and S pipes. Example pipeline operations for parallel instruction execution are shown below.

Case 1: Parallel execution of a left-side instruction (O–) and a right-side instruction (–S)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>O Pipe</th>
<th>S Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, @R2</td>
<td>IF D E MEM WB</td>
<td>IF D E1 E2 WB</td>
</tr>
<tr>
<td>MULHI R3, R4</td>
<td>IF D E1 E2 WB</td>
<td></td>
</tr>
</tbody>
</table>

Case 2: Parallel execution of a left-side instruction (O–) and a both-side instruction (OS)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>O Pipe</th>
<th>S Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, @R2</td>
<td>IF D E MEM WB</td>
<td>IF D E1 WB</td>
</tr>
<tr>
<td>ADD R3, R4</td>
<td>IF D E1 WB</td>
<td></td>
</tr>
</tbody>
</table>

Case 3: Parallel execution of a both-side instruction (OS) and a right-side instruction (–S)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>O Pipe</th>
<th>S Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2</td>
<td>IF D E WB</td>
<td>IF D E1 E2 WB</td>
</tr>
<tr>
<td>MULHI R3, R4</td>
<td>IF D E1 E2 WB</td>
<td></td>
</tr>
</tbody>
</table>

Case 4: Parallel execution of a both-side instruction (OS) and another both-side instruction (OS)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>O Pipe</th>
<th>S Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2</td>
<td>IF D E WB</td>
<td>IF D E1 WB</td>
</tr>
<tr>
<td>ADD R3, R4</td>
<td>IF D E1 WB</td>
<td></td>
</tr>
</tbody>
</table>

Appendix Figure 1.5 Pipelined Processing of Parallel Instructions
Appendix 1.5 Basic Pipeline Operation

In ideal pipelined instruction processing, it can be expected that each stage is executed in one cycle. However, pipeline operation may be disturbed by processing at a particular stage or by execution of a branch instruction.

The following shows basic pipeline operation for several typical cases.

<Case 1>: When executing an instruction that requires two or more cycles for execution at the E stage

\[
\begin{align*}
\text{DIV} & \quad R1, R2 \quad \begin{array}{c|c|c|c|c|c}
\text{IF} & \text{D} & \text{E} & \text{E} & \cdots & \text{E} \quad \text{WB} \\
\end{array} \\
\text{ADD} & \quad R3, R4 \quad \begin{array}{c|c|c|c|c|c}
\text{IF} & \text{D} & \text{stall} & \cdots & \text{stall} & \text{E} \quad \text{WB} \\
\end{array} \\
\text{ADD} & \quad R5, R6 \quad \begin{array}{c|c|c|c|c|c}
\text{IF} & \text{stall} & \cdots & \text{stall} & \text{D} \quad \text{E} \quad \text{WB} \\
\end{array} \\
\text{ADD} & \quad R7, R8 \quad \begin{array}{c|c|c|c|c|c}
\text{stall} & \cdots & \text{stall} & \text{IF} \quad \text{D} \quad \text{E} \quad \text{WB} \\
\end{array}
\end{align*}
\]

<Case 2>: When operand access cannot be finished in one cycle

\[
\begin{align*}
\text{LD} & \quad R1, @R2 \quad \begin{array}{c|c|c|c|c|c|c}
\text{IF} & \text{D} & \text{E} & \text{MEM} & \text{MEM} & \cdots & \text{MEM} \quad \text{WB} \\
\end{array} \\
\text{LD} & \quad R3, @R4 \quad \begin{array}{c|c|c|c|c|c|c}
\text{IF} & \text{D} & \text{E} & \text{stall} & \cdots & \text{stall} & \text{MEM} \quad \text{WB} \\
\end{array} \\
\text{ADD} & \quad R5, R6 \quad \begin{array}{c|c|c|c|c|c|c}
\text{IF} & \text{D} & \text{stall} & \cdots & \text{stall} & \text{E} \quad \text{WB} \\
\end{array} \\
\text{ADD} & \quad R7, R8 \quad \begin{array}{c|c|c|c|c|c|c}
\text{IF} & \text{stall} & \cdots & \text{stall} & \text{D} \quad \text{E} \quad \text{WB} \\
\end{array}
\end{align*}
\]

Memory access with other than zero wait states

\[
\begin{align*}
\text{LD} & \quad R1, @R2 \\
\text{LD} & \quad R3, @R4 \\
\text{ADD} & \quad R5, R6 \\
\text{ADD} & \quad R7, R8
\end{align*}
\]

stall : Pipeline stall

Appendix Figure 1.6 Cases where Pipeline Operation is Disturbed - 1
<Case 3>: When executing a branch instruction (except for conditional branch instructions that did not cause control to branch off)

Branch instruction executed

Branch instruction

\[
\begin{array}{cccc}
IF & D & E & WB \\
IF & D & IF & D & E & WB \\
IF & stall & IF & D & E & WB \\
stall & stall & IF & D & E & WB \\
\end{array}
\]

<Case 4>: Where the subsequent instruction uses the operand read from memory

LD R1, @R2

\[
\begin{array}{cccc}
IF & D & E & MEM & WB \\
\end{array}
\]

ADD R3, R1

\[
\begin{array}{cccc}
IF & D & stall & stall & E & WB \\
\end{array}
\]

<Case 5>: Where after writing to the PSW register SM bit in MVTC instruction, the subsequent instruction reads out R15

MVTC R1, PSW

\[
\begin{array}{cccc}
IF & D & E & WB \\
\end{array}
\]

SUB R3, R15

\[
\begin{array}{cccc}
IF & D & stall & E & WB \\
\end{array}
\]

stall : Pipeline stall

Appendix Figure 1.7 Cases where Pipeline Operation is Disturbed - 2
<Case 6>: Where after executing an instruction that writes data to an accumulator (e.g., DSP function instruction of MULHI), the same accumulator is read out in MVFAC instruction

MULHI R1, R2, A0

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E1</th>
<th>E2</th>
<th>WB</th>
</tr>
</thead>
</table>

MVFACHI R3, A0

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>stall</th>
<th>E1</th>
<th>WB</th>
</tr>
</thead>
</table>

<Case 7>: Where case 1 and case 4 occur at the same time in parallel instruction processing

LD R1, @R2

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>

MUL R3, R4

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>WB</th>
</tr>
</thead>
</table>

ADD R1, R2

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>stall</th>
<th>stall</th>
<th>E</th>
<th>WB</th>
</tr>
</thead>
</table>

ADD R5, R3

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>stall</th>
<th>stall</th>
<th>E1</th>
<th>WB</th>
</tr>
</thead>
</table>

stall : Pipeline stall

Appendix Figure 1.8 Cases where Pipeline Operation is Disturbed - 3
Shown below are special cases where the pipeline operation is not disturbed.

- Where the WB stage of a load instruction and that of a non-load instruction overlap
  (Pipeline processing is not disturbed because writes to registers or accumulators can be performed at the same time.)

```
LD R1, @R2
ADD R1, R2
ADD R5, R3
```

- Where the subsequent instruction uses the register that was written to by the preceding instruction
  (For register-to-register operations, pipeline processing is protected from becoming disturbed by a bypass mechanism.)

```
ADD R1, R2
SUB R3, R1
```

- Where before a load instruction finishes, the subsequent instruction writes to the same register.
  (Load instruction execution at the WB stage is canceled.)

```
LD R1, @R2
```

Appendix Figure 1.9  Special Cases where Pipeline Operation is Not Disturbed
Appendix 2 Instruction Processing Time

The instruction processing time of the OPSP CPU normally is represented by the number of instruction execution cycles at the E stage. Depending on pipeline operation, however, this instruction processing time may be affected by instruction execution at other stages.

The following shows instruction processing time at each pipeline stage of the OPSP CPU.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Number of Execution Cycles at Each Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
</tr>
<tr>
<td>Load instructions (LD, LDB, LDUB, LDH, LDUH, LOCK)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Store instructions (ST, STB, STH, UNLOCK)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>BSET and BCLR instructions</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Multiplication instructions (MUL)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Division/remainder instructions (DIVB, DIVUB, REMB, REMUB)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Division/remainder instructions (DIVH, DIVUH, REMH, REMUH)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Division/remainder instructions (DIV, DIVU, REM, REMU)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>DSP function instructions [1] (When writing data to accumulators)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>DSP function instructions [2] (When not writing data to accumulators)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Instructions other than the above (including DSP function instructions and BTST, SETPSW and CLRPSW instructions)</td>
<td>R&lt;sup&gt;Note2&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Note 1: The DSP function instructions that do not write data to accumulators include MVFACHI, MVFACLO, MVFACMI, SATB and SATH.

Note 2: The number of execution cycles indicated by R and W depends on the type of microcomputer used.

Note 3: For the instructions executed in the O pipe, this indicates the number of execution cycles at the E stage. For the instructions executed in the S pipe, this indicates a total number of execution cycles at both E1 and E2 stages.

Note 4: Although DSP function instructions [1] require two cycles for execution at the E stage, the actual throughput is one because of pipelined instruction processing.