Synergistic Processor Unit
Instruction Set Architecture

Version 1.2

January 27, 2007

All Rights Reserved

"SONY" and "*****" are registered trademarks of Sony Corporation.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change Sony and SCEI product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Sony and SCEI or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments can vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will Sony and SCEI be liable for damages arising directly or indirectly from any use of the information contained in this document.

Sony Corporation
6-7-35 Kitashinagawa, Shinagawa-ku, Tokyo, 141-0001 Japan
(1-7-1 Konan, Minato-ku, Tokyo, 108-0075 Japan: from February 2007)

Sony Computer Entertainment Inc.
2-6-21 Minami-Aoyama, Minato-ku,
Tokyo, 107-0062 Japan

The Sony home page can be found at http://www.sony.net
The SCEI home page can be found at http://www.scei.co.jp

The Cell Broadband Engine home page can be found at http://cell.scei.co.jp

Version 1.2
January 27, 2007
## Contents

**List of Figures** ........................................................................................................  9

**List of Tables** ........................................................................................................ 11

**Preface** ...................................................................................................................... 13

- Who Should Read This Document ........................................................................... 13
- Related Documents .................................................................................................. 13
- Document Organization ........................................................................................... 13
- Version Numbering .................................................................................................. 14
- How to Use the Instruction Descriptions .................................................................. 15
- Conventions and Notations Used in This Manual ....................................................... 16
  - Byte Ordering ......................................................................................................... 16
  - Bit Ordering .......................................................................................................... 16
  - Bit Encoding ......................................................................................................... 16
  - Instructions, Mnemonics, and Operands ................................................................. 16
  - Referencing Registers or Channels, Fields, and Bit Ranges ............................... 17
  - Register Transfer Language Instruction Definitions .............................................. 18
  - Instruction Fields .................................................................................................. 19
  - Instruction Operation Notations ............................................................................ 20

**Revision Log** ........................................................................................................... 21

**1. Introduction** .......................................................................................................... 23

**2. SPU Architectural Overview** .............................................................................. 25

- 2.1 Data Representation ........................................................................................... 25
- 2.2 Data Layout in Registers ..................................................................................... 28
- 2.3 Instruction Formats ............................................................................................. 28

**3. Memory—Load/Store Instructions** .................................................................... 31

- Load Quadword (d-form) ......................................................................................... 32
- Load Quadword (x-form) ........................................................................................ 33
- Load Quadword (a-form) ......................................................................................... 34
- Load Quadword Instruction Relative (a-form) ......................................................... 35
- Store Quadword (d-form) ....................................................................................... 36
- Store Quadword (x-form) ....................................................................................... 37
- Store Quadword (a-form) ....................................................................................... 38
- Store Quadword Instruction Relative (a-form) ....................................................... 39
- Generate Controls for Byte Insertion (d-form) ...................................................... 40
- Generate Controls for Byte Insertion (x-form) ...................................................... 41
- Generate Controls for Halfword Insertion (d-form) .............................................. 42
- Generate Controls for Halfword Insertion (x-form) .............................................. 43
- Generate Controls for Word Insertion (d-form) .................................................... 44
- Generate Controls for Word Insertion (x-form) .................................................... 45
6. Shift and Rotate Instructions  

Shift Left Halfword ................................................................. 118
Shift Left Halfword Immediate ............................................... 119
Shift Left Word ........................................................................ 120
Shift Left Word Immediate ...................................................... 121
Shift Left Quadword by Bits .................................................. 122
Shift Left Quadword by Bits Immediate ................................. 123
Shift Left Quadword by Bytes ................................................ 124
Shift Left Quadword by Bytes Immediate ............................. 125
Shift Left Quadword by Bytes from Bit Shift Count ............. 126
Rotate Halfword ..................................................................... 127
Rotate Halfword Immediate ................................................ 128
Rotate Word ........................................................................... 129
Rotate Word Immediate ........................................................ 130
Rotate Quadword by Bytes ....................................................... 131
Rotate Quadword by Bytes Immediate .................................... 132
Rotate Quadword by Bytes from Bit Shift Count ................. 133
Rotate Quadword by Bits .......................................................... 134
Rotate Quadword by Bits Immediate ...................................... 135
Rotate and Mask Halfword .................................................... 136
Rotate and Mask Halfword Immediate .................................. 137
Rotate and Mask Word .......................................................... 138
Rotate and Mask Word Immediate ......................................... 139
Rotate and Mask Quadword by Bytes ................................ 140
Rotate and Mask Quadword by Bytes Immediate ............... 141
Rotate and Mask Quadword Bytes from Bit Shift Count ...... 142
Rotate and Mask Quadword by Bits ....................................... 143
Rotate and Mask Quadword by Bits Immediate ................... 144
### 7. Compare, Branch, and Halt Instructions

- Halt If Equal ................................................................. 150
- Halt If Equal Immediate ................................................ 151
- Halt If Greater Than ..................................................... 152
- Halt If Greater Than Immediate ...................................... 153
- Halt If Logically Greater Than ........................................ 154
- Halt If Logically Greater Than Immediate ....................... 155
- Compare Equal Byte ..................................................... 156
- Compare Equal Byte Immediate ...................................... 157
- Compare Equal Halfword ............................................... 158
- Compare Equal Halfword Immediate ............................... 159
- Compare Equal Word .................................................... 160
- Compare Equal Word Immediate ..................................... 161
- Compare Greater Than Byte ........................................... 162
- Compare Greater Than Byte Immediate ........................... 163
- Compare Greater Than Halfword .................................... 164
- Compare Greater Than Halfword Immediate ..................... 165
- Compare Greater Than Word .......................................... 166
- Compare Greater Than Word Immediate .......................... 167
- Compare Logical Greater Than Byte ............................... 168
- Compare Logical Greater Than Byte Immediate ................ 169
- Compare Logical Greater Than Halfword ......................... 170
- Compare Logical Greater Than Halfword Immediate .......... 171
- Compare Logical Greater Than Word ............................... 172
- Compare Logical Greater Than Word Immediate ............... 173
- Branch Relative .............................................................. 174
- Branch Absolute ............................................................ 175
- Branch Relative and Set Link ........................................ 176
- Branch Absolute and Set Link ....................................... 177
- Branch Indirect .............................................................. 178
- Interrupt Return ............................................................ 179
- Branch Indirect and Set Link if External Data .................... 180
- Branch Indirect and Set Link ........................................... 181
- Branch If Not Zero Word ................................................. 182
- Branch If Zero Word ....................................................... 183
- Branch If Not Zero Halfword .......................................... 184
- Branch If Zero Halfword ................................................. 185
- Branch Indirect If Zero ................................................... 186
- Branch Indirect If Not Zero .............................................. 187
- Branch Indirect If Zero Halfword ..................................... 188
- Branch Indirect If Not Zero Halfword ............................. 189
8. Hint-for-Branch Instructions ................................................................. 191
   Hint for Branch (r-form) ........................................................................ 192
   Hint for Branch (a-form) ...................................................................... 193
   Hint for Branch Relative .................................................................... 194

9. Floating-Point Instructions ................................................................. 195
   9.1 Single Precision (Extended-Range Mode) .................................... 195
   9.2 Double Precision .......................................................................... 197
      9.2.1 Conversions Between Single-Precision and Double-Precision Format ................ 198
      9.2.2 Exception Conditions ......................................................... 198
   9.3 Floating-Point Status and Control Register ............................. 200
      Floating Add ................................................................................ 202
      Double Floating Add .................................................................... 203
      Floating Subtract ........................................................................ 204
      Double Floating Subtract .......................................................... 205
      Floating Multiply ......................................................................... 206
      Double Floating Multiply ............................................................ 207
      Floating Multiply and Add ............................................................ 208
      Double Floating Multiply and Add .............................................. 209
      Floating Negative Multiply and Subtract .................................... 210
      Double Floating Negative Multiply and Subtract ....................... 211
      Floating Multiply and Subtract .................................................... 212
      Double Floating Multiply and Subtract ....................................... 213
      Double Floating Negative Multiply and Add ............................... 214
      Floating Reciprocal Estimate ....................................................... 215
      Floating Reciprocal Absolute Square Root Estimate .................. 217
      Floating Interpolate ..................................................................... 219
      Convert Signed Integer to Floating ............................................. 220
      Convert Floating to Signed Integer .............................................. 221
      Convert Unsigned Integer to Floating .......................................... 222
      Convert Floating to Unsigned Integer .......................................... 223
      Floating Round Double to Single ............................................... 224
      Floating Extend Single to Double ............................................... 225
      Double Floating Compare Equal ............................................... 226
      Double Floating Compare Magnitude Equal ............................... 227
      Double Floating Compare Greater Than ...................................... 228
      Double Floating Compare Magnitude Greater Than .................... 229
      Double Floating Test Special Value ............................................. 230
      Floating Compare Equal ............................................................. 231
      Floating Compare Magnitude Equal ............................................. 232
      Floating Compare Greater Than .................................................. 233
      Floating Compare Magnitude Greater Than ............................... 234
      Floating-Point Status and Control Register Write ...................... 235
      Floating-Point Status and Control Register Read ...................... 236

10. Control Instructions ................................................................. 237
    Stop and Signal .............................................................................. 238
    Stop and Signal with Dependencies ............................................... 239
    No Operation (Load) ................................................................. 240
List of Figures

Figure i. Format of an Instruction Description ................................................................. 15
Figure 2-1. Bit and Byte Numbering of Halfwords ........................................................... 26
Figure 2-2. Bit and Byte Numbering of Words ................................................................. 26
Figure 2-3. Bit and Byte Numbering of Doublewords ..................................................... 26
Figure 2-4. Bit and Byte Numbering of Quadwords ......................................................... 27
Figure 2-5. Register Layout of Data Types ................................................................. 28
Figure 2-6. RR Instruction Format .............................................................................. 28
Figure 2-7. RRR Instruction Format ............................................................................. 28
Figure 2-8. RI7 Instruction Format .............................................................................. 28
Figure 2-9. RI10 Instruction Format ............................................................................. 29
Figure 2-10. RI16 Instruction Format ........................................................................... 29
Figure 2-11. RI18 Instruction Format ........................................................................... 29
Figure 13-1. Systems with Multiple Accesses to Local Storage ................................. 253
List of Tables

Table i. Temporary Names Used in the RTL and Their Widths .......................................................... 18
Table ii. Instruction Fields .................................................................................................................. 19
Table iii. Instruction Operation Notations ........................................................................................ 20
Table 1-1. Key Features of the SPU ISA Architecture and Implementation ........................................ 23
Table 2-1. Bit and Byte Numbering Figures ......................................................................................... 26
Table 3-1. Example LSLR Values and Corresponding Local Storage Sizes ......................................... 31
Table 5-1. Binary Values in Register RC and Byte Results ................................................................. 116
Table 9-1. Single-Precision (Extended-Range Mode) Minimum and Maximum Values ...................... 195
Table 9-2. Instructions and Exception Settings .................................................................................... 196
Table 9-3. Double-Precision (IEEE Mode) Minimum and Maximum Values ........................................ 197
Table 9-4. Single-Precision (IEEE Mode) Minimum and Maximum Values ........................................ 198
Table 9-5. Instructions and Exception Settings .................................................................................... 200
Table 12-1. Feature Bits [D] and [E] Settings and Results ................................................................. 251
Table 13-1. Local Storage Accesses ................................................................................................... 253
Table 13-2. Synchronization Instructions ............................................................................................ 255
Table 13-3. Synchronizing Multiple Accesses to Local Storage ......................................................... 256
Table 13-4. Sending Data and Synchronizing through Local Storage .................................................... 257
Table 13-5. Receiving Data and Synchronizing through Local Storage ................................................. 257
Table 13-6. Synchronizing through the Channel Interface ................................................................. 258
Table A-1. Instructions Sorted by Mnemonic ....................................................................................... 259
Table B-1. Byte Insertion: Rightmost 4 Bits of the Effective Address and Created Mask ..................... 265
Table B-2. Halfword Insertion: Rightmost 4 Bits of the Effective Address and Created Mask ................ 266
Table B-3. Word Insertion: Rightmost 4 Bits of the Effective Address and Created Mask .................... 266
Table B-4. Doubleword Insertion: Rightmost 4 Bits of Effective Address and Created Mask ............... 266
Preface

The purpose of this document is to describe the Synergistic Processor Unit (SPU) Instruction Set Architecture (ISA) as it relates to the Cell Broadband Engine™ Architecture (CBEA).

Who Should Read This Document

This document is intended for designers who plan to develop products using the SPU ISA. Use this document in conjunction with the documents listed in Related Documents on page 13.

Related Documents

The following documents are reference materials for the SPU ISA.

<table>
<thead>
<tr>
<th>Title</th>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Broadband Engine Architecture</td>
<td>1.01</td>
<td>October 2006</td>
</tr>
<tr>
<td>PowerPC User Instruction Set Architecture, Book I</td>
<td>2.02</td>
<td>January 2005</td>
</tr>
<tr>
<td>PowerPC Virtual Environment Architecture, Book II</td>
<td>2.02</td>
<td>January 2005</td>
</tr>
<tr>
<td>PowerPC Operating Environment Architecture, Book III</td>
<td>2.02</td>
<td>January 2005</td>
</tr>
</tbody>
</table>

Document Organization

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front Matter</td>
<td>Title Page, Copyright and Disclaimer, Contents, List of Figures, List of Tables</td>
</tr>
<tr>
<td>Preface</td>
<td>Describes this document, related documents, responsibilities, and other general information</td>
</tr>
<tr>
<td>Revision Log</td>
<td>High-level list of changes from the last version to this version</td>
</tr>
<tr>
<td>Section 1 Introduction on page 23</td>
<td>Provides a high-level description of the SPU architecture and its purpose.</td>
</tr>
<tr>
<td>Section 2 SPU Architectural Overview on page 25</td>
<td>Provides an overview of the SPU architecture.</td>
</tr>
<tr>
<td>Section 3 Memory—Load/Store Instructions on page 31</td>
<td>Lists and describes the SPU load/store instructions.</td>
</tr>
<tr>
<td>Section 4 Constant-Formation Instructions on page 49</td>
<td>Lists and describes the SPU constant-formation instructions.</td>
</tr>
<tr>
<td>Section 5 Integer and Logical Instructions on page 57</td>
<td>Lists and describes the SPU integer and logical instructions.</td>
</tr>
<tr>
<td>Section 6 Shift and Rotate Instructions on page 117</td>
<td>Lists and describes the SPU shift and rotate instructions.</td>
</tr>
<tr>
<td>Section 7 Compare, Branch, and Halt Instructions on page 149</td>
<td>Lists and describes the SPU compare, branch, and halt instructions.</td>
</tr>
<tr>
<td>Section 8 Hint-for-Branch Instructions on page 191</td>
<td>Lists and describes the SPU hint-for-branch instruction.</td>
</tr>
<tr>
<td>Section 9 Floating-Point Instructions on page 195</td>
<td>Lists and describes the SPU floating-point instructions.</td>
</tr>
<tr>
<td>Section 10 Control Instructions on page 237</td>
<td>Lists and describes the SPU control instructions.</td>
</tr>
<tr>
<td>Section 11 Channel Instructions on page 247</td>
<td>Describes the instructions used to communicate between the SPU and external devices through the channel interfaces.</td>
</tr>
</tbody>
</table>
**Synergistic Processor Unit**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Section 12 SPU Interrupt Facility on page 251</em></td>
<td>Describes the SPU interrupt facility.</td>
</tr>
<tr>
<td><em>Section 13 Synchronization and Ordering on page 253</em></td>
<td>Describes the SPU sequentially ordered programming model.</td>
</tr>
<tr>
<td><em>Appendix A Instruction Table Sorted by Instruction Mnemonic on page 259</em></td>
<td>Lists the SPU instructions sorted by their mnemonics.</td>
</tr>
<tr>
<td><em>Appendix B Details of the Generate Controls Instructions on page 265</em></td>
<td>Provides the details of the masks that are generated by the generate controls instructions.</td>
</tr>
</tbody>
</table>

**Version Numbering**

The document version number appears on the title page and in the footer of every page. The format of the version number is V.xy, where:

- **V** is the major version level. This number is incremented when a new required feature is added to the architecture. The major and minor revision numbers are set to zero. For example, version 1.12 becomes version 2.00.

- **x** is the major revision level. This number is incremented when a new, optional feature is added to the architecture or a major change is added that could affect a programmer. The minor revision level is set to zero. For example, version 1.12 becomes version 1.20.

- **y** is the minor revision level. This number is incremented for every new release that does not contain any new required or optional features. For example, version 1.12 becomes version 1.13.
How to Use the Instruction Descriptions

Figure i illustrates how to use the instruction descriptions provided in this document.

Figure i. Format of an Instruction Description

Instruction Name: Load Quadword (d-form)

Instruction Mnemonic: lqi

Instruction Operands: rt, symbol(ra)

Instruction Format:

<table>
<thead>
<tr>
<th>Required or Optional</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>v 1.0</td>
</tr>
</tbody>
</table>

Instruction Opcode (Binary):

The local storage address is computed by adding the signed value in the H0 field, with 4 zero bits appended, to the value in the preferred slot of register RA and forcing the rightmost 4 bits of the sum to zero. The 16 bytes at the local storage address are placed into register RT. This instruction is computed using the following formula:

\[
\text{LSA} \leftarrow \text{RoundToBin}(\text{H0} \oplus \text{RA}^{02} \& \text{LSUR}_{8} \& 0x\_FFFFFD) \\
\text{RT} \leftarrow \text{Loc8Sec}(\text{LSA}, 16)
\]
Conventions and Notations Used in This Manual

Byte Ordering
Throughout this document, standard IBM big-endian notation is used, meaning that bytes are numbered in ascending order from left to right. Big-endian and little-endian byte ordering are described in the Cell Broadband Engine Architecture document.

Bit Ordering
Bits are numbered in ascending order from left to right with bit 0 representing the most-significant bit (MSb) and bit 31 the least-significant bit (LSb).

Bit Encoding
The notation for bit encoding is as follows:

- Hexadecimal values are preceded by 0x.
  For example: 0x0A00.
- Binary values are preceded by 0b.
  For example: 0b1010.

Instructions, Mnemonics, and Operands
This document follows the following conventions for instructions, mnemonics, and operands:

- Instruction mnemonics are written in **bold** type. For example, **sync** for the synchronize instruction.
- Each instruction description in this document indicates whether the instruction is optional or required and which version of the architecture introduced the instruction. The instruction description includes the mnemonic and a formatted list of operands as shown in *Figure i* on page 15. In addition, each instruction description provides a sample assembler language statement showing the format supported by the assembler.
- Variables are written in italic type.
Referencing Registers or Channels, Fields, and Bit Ranges

Registers and channels are referred to by their full name or by their mnemonic (also called the short name). Fields are referred to by their field name or by their bit position.

Usually, the register mnemonic is followed by the field name or bit position enclosed in brackets. For example: MSR[R]. An equal sign followed by a value indicates the value to which the field is set; for example, MSR[R] = 0. When referencing a range of bit numbers, the starting and ending bit numbers are enclosed in brackets and separated by a colon; for example, [0:34].

The following table describes how registers, fields, and bit ranges are referred to in this document and provides examples of the references.

<table>
<thead>
<tr>
<th>Type of Reference</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference to a specific register and a specific field using the register short name and the field name</td>
<td>Register_Short_Name[Field_Name]</td>
<td>MSR[R]</td>
</tr>
<tr>
<td>Reference to a field using the field name</td>
<td>[Field_Name]</td>
<td>[R]</td>
</tr>
<tr>
<td>Reference to a specific register and to multiple fields using the register short name and the field names</td>
<td>Register_Short_Name[Field_Name1, Field_Name2]</td>
<td>MSR[FE0, FE1]</td>
</tr>
<tr>
<td>Reference to a specific register and to multiple fields using the register short name and the bit positions.</td>
<td>Register_Short_Name[Bit_Number, Bit_Number]</td>
<td>MSR[52, 55]</td>
</tr>
<tr>
<td>Reference to a specific register and to a field using the register short name and the bit position or the bit range.</td>
<td>Register_Short_Name[Bit_Number]</td>
<td>MSR[52]</td>
</tr>
<tr>
<td></td>
<td>Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]</td>
<td>MSR[39:44]</td>
</tr>
<tr>
<td>A field name followed by an equal sign (=) and a value indicates the value for that field.</td>
<td>Register_Short_Name[Field_Name]=n</td>
<td>MSR[FE0]=0b1, MSR[FE]=0x1</td>
</tr>
<tr>
<td></td>
<td>Register_Short_Name[Bit_Number]=n</td>
<td>MSR[52]=0b0, MSR[52]=0x0</td>
</tr>
<tr>
<td></td>
<td>Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]=n</td>
<td>MSR[39:43]=0b10010, MSR[39:43]=0x11</td>
</tr>
</tbody>
</table>

1. Where \( n \) is the binary or hexadecimal value for the field or bits specified in the brackets.
Register Transfer Language Instruction Definitions

This document generally follows the register transfer language (RTL) terminology and notation in the PowerPC® Architecture™.

RTL descriptions are provided for most instructions and are intended to clarify the verbal description, which is the primary definition. The following conventions apply to the RTL:

- **LocStor(x,y)** refers to the y bytes starting at local storage location x.
- **RepLeftBit(x,y)** returns the value x with its leftmost bit replicated enough times to produce a total length of y.
- The program counter (PC) contains the address of the instruction being executed when used as an operand, or the address of the next instruction when used as a target.
- Temporary names used in the RTL descriptions have the widths shown in Table i.

Table i. Temporary Names Used in the RTL and Their Widths

<table>
<thead>
<tr>
<th>Temporary Name</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>b, byte, byte1, byte2, c</td>
<td>8 bits</td>
</tr>
<tr>
<td>r, s</td>
<td>16 bits</td>
</tr>
<tr>
<td>bbbb, EA, QA, t, t0, t1, t2, t3, u, v</td>
<td>32 bits</td>
</tr>
<tr>
<td>Q, R, Memdata</td>
<td>128 bits</td>
</tr>
<tr>
<td>Rconcat</td>
<td>256 bits</td>
</tr>
<tr>
<td>i, j, k, m</td>
<td>Meta (for description only)</td>
</tr>
</tbody>
</table>
Instruction Fields

The instructions in this document can contain one or more of the fields described in Table ii.

Table ii. Instruction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/, //, ///</td>
<td>Reserved field in an instruction. Reserved fields that are currently not in use contain zeros even where this is not checked by the architecture; this allows for future use without causing incompatibility.</td>
</tr>
<tr>
<td>I7</td>
<td>7-bit immediate</td>
</tr>
<tr>
<td>I8</td>
<td>8-bit immediate</td>
</tr>
<tr>
<td>I10</td>
<td>10-bit immediate</td>
</tr>
<tr>
<td>I16</td>
<td>16-bit immediate</td>
</tr>
<tr>
<td>OP or OPCD</td>
<td>Opcode</td>
</tr>
<tr>
<td>RA[18-24]</td>
<td>Field used to specify a general-purpose register (GPR) to be used as a source or as a target.</td>
</tr>
<tr>
<td>RB[11-17]</td>
<td>Field used to specify a GPR to be used as a source or as a target.</td>
</tr>
<tr>
<td>RC[4-10]</td>
<td>Field used to specify a GPR to be used as a source or as a target.</td>
</tr>
<tr>
<td>RT[25-31]</td>
<td>Field used to specify a GPR to be used as a target.</td>
</tr>
</tbody>
</table>
Instruction Operation Notations

The instructions in this document use the notations described in Table iii. This table is ordered with respect to the order of precedence, where the first operator in the table binds most tightly.

Table iii. Instruction Operation Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>See Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_p$</td>
<td>Means bit $p$ of register or value field $X$</td>
<td></td>
</tr>
<tr>
<td>$X_{p\colon q}$</td>
<td>Means bits $p$ through $q$ inclusive of register or value $X$</td>
<td></td>
</tr>
<tr>
<td>$X^p$</td>
<td>Means byte $p$ of register or value $X$</td>
<td></td>
</tr>
<tr>
<td>$X_{p\colon q}$</td>
<td>Means bytes $p$ through $q$ inclusive of register or value $X$</td>
<td></td>
</tr>
<tr>
<td>$X_{p\colon q}$</td>
<td>Means bits $p$ and the bits that follow for a total of $q$ bits</td>
<td></td>
</tr>
<tr>
<td>$X^p_{\colon q}$</td>
<td>Means bytes $p$ and the bytes that follow for a total of $q$ bytes</td>
<td></td>
</tr>
<tr>
<td>$p0$ and $p1$</td>
<td>Mean a string of $p$ 0 bits and of $p$ 1 bits.</td>
<td>1</td>
</tr>
<tr>
<td>$\neg$</td>
<td>unary NOT operator</td>
<td>2</td>
</tr>
<tr>
<td>$\ast$, $</td>
<td>\ast</td>
<td>$</td>
</tr>
<tr>
<td>$+$</td>
<td>Two’s complement addition</td>
<td>2</td>
</tr>
<tr>
<td>$-$</td>
<td>Two’s complement subtraction, unary minus</td>
<td>2</td>
</tr>
<tr>
<td>$=$</td>
<td>Equals</td>
<td>2</td>
</tr>
<tr>
<td>$\neq$</td>
<td>Not Equals relations</td>
<td>2</td>
</tr>
<tr>
<td>$&lt;, \leq, &gt;, \geq$</td>
<td>Signed comparison relations</td>
<td></td>
</tr>
<tr>
<td>$&lt;u, &gt;u$</td>
<td>Unsigned comparison relations</td>
<td></td>
</tr>
<tr>
<td>$&amp;$</td>
<td>AND</td>
<td>2</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>OR</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>Exclusive OR ($a &amp; \neg b \mid \neg a &amp; b$)</td>
<td>2</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td>LSA</td>
<td>Local Storage Address</td>
<td></td>
</tr>
<tr>
<td>LSLR</td>
<td>Local Storage Limit Register</td>
<td></td>
</tr>
<tr>
<td>LocStor(LSA,$width$)</td>
<td>Contents of the number of bytes indicated by the $width$ variable in local storage at the LSA.</td>
<td></td>
</tr>
<tr>
<td>if (cond) then ... else ...</td>
<td>Conditional execution. Else is optional. The range of the then and else clauses is indicated by indentation. When the clauses are single statements, they are shown on the same line as the corresponding if and else.</td>
<td></td>
</tr>
<tr>
<td>for ... end</td>
<td>For loop. To and by clauses specify incrementing an iteration variable, and a while clause provides termination conditions.</td>
<td></td>
</tr>
<tr>
<td>do ... while (cond)</td>
<td>Do loop. While clause provides termination conditions.</td>
<td></td>
</tr>
<tr>
<td>$l, ll, lll$</td>
<td>Reserved field in an instruction. Reserved fields are presently unused and should contain zeros, even where this is not checked by the architecture, to allow for future use without causing incompatibility.</td>
<td></td>
</tr>
</tbody>
</table>

1. This is different from the PowerPC notation, which uses a leading superscript rather than a subscript.
2. The result of this operator is a bit vector of the same width as the input operands.
3. The result of this operator is a bit vector of the width of the sum of the operand widths.
Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was significantly modified from the previous release of this document.

<table>
<thead>
<tr>
<th>Revision Date</th>
<th>Contents of Modification</th>
<th>Errata?</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 27, 2007</td>
<td>Version 1.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Revised the introduction to the revision log (see Revision Log on page 21).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Updated a figure to illustrate the revised instruction format (see Figure i Format of an Instruction Description on page 15). Also, updated the description on instruction conventions (see Instructions, Mnemonics, and Operands on page 16).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Corrected and clarified the programming note associated with the Multiply High instructions and added a code sample (see Multiply High on page 77).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deleted “nonzero” from the description of an IEEE noncompliant result (see Section 9.1 Single Precision (Extended-Range Mode) on page 195).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Indicated that an exponent field of all ones is reserved for Infinity as well as Not-a-Number (NaN) fields (see Table 9-3 Double-Precision (IEEE Mode) Minimum and Maximum Values on page 197).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Changed the description of handling denormal inputs (see Section 9.2.1 Conversions Between Single-Precision and Double-Precision Format on page 198).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deleted “nonzero” from the description of FPSCR[31] (see Section 9.3 Floating-Point Status and Control Register on page 200).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Added five optional instructions (see Double Floating Compare Equal on page 226, Double Floating Compare Magnitude Equal on page 227, Double Floating Compare Greater Than on page 228, Double Floating Compare Magnitude Greater Than on page 229, and Double Floating Test Special Value on page 230).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Changed “coherency” to “consistency” in two places to conform to the terminology used in Table 13-2 Synchronization Instructions on page 255 (see Section 13.3 Synchronization Primitives on page 254).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Added the new instructions to Appendix A (see Table A-1 Instructions Sorted by Mnemonic on page 259).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Made various editorial changes to the glossary (see Glossary on page 267).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Revised the format of the instruction descriptions throughout. The instruction heading now indicates whether the instruction is optional or required and in which version of the architecture the instruction was introduced.</td>
<td></td>
</tr>
</tbody>
</table>
## Revision Log

<table>
<thead>
<tr>
<th>Revision Date</th>
<th>Contents of Modification</th>
<th>Errata?</th>
</tr>
</thead>
</table>
| October 4, 2006     | - Version 1.11:  
  - Explained the version numbering scheme (see Version Numbering on page 14).  
  - Changed hexadecimal and binary representation throughout (see Bit Encoding on page 16).  
  - Changed the description of bit encoding and the convention for representing variables (see Conventions and Notations Used in This Manual on page 16).  
  - Corrected the expansion of the `bisle` instruction mnemonic (see Section 2 SPU Architectural Overview on page 25).  
  - Corrected the mnemonic for the Add Word instruction (see Multiply High on page 77).  
  - Revised the description of the Select Bits instruction (see page 115). Revised several programming notes to explain how logical right shift and algebraic right shift are supported (see Rotate and Mask Halfword on page 136, Rotate and Mask Halfword Immediate on page 137, Rotate and Mask Word on page 138, and Rotate and Mask Word Immediate on page 139).  
  - Explained the inline prefetch (see Hint for Branch (r-form) on page 192).  
  - Revised the introduction to Section 9 Floating-Point Instructions on page 195 and added an implementation note that explains that the results of floating-point instructions are implementation dependent.  
  - Improved Table 9-1 Single-Precision (Extended-Range Mode) Minimum and Maximum Values on page 195, Table 9-3 Double-Precision (IEEE Mode) Minimum and Maximum Values on page 197, and Table 9-4 Single-Precision (IEEE Mode) Minimum and Maximum Values on page 198.  
  - Improved the description of double-precision instructions and indicated that the rounding mode for each slice can be controlled independently (see Section 9.2 on page 197).  
  - Expanded the explanation of how denormal inputs are handled (see Section 9.2.1 on page 198).  
  - In the Floating-Point Status and Control Register, defined bits 20:21 and redefined bits 22:23 (see Section 9.3 on page 200).  
  - Corrected the description of the `lnop` instruction (see No Operation (Load) on page 240).  
  - Explained how 32-bit values are handled by the 128-bit `mtspr` and `mfspr` instructions (see Move from Special-Purpose Register on page 244 and Move to Special-Purpose Register on page 245).  
  - Explained how 32-bit wide channels are handled by the `rdch` and `wrch` instructions (see Read Channel on page 248 and Write Channel on page 250).  
  - Explained how to synchronize multiple accesses the local storage (see Table 13-3 on page 256).  
  - Provided a more detailed description of external local storage access (see Section 13.6 on page 256).  
  - Simplified and clarified the RTL descriptions of several instructions.  
  - Deleted Appendix A Programming Examples. Appendix B Instruction Table Sorted by Instruction Mnemonic is now Appendix A.  
  - Added an index (see Index on page 271).  
  - Added a glossary (see Glossary on page 267).  
  - Changed “local store” to “local storage” throughout.  
  - Made other changes for consistency and clarity.  | Yes     |
| January 30, 2006    | Version 1.1:  
  - Corrected the pseudocode associated with Rotate and Mask Halfword Immediate (see page 137). |        |
| August 1, 2005      | Initial public release.                                                                                                                                          |        |
1. Introduction

The purpose of the Synergistic Processor Unit (SPU) Instruction Set Architecture (ISA) document is to describe a processor architecture that can fill a void between general-purpose processors and special-purpose hardware. Whereas the objective of general-purpose processor architectures is to achieve the best average performance on a broad set of applications, and the objective of special-purpose hardware is to achieve the best performance on a single application, the purpose of the architecture described in this document is to achieve leadership performance on critical workloads for game, media, and broadband systems. The purpose of the Synergistic Processor Unit Instruction Set Architecture (SPU ISA) and the Cell Broadband Engine Architecture (CBEA) is to provide information that allows a high degree of control by expert (real-time) programmers while still maintaining ease of programming.

The SPU has the following key workloads:
- The graphics pipeline, which includes surface subdivision and rendering
- Stream processing, which includes encoding, decoding, encryption, and decryption
- Modeling, which includes game physics

The implementations of the SPU ISA achieve better performance to cost ratios than general-purpose processors because the SPU ISA implementations require approximately half the power and approximately half the chip area for equivalent performance. This is made possible by the key features of the architecture and implementation listed in Table 1-1.

Table 1-1. Key Features of the SPU ISA Architecture and Implementation (Page 1 of 2)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit SIMD execution unit organization</td>
<td>Many of the applications previously mentioned allow for single-instruction, multiple-data (SIMD) concurrency. In an SIMD architecture, the cost (area and power) of fetching and decoding instructions is amortized over the multiple data elements processed. A 128-bit (most commonly 4-way 32-bit) SIMD has commonality with SIMD processing units in other general-purpose processor architectures and the existing code base to support it.</td>
</tr>
<tr>
<td>Software-managed memory</td>
<td>Whereas most processors reduce latency to memory by employing caches, the SPU in the CBEA implements a small local memory rather than a cache. This approach requires approximately half the area per byte and significantly less power per access, as compared to a cache hierarchy. In addition, it provides a high degree of control for real-time programming. Because the latency and instruction overhead associated with direct memory access (DMA) transfers exceeds that of the latency of servicing a cache miss, this approach achieves an advantage only if the DMA transfer size is sufficiently large and is sufficiently predictable (that is, DMA can be issued before data is needed).</td>
</tr>
<tr>
<td>Load/store architecture to support efficient static random access memory (SRAM) design</td>
<td>The SPU ISA microarchitecture is organized to enable efficient implementations that use single-ported (local storage) memory.</td>
</tr>
<tr>
<td>Large unified register file</td>
<td>The 128-entry register file in the SPU architecture allows for deeply pipelined high-frequency implementations without requiring register renaming to avoid register starvation. This is especially important when latencies are covered by software loop unrolling or other interleaving techniques. Rename hardware typically consumes a significant fraction of the area and power in modern high-frequency general-purpose processors.</td>
</tr>
<tr>
<td>ISA support to eliminate branches</td>
<td>The SPU ISA defines compare instructions to set masks that can be used in three operand select instructions to create efficient conditional assignments. Such conditional assignments can be used to avoid difficult-to-predict branches.</td>
</tr>
</tbody>
</table>
The SPU hint-for-branch instructions allow programs to avoid a penalty on taken branches when the branch can be predicted sufficiently early. This mechanism achieves an advantage over common branch prediction schemes in that it does not require storing history associated with previous branches and thus saves area and power. The ISA solves the problem associated with hint bits in the branch instructions themselves, where considerable look-ahead (branch scan) in the instruction stream is necessary to process branches early enough that their targets are available when needed.

Much of the code base for game applications assumes a single-precision floating-point format that is distinct from the IEEE 754 format commonly implemented on general-purpose processors. For details on the single-precision format, see Section 9 Floating-Point Instructions on page 195.

Blocking channels for communication with the synergistic Memory Flow Controller (MFC) or other parts of the system external to the SPU, provide an efficient mechanism to wait for the completion of external events without polling or interrupts/wait loops, both of which burn power needlessly.

The SPU does not include certain features common in general-purpose processors. Specifically, the processor does not support a supervisor mode.
2. SPU Architectural Overview

This section provides an overview of the SPU architecture.

The SPU architecture defines a set of 128 general-purpose registers (GPRs), each of which contains 128 data bits. Registers are used to hold fixed-point and floating-point data. Instructions operate on the full width of the register, treating the register as multiple operands of the same format.

The SPU supports halfword (16-bit) and word (32-bit) integers in signed format, and it provides limited support for 8-bit unsigned integers. The number representation is two’s complement.

The SPU supports single-precision (32-bit) and double-precision (64-bit) floating-point data in IEEE 754 format. However, full single-precision IEEE 754 arithmetic is not implemented.

The architecture does not use a condition register. Instead, comparison operations set results that are either 0 (false) or 1 (true), and that are the same width as the operands being compared. These results can be used for bitwise masking, the select instruction, or conditional branches.

The SPU loads and stores access a private memory called local storage. The SPU loads and stores transfer quadwords between GPRs and local storage. Implementations can feature varying local storage sizes; however, the local storage address space is limited to 4 GB.

The SPU can send and receive data to external devices through the channel interface. SPU channel instructions transfer quadwords (128 bits) between GPRs and the channel interface. Up to 128 channels are supported. Two channels are defined to access Save-and-Restore Register 0 (SRR0), which holds the address used by the Interrupt Return instruction (iret). The SPU also supports up to 128 special-purpose registers (SPRs). The Move To Special Purpose Register (mtspr) and Move From Special Purpose Register (mfspr) instructions move 128-bit data between GPRs and SPRs.

The SPU also monitors a status signal called the external condition. The Branch Indirect and Set Link If External Data (bisled) instruction conditionally branches based upon the status of the external condition. The SPU interrupt facility can be configured to branch to an interrupt handler at address 0 when the external condition is true.

2.1 Data Representation

The architecture defines the following:

- An 8-bit byte
- A 16-bit halfword
- A 32-bit word
- A 64-bit doubleword
- A 128-bit quadword

Byte ordering defines how the bytes that make up halfwords, words, doublewords, and quadwords are ordered in memory. The SPU supports most-significant byte (MSB) ordering. With MSB ordering, also called big endian, the most-significant byte is located in the lowest addressed byte position in a storage unit (byte 0). Instructions are described in this document as they appear in memory, with successively higher addressed bytes appearing toward the right.

The conventions for bit and byte numbering within the various width storage units are shown in the figures listed in Table 2-1.
These conventions apply to integer and floating-point data (where the most-significant byte holds the sign and at a minimum the start of the exponent). The figures show byte numbers on the top and bit numbers below.

**Figure 2-1. Bit and Byte Numbering of Halfwords**

<table>
<thead>
<tr>
<th>MSb</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
</tbody>
</table>

**Figure 2-2. Bit and Byte Numbering of Words**

<table>
<thead>
<tr>
<th>MSb</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14 15</td>
<td>16 17 18 19 20 21 22 23</td>
<td>24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>

**Figure 2-3. Bit and Byte Numbering of Doublewords**

<table>
<thead>
<tr>
<th>MSb</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14 15</td>
<td>16 17 18 19 20 21 22 23</td>
<td>24 25 26 27 28 29 30 31</td>
<td>32 33 34 35 36 37 38 39</td>
<td>40 41 42 43 44 45 46 47</td>
<td>48 49 50 51 52 53 54 55</td>
<td>56 57 58 59 60 61 62 63</td>
</tr>
</tbody>
</table>
Figure 2-4. Bit and Byte Numbering of Quadwords

| MSb | 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 |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     | 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 |
|     | 32| 33| 34| 35| 36| 37| 38| 39| 40| 41| 42| 43| 44| 45| 46| 47| 48| 49| 50| 51| 52| 53| 54| 55| 56| 57| 58| 59| 60| 61| 62| 63 |
|     | 64| 65| 66| 67| 68| 69| 70| 71| 72| 73| 74| 75| 76| 77| 78| 79| 80| 81| 82| 83| 84| 85| 86| 87| 88| 89| 90| 91| 92| 93| 94| 95 |
|     | 96| 97| 98| 99|100|101|102|103|104|105|106|107|108|109|110|111|112|113|114|115|116|117|118|119|120|121|122|123|124|125|126|127 |

LSb

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
</tr>
<tr>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
</tr>
<tr>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
</tr>
<tr>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
</tr>
<tr>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
</tr>
</tbody>
</table>
2.2 Data Layout in Registers

All GPRs are 128 bits wide. The leftmost word (bytes 0, 1, 2, and 3) of a register is called the *preferred slot*. When instructions use or produce scalar operands or addresses, the values are in the preferred slot. A set of store assist instructions is available to help store bytes, halfwords, words, and doublewords. *Figure 2-5* illustrates how these data types are laid out in a general purpose register (GPR).

*Figure 2-5. Register Layout of Data Types*

<table>
<thead>
<tr>
<th>Preferred Slot</th>
<th>Byte Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8 9 10 11 12 13 14 15</td>
</tr>
</tbody>
</table>

- **BYTE**
- **HALFWORD**
- **ADDRESS**
- **WORD**
- **DOUBLEWORD**
- **QUADWORD**

2.3 Instruction Formats

There are six basic instruction formats. These instructions are all 32 bits long. Minor variations of these formats are also used. Instructions in memory must be aligned on word boundaries. The instruction formats are shown in *Figures 2-6 through 2-11*.

**Note:** The OP code field is presented throughout this document in binary format.

*Figure 2-6. RR Instruction Format*

<table>
<thead>
<tr>
<th>OP</th>
<th>RB</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>17</td>
</tr>
</tbody>
</table>

*Figure 2-7. RRR Instruction Format*

<table>
<thead>
<tr>
<th>OP</th>
<th>RT</th>
<th>RB</th>
<th>RA</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

*Figure 2-8. RI7 Instruction Format*

<table>
<thead>
<tr>
<th>OP</th>
<th>I7</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>17</td>
</tr>
</tbody>
</table>
Figure 2-9. **RI10 Instruction Format**

<table>
<thead>
<tr>
<th>OP</th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 2-10. **RI16 Instruction Format**

<table>
<thead>
<tr>
<th>OP</th>
<th>I16</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 2-11. **RI18 Instruction Format**

<table>
<thead>
<tr>
<th>OP</th>
<th>I18</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
3. Memory—Load/Store Instructions

This section lists and describes the SPU load/store instructions.

The SPU architecture defines a private memory, also called local storage, which is byte-addressed. Load and store instructions combine operands from one or two registers and an immediate value to form the effective address of the memory operand. Only aligned 16-byte-long quadwords can be loaded and stored. Therefore, the rightmost 4 bits of an effective address are always ignored and are assumed to be zero.

The size of the SPU local storage address space is $2^{32}$ bytes. However, an implementation generally has a smaller actual memory size. The effective size of the memory is specified by the Local Storage Limit Register (LSLR). Implementations can provide methods for accessing the LSLR; however, these methods are outside the scope of the SPU Instruction Set Architecture. Implementations can allow modifications to the LSLR value; however, the LSLR must not change while the SPU is running. Every effective address is ANDed with the LSLR before it is used to reference memory. The LSLR can be used to make the memory appear to be smaller than it is, thus providing compatibility for programs compiled for a smaller memory size. The LSLR value is a mask that controls the effective memory size. This value must have the following properties:

- Limit the effective memory size to be less than or equal to the actual memory size
- Be monotonic, so that the least-significant 4 mask bits are ones and so that there is at most a single transition from ‘1’ to ‘0’ and no transitions from ‘0’ to ‘1’ as the bits are read from the least-significant to the most-significant bit. That is, the value must be $2^n - 1$, where $n$ is $\log_2$ (effective memory size).

The effect of this is that references to memory beyond the last byte of the effective size are wrapped—that is, interpreted modulo the effective size. This definition allows an address to be used for a load before it has been checked for validity, and makes it possible to overlap memory latency with other operations more easily.

Stores of less than a quadword are performed by a load-modify-store sequence. A group of assist instructions is provided for this type of sequence. The assist instruction names are prefixed with Generate Control. These instructions are described in this section. For example, see Generate Controls for Byte Insertion (d-form) on page 40.

In a typical system configuration, the SPU local storage is externally accessible. The possibility therefore exists of SPU memory being modified asynchronously during the course of execution of an SPU program. All references (loads, stores) to local storage by an SPU program, and aligned external references to SPU memory, are atomic. Unaligned references are not atomic, and portions of such operations can be observed by a program executing in the SPU. Table 3-1 shows sample LSLRs and the local storage address space size they correspond to.

<table>
<thead>
<tr>
<th>LSLR</th>
<th>Local Storage Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0003 FFFF</td>
<td>256 KB</td>
</tr>
<tr>
<td>0x0001 FFFF</td>
<td>128 KB</td>
</tr>
<tr>
<td>0x0000 FFFF</td>
<td>64 KB</td>
</tr>
<tr>
<td>0x0000 7FFF</td>
<td>32 KB</td>
</tr>
</tbody>
</table>
Load Quadword (d-form)  

\[
lqd \quad rt, \text{symbol}(ra)
\]

<table>
<thead>
<tr>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

The local storage address is computed by adding the signed value in the I10 field, with 4 zero bits appended, to the value in the preferred slot of register RA and forcing the rightmost 4 bits of the sum to zero. The 16 bytes at the local storage address are placed into register RT. This instruction is computed using the following formula:

\[
\begin{align*}
\text{LSA} & \leftarrow (\text{RepLeftBit}(I10 \| 0b0000,32) + RA^{\text{I10}}) \& \text{LSLR} \& 0xFFFFF0 \\
\text{RT} & \leftarrow \text{LocStor}(\text{LSA}, 16)
\end{align*}
\]
Load Quadword (x-form)

The local storage address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the rightmost 4 bits of the sum to zero. The 16 bytes at the local storage address are placed into register RT. This instruction is computed using the following formula:

\[
\text{LSA} \leftarrow (\text{RA}^{0..3} + \text{RB}^{0..3}) \& \text{LSLR} \& 0xFFFFFFF0 \\
\text{RT} \leftarrow \text{LocStor}(\text{LSA},16)
\]

lqx    rt,ra,rb

```
0 0 1 1 1 0 0 1 0 0
RB   RA   RT
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
```

**Required** v 1.0
Load Quadword (a-form)

The value in the I16 field, with 2 zero bits appended and extended on the left with copies of the most-significant bit, is used as the local storage address. The 16 bytes at the local storage address are loaded into register RT.

| LSA | ← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0 |
|-----|--------------------------------------------------|
| RT  | ← LocStor(LSA,16)                               |
Load Quadword Instruction Relative (a-form)  

**lqr**  
**rt**,symbol

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>16 bytes</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

The value in the I16 field, with 2 zero bits appended, is added to the program counter (PC) to form the local storage address. The 16 bytes at the local storage address are loaded into register RT.

| LSA | ←  (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFFF0 |
| RT  | ← LocStor(LSA,16) |
Store Quadword (d-form)  

\texttt{stqd rt,symbol(ra)}

\begin{verbatim}
0 0 1 0 0 1 0 0
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
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
\end{verbatim}

The local storage address is computed by adding the signed value in the I10 field, with 4 zero bits appended, to the value in the preferred slot of register RA and forcing the rightmost 4 bits of the sum to zero. The contents of register RT are stored at the local storage address.

\begin{verbatim}
LSA ← (RepLeftBit(I10 || 0b0000,32) + RA[0:3]) & LSLR & 0xFFFFFFF0
LocStor(LSA,16) ← RT
\end{verbatim}
Store Quadword (x-form) Required v 1.0

stqx rt,ra,rb

```
0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0
```

The local storage address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the rightmost 4 bits of the sum to zero. The contents of register RT are stored at the local storage address.

```
LSA ← (RA^0.3 + RB^0.3) & LSLR & 0xFFFFFFFF0
LocStor(LSA,16) ← RT
```
Store Quadword (a-form)  

\[
\text{stqa rt,symbol}
\]

The value in the I16 field, with 2 zero bits appended and extended on the left with copies of the most-significant bit, is used as the local storage address. The contents of register RT are stored at the location given by the local storage address.

\[
\begin{align*}
\text{LSA} & \leftarrow \text{RepLeftBit}(16 \| 0b00,32) \& \text{LSLR} \& 0xFFFFFFFF0 \\
\text{LocStor}(\text{LSA},16) & \leftarrow \text{RT}
\end{align*}
\]

The value in the I16 field, with 2 zero bits appended and extended on the left with copies of the most-significant bit, is used as the local storage address. The contents of register RT are stored at the location given by the local storage address.
Store Quadword Instruction Relative (a-form)

\textbf{stqr} \quad rt,symbol

\begin{verbatim}
 0 0 1 0 0 0 1 1 1
\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow
\begin{array}{cccccccccccccccccccccccccc}
\end{array}
\end{verbatim}

The value in the I16 field, with two zero bits appended and extended on the left with copies of the most-significant bit, is added to the program counter (PC) to form the local storage address. The contents of register RT are stored at the location given by the local storage address.

\begin{verbatim}
LSA ← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0
LocStor(LSA,16) ← RT
\end{verbatim}
A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA. The address is used to determine the position of the addressed byte within a quadword. Based on the position, a mask is generated that can be used with the Shuffle Bytes (shufb) instruction to insert a byte at the indicated position within a (previously loaded) quadword. The byte is taken from the rightmost byte position of the preferred slot of the RA operand of the shufb instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

<table>
<thead>
<tr>
<th>t</th>
<th>← (RA0:3 + RepLeftBit(I7,32)) &amp; 0x000000F</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>← 0x101112131415161718191A1B1C1D1E1F</td>
</tr>
<tr>
<td>RTt</td>
<td>← 0x03</td>
</tr>
</tbody>
</table>
Generate Controls for Byte Insertion (x-form)  

<table>
<thead>
<tr>
<th>cbx</th>
<th>rt, ra, rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>11010100</td>
</tr>
</tbody>
</table>

A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB. The address is used to determine the position of the addressed byte within a quadword. Based on the position, a mask is generated that can be used with the `shufb` instruction to insert a byte at the indicated position within a (previously loaded) quadword. The byte is taken from the rightmost byte position of the preferred slot of the RA operand of the `shufb` instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

<table>
<thead>
<tr>
<th>t</th>
<th>← (RA\textsuperscript{0:3} + RB\textsuperscript{0:3}) &amp; 0x0000000F</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>← 0x101112131415161718191A1B1C1D1E1F</td>
</tr>
<tr>
<td>RT\textsuperscript{T}</td>
<td>← 0x03</td>
</tr>
</tbody>
</table>
Generate Controls for Halfword Insertion (d-form) Required v 1.0

A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA and forcing the least-significant bit to zero. The address is used to determine the position of an aligned halfword within a quadword. Based on the position, a mask is generated that can be used with the `shufb` instruction to insert a halfword at the indicated position within a quadword. The halfword is taken from the rightmost 2 bytes of the preferred slot of the RA operand of the `shufb` instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

<table>
<thead>
<tr>
<th>chd</th>
<th>rt,symbol(ra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 1 0 1 0 1</td>
<td>I7 RA RT</td>
</tr>
</tbody>
</table>

- $t \leftarrow (RA^{0:3} + \text{RepLeftBit}(I7,32)) \& 0x0000000E$
- $RT \leftarrow 0x101112131415161718191A1B1C1D1E1F$
- $RT_{t:2} \leftarrow 0x0203$
A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the least-significant bit to zero. The address is used to determine the position of an aligned halfword within a quadword. Based on the position, a mask is generated that can be used with the \texttt{shufb} instruction to insert a halfword at the indicated position within a quadword. The halfword is taken from the rightmost 2 bytes of the preferred slot of the RA operand of the \texttt{shufb} instruction. See \textit{Appendix B Details of the Generate Controls Instructions} on page 265 for the details of the created mask.

\begin{tabular}{|c|c|c|}
\hline
\texttt{chx} & \texttt{rt, ra, rb} & \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\texttt{t} & \texttt{← (RA\textsuperscript{0:3} + RB\textsuperscript{0:3}) \& 0x0000000E} \\
\hline
\texttt{RT} & \texttt{← 0x101112131415161718191A1B1C1D1E1F} \\
\hline
\texttt{RT\textsuperscript{1:2}} & \texttt{← 0x0203} \\
\hline
\end{tabular}
A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA and forcing the least-significant 2 bits to zero. The address is used to determine the position of an aligned word within a quadword. Based on the position, a mask is generated that can be used with the \texttt{shufb} instruction to insert a word at the indicated position within a quadword. The word is taken from the preferred slot of the RA operand of the \texttt{shufb} instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

\begin{verbatim}
| t  | \texttt{RA^{0;3} + RepLeftBit(I7,32)} & 0x0000000C |
| RT | \texttt{0x101112131415161718191A1B1C1D1E1F} |
| RT_{\leq 4} | \texttt{0x00010203} |
\end{verbatim}
A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the least-significant 2 bits to zero. The address is used to determine the position of an aligned word within a quadword. Based on the position, a mask is generated that can be used with the `shufb` instruction to insert a word at the indicated position within a quadword. The word is taken from the preferred slot of the RA operand of the `shufb` instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

<table>
<thead>
<tr>
<th>t</th>
<th>← (RA\textsuperscript{0:3} + RB\textsuperscript{0:3}) &amp; 0x0000000C</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>← 0x101112131415161718191A1B1C1D1E1F</td>
</tr>
<tr>
<td>RT\textsuperscript{1:4}</td>
<td>← 0x00010203</td>
</tr>
</tbody>
</table>
A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA and forcing the least-significant 3 bits to zero. The address is used to determine the position of an aligned doubleword within a quadword. Based on the position, a mask is generated that can be used with the `shufb` instruction to insert a doubleword at the indicated position within a quadword. The doubleword is taken from the leftmost 8 bytes of the RA operand of the `shufb` instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

\[
t ← (RA^{0...3} + \text{RepLeftBit}(I7,32)) \& 0x00000008 \\
RT ← 0x101112131415161718191A1B1C1D1E1F \\
RT^{t..8} ← 0x0001020304050607
\]
Generate Controls for Doubleword Insertion (x-form)  Required  v 1.0

cdx  rt,ra,rb

A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the least-significant 3 bits to zero. The address is used to determine the position of the addressed doubleword within a quadword. Based on the position, a mask is generated that can be used with the shufb instruction to insert a doubleword at the indicated position within a quadword. The quadword is taken from the leftmost 8 bytes of the RA operand of the shufb instruction. See Appendix B Details of the Generate Controls Instructions on page 265 for the details of the created mask.

t ← (RA0:3 + RB0:3) & 0x00000008
RT ← 0x101112131415161718191A1B1C1D1E1F
RT1:8 ← 0x0001020304050607
4. Constant-Formation Instructions

This section lists and describes the SPU constant-formation instructions.
Immediate Load Halfword

\texttt{ilh} \quad \texttt{rt}, \texttt{symbol}

<table>
<thead>
<tr>
<th>I16</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>0</td>
<td>31</td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The value in the I16 field is placed in register RT.

\textbf{Programming Note}: There is no Immediate Load Byte instruction. However, that function can be performed by the \texttt{ilh} instruction with a suitable value in the I16 field.

<table>
<thead>
<tr>
<th>s</th>
<th>← I16</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT\textsuperscript{0:1}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{2:3}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{4:5}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{6:7}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{8:9}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{10:11}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{12:13}</td>
<td>← s</td>
</tr>
<tr>
<td>RT\textsuperscript{14:15}</td>
<td>← s</td>
</tr>
</tbody>
</table>
Immediate Load Halfword Upper

\[ \text{ilhu \hspace{1cm} rt, symbol} \]

For each of four word slots:
- The value in the I16 field is placed in the leftmost 16 bits of the word.
- The remaining bits of the word are set to zero.

**Programming Note:** This instruction, when used in conjunction with Immediate Or Halfword Lower (iohl), can be used to form an arbitrary 32-bit value in each word slot of a register. It can also be used alone to load an immediate floating-point constant with up to 7 bits of significance in its fraction.

| RT[0:3]  | ← t |
| RT[4:7]  | ← t |
| RT[8:11] | ← t |
| RT[12:15]| ← t |
Immediate Load Word

```
il rt,symbol

0 1 0 0 0 0 0 0 1  I16  RT
\downarrow  \downarrow  \downarrow  \downarrow  \downarrow  \downarrow  \downarrow  \downarrow  \downarrow  \downarrow
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
```

For each of four word slots:

- The value in the I16 field is expanded to 32 bits by replicating the leftmost bit.
- The resulting value is placed in register RT.

```
t ← RepLeftBit(I16,32)

RT_{0:3} ← t
RT_{4:7} ← t
RT_{8:11} ← t
RT_{12:15} ← t
```
Immediate Load Address

For each of four word slots:

- The value in the I18 field is placed unchanged in the rightmost 18 bits of register RT.
- The remaining bits of register RT are set to zero.

**Programming Note:** Immediate Load Address can be used to load an immediate value, such as an address or a small constant, without sign extension.

| t | ← | 140 || I18 |
| RT⁰:³ | ← | t |
| RT⁴:⁷ | ← | t |
| RT⁸:¹¹ | ← | t |
| RT¹²:¹⁵ | ← | t |
Immediate Or Halfword Lower

iohl \( \text{rt,symbol} \)

```
0 1 1 0 0 0 0 0 1
↓ ↓ ↓ ↓ ↓ ↓ ↓  I16
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
```

For each of four word slots:

- The value in the I16 field is prefaced with zeros and ORed with the value in register RT.
- The result is placed into register RT.

**Programming Note:** Immediate Or Halfword Lower can be used in conjunction with Immediate Load Halfword Upper to load a 32-bit immediate value.

```
t ← 0x0000 || I16
RT^0:3 ← RT^0:3 | t
RT^4:7 ← RT^4:7 | t
RT^8:11 ← RT^8:11 | t
RT^12:15 ← RT^12:15 | t
```
Form Select Mask for Bytes Immediate

fsmbi  rt, symbol

0 0 1 1 0 0 1 1
ขฐาน 0 1 2 3 4 5 6 7
RT
I16

The I16 field is used to create a mask in register RT by making eight copies of each bit. Bits in the operand are related to bytes in the result in a left-to-right correspondence.

**Programming Note:** This instruction can be used to create a mask for use with the Select Bits instruction. It can also be used to create masks for halfwords, words, and doublewords.

\[
\begin{align*}
    s & \leftarrow I16 \\
    \text{for } j = 0 \text{ to } 15 \\
    & \quad \text{If } s_j = 0 \text{ then } r_j \leftarrow 0x00 \\
    & \quad \text{else } r_j \leftarrow 0xFF \\
    \text{end} \\
    RT & \leftarrow r
\end{align*}
\]
5. Integer and Logical Instructions

This section lists and describes the SPU integer and logical instructions.
## Add Halfword

**ah** \( \rightarrow \) \( rt, ra, rb \)

<table>
<thead>
<tr>
<th>0 0 0 1 1 0 0 0 0</th>
<th>RB</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12 13 14 15 16 17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21 22 23 24 25 26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of eight halfword slots:
- The operand from register \( RA \) is added to the operand from register \( RB \).
- The 16-bit result is placed in \( RT \).
- Overflows and carries are not detected.

<table>
<thead>
<tr>
<th>( RT^{0:1} )</th>
<th>( \leftarrow RA^{0:1} + RB^{0:1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( RT^{2:3} )</td>
<td>( \leftarrow RA^{2:3} + RB^{2:3} )</td>
</tr>
<tr>
<td>( RT^{4:5} )</td>
<td>( \leftarrow RA^{4:5} + RB^{4:5} )</td>
</tr>
<tr>
<td>( RT^{6:7} )</td>
<td>( \leftarrow RA^{6:7} + RB^{6:7} )</td>
</tr>
<tr>
<td>( RT^{8:9} )</td>
<td>( \leftarrow RA^{8:9} + RB^{8:9} )</td>
</tr>
<tr>
<td>( RT^{10:11} )</td>
<td>( \leftarrow RA^{10:11} + RB^{10:11} )</td>
</tr>
<tr>
<td>( RT^{12:13} )</td>
<td>( \leftarrow RA^{12:13} + RB^{12:13} )</td>
</tr>
<tr>
<td>( RT^{14:15} )</td>
<td>( \leftarrow RA^{14:15} + RB^{14:15} )</td>
</tr>
</tbody>
</table>
# Add Halfword Immediate

**ahi**  rt,ra,value

<table>
<thead>
<tr>
<th></th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 1 1 1 0 1</td>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The signed value in the I10 field is added to the value in register RA.
- The 16-bit result is placed in RT.
- Overflows and carries are not detected.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>←</td>
<td>RepLeftBit(I10,16)</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{0:1}</td>
<td>←</td>
<td>RA\textsuperscript{0:1} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{2:3}</td>
<td>←</td>
<td>RA\textsuperscript{2:3} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{4:5}</td>
<td>←</td>
<td>RA\textsuperscript{4:5} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{6:7}</td>
<td>←</td>
<td>RA\textsuperscript{6:7} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{8:9}</td>
<td>←</td>
<td>RA\textsuperscript{8:9} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{10:11}</td>
<td>←</td>
<td>RA\textsuperscript{10:11} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{12:13}</td>
<td>←</td>
<td>RA\textsuperscript{12:13} + s</td>
<td></td>
</tr>
<tr>
<td>RT\textsuperscript{14:15}</td>
<td>←</td>
<td>RA\textsuperscript{14:15} + s</td>
<td></td>
</tr>
</tbody>
</table>
Add Word

\[ \text{rt,ra,rb} \]

For each of four word slots:

- The operand from register RA is added to the operand from register RB.
- The 32-bit result is placed in register RT.
- Overflows and carries are not detected.

<table>
<thead>
<tr>
<th>RT&lt;sup&gt;0:3&lt;/sup&gt;</th>
<th>← RA&lt;sup&gt;0:3&lt;/sup&gt; + RB&lt;sup&gt;0:3&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;sup&gt;4:7&lt;/sup&gt;</td>
<td>← RA&lt;sup&gt;4:7&lt;/sup&gt; + RB&lt;sup&gt;4:7&lt;/sup&gt;</td>
</tr>
<tr>
<td>RT&lt;sup&gt;8:11&lt;/sup&gt;</td>
<td>← RA&lt;sup&gt;8:11&lt;/sup&gt; + RB&lt;sup&gt;8:11&lt;/sup&gt;</td>
</tr>
<tr>
<td>RT&lt;sup&gt;12:15&lt;/sup&gt;</td>
<td>← RA&lt;sup&gt;12:15&lt;/sup&gt; + RB&lt;sup&gt;12:15&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Add Word Immediate

```
ai rt,ra,value
```

For each of four word slots:
- The signed value in the I10 field is added to the operand in register RA.
- The 32-bit result is placed in register RT.
- Overflows and carries are not detected.

```
t ← RepLeftBit(I10,32)
RT0:3 ← RA0:3 + t
RT4:7 ← RA4:7 + t
RT8:11 ← RA8:11 + t
RT12:15 ← RA12:15 + t
```
Subtract from Halfword

```
sfh          rt, ra, rb
```

For each of eight halfword slots:
- The value in register RA is subtracted from the value in RB.
- The 16-bit result is placed in register RT.
- Overflows and carries are not detected.

| RT0:1 | ← RB0:1 + (¬RA0:1) + 1 |
| RT2:3 | ← RB2:3 + (¬RA2:3) + 1 |
| RT4:5 | ← RB4:5 + (¬RA4:5) + 1 |
| RT6:7 | ← RB6:7 + (¬RA6:7) + 1 |
| RT8:9 | ← RB8:9 + (¬RA8:9) + 1 |
| RT10:11| ← RB10:11 + (¬RA10:11) + 1 |
| RT12:13| ← RB12:13 + (¬RA12:13) + 1 |
| RT14:15| ← RB14:15 + (¬RA14:15) + 1 |
Subtract from Halfword Immediate

**sfhi**

```
rt, ra, value
```

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
</table>

For each of eight halfword slots:

- The value in register RA is subtracted from the signed value in the I10 field.
- The 16-bit result is placed in register RT.
- Overflows are not detected.

**Programming Note:** Although there is no Subtract Halfword Immediate instruction, its effect can be achieved by using the Add Halfword Immediate with a negative immediate field.

```
t ← RepLeftBit(I10,16)
RT0:1 ← t + (¬RA0:1) + 1
RT2:3 ← t + (¬RA2:3) + 1
RT4:5 ← t + (¬RA4:5) + 1
RT6:7 ← t + (¬RA6:7) + 1
RT8:9 ← t + (¬RA8:9) + 1
RT10:11 ← t + (¬RA10:11) + 1
RT12:13 ← t + (¬RA12:13) + 1
RT14:15 ← t + (¬RA14:15) + 1
```
Subtract from Word

sf rt, ra, rb

For each of four word slots:

- The value in register RA is subtracted from the value in register RB.
- The result is placed in register RT.
- Overflows and carries are not detected.

<table>
<thead>
<tr>
<th></th>
<th>RT&lt;0:3&gt;</th>
<th>← RB&lt;0:3&gt; + (¬RA&lt;0:3&gt;) + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;4:7&gt;</td>
<td>← RB&lt;4:7&gt; + (¬RA&lt;4:7&gt;) + 1</td>
<td></td>
</tr>
<tr>
<td>RT&lt;8:11&gt;</td>
<td>← RB&lt;8:11&gt; + (¬RA&lt;8:11&gt;) + 1</td>
<td></td>
</tr>
<tr>
<td>RT&lt;12:15&gt;</td>
<td>← RB&lt;12:15&gt; + (¬RA&lt;12:15&gt;) + 1</td>
<td></td>
</tr>
</tbody>
</table>
Subtract from Word Immediate

**sfi** rt,ra,value

For each of four word slots:
- The value in register RA is subtracted from the value in the I10 field.
- The result is placed in register RT.
- Overflows and carries are not detected.

**Programming Note:** Although there is no Subtract Immediate instruction, its effect can be achieved by using the Add Immediate with a negative immediate field.

<table>
<thead>
<tr>
<th>t</th>
<th>← RepLeftBit(I10,32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT0:3</td>
<td>← t + (¬RA0:3) + 1</td>
</tr>
<tr>
<td>RT4:7</td>
<td>← t + (¬RA4:7) + 1</td>
</tr>
<tr>
<td>RT8:11</td>
<td>← t + (¬RA8:11) + 1</td>
</tr>
<tr>
<td>RT12:15</td>
<td>← t + (¬RA12:15) + 1</td>
</tr>
</tbody>
</table>
Add Extended

```
addx rt, ra, rb
```

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RB</td>
<td>RA</td>
<td>RT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For each of four word slots:

- The operand from register RA is added to the operand from register RB and the least-significant bit of the operand from register RT.
- The 32-bit result is placed in register RT. Bits 0 to 30 of the RT input are reserved and should be zero.

<table>
<thead>
<tr>
<th>RT</th>
<th>RA</th>
<th>RB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:3</td>
<td>RA3</td>
<td>RB3</td>
</tr>
<tr>
<td>4:7</td>
<td>RA7</td>
<td>RB7</td>
</tr>
<tr>
<td>8:11</td>
<td>RA11</td>
<td>RB11</td>
</tr>
<tr>
<td>12:15</td>
<td>RA15</td>
<td>RB15</td>
</tr>
</tbody>
</table>

\[
RT^0:3 ← RA^0:3 + RB^0:3 + RT_{31}
\]
\[
RT^4:7 ← RA^4:7 + RB^4:7 + RT_{63}
\]
\[
RT^8:11 ← RA^8:11 + RB^8:11 + RT_{95}
\]
\[
RT^{12:15} ← RA^{12:15} + RB^{12:15} + RT_{127}
\]
Carry Generate

cg rt, ra, rb

For each of four word slots:

- The operand from register RA is added to the operand from register RB.
- The carry-out is placed in the least-significant bit of register RT.
- The remaining bits of RT are set to zero.

for j = 0 to 15 by 4
  \[ t_{0:32} = ((0 \ll RA^{j:4}) + (0 \ll RB^{j:4})) \]
  \[ RT^{j:4} \leftarrow 31 \ll t_0 \]
end
Carry Generate Extended

cgx rt, ra, rb

For each of four word slots:

- The operand from register RA is added to the operand from register RB and the least-significant bit of register RT.
- The carry-out is placed in the least-significant bit of register RT.
- The remaining bits of RT are set to zero. Bits 0 to 30 of the RT input are reserved and should be zero.

\[
\text{for } j = 0 \text{ to } 15 \text{ by } 4 \\
\quad t_{0:32} = (0 \ll RA[j:4]) + (0 \ll RB[j:4]) + (320 \ll RT_j) + 31 \\
\quad RT[j:4] \leftarrow 310 \ll t_0 
\]
Subtract from Extended

For each of four word slots:

- The operand from register RA is subtracted from the operand from register RB. An additional '1' is subtracted from the result if the least-significant bit of RT is '0'.
- The 32-bit result is placed in register RT. Bits 0 to 30 of the RT input are reserved and should be zero.

<table>
<thead>
<tr>
<th>y</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{RT}^{0:3} \leftarrow \text{RB}^{0:3} + (\neg \text{RA}^{0:3}) + \text{RT}_{31}
\]
\[
\text{RT}^{4:7} \leftarrow \text{RB}^{4:7} + (\neg \text{RA}^{4:7}) + \text{RT}_{63}
\]
\[
\text{RT}^{8:11} \leftarrow \text{RB}^{8:11} + (\neg \text{RA}^{8:11}) + \text{RT}_{95}
\]
\[
\text{RT}^{12:15} \leftarrow \text{RB}^{12:15} + (\neg \text{RA}^{12:15}) + \text{RT}_{127}
\]
Borrow Generate

\[ \text{bg} \quad \text{rt}, \text{ra}, \text{rb} \]

For each of four word slots:

- If the unsigned value of RA is greater than the unsigned value of RB, then ‘0’ is placed in register RT. Otherwise, ‘1’ is placed in register RT.

\[
\text{for } j = 0 \text{ to } 15 \text{ by } 4 \\
\quad \text{if } (\text{RB}^{j/4} \geq u \text{ RA}^{j/4}) \text{ then } \text{RT}^{j/4} \leftarrow 1 \\
\quad \text{else } \quad \text{RT}^{j/4} \leftarrow 0 \\
\text{end}
\]
Borrow Generate Extended

\texttt{bgx rt,ra,rb}

For each of four word slots:

- The operand from register RA is subtracted from the operand from register RB. An additional ‘1’ is subtracted from the result if the least-significant bit of RT is ‘0’. If the result is less than zero, a ‘0’ is placed in register RT. Otherwise, register RT is set to ‘1’. Bits 0 to 30 of the RT input are reserved and should be zero.

\begin{verbatim}
for j = 0 to 15 by 4
  if (RT\textsubscript{j} \cdot 8 + 31) then
    if (RB\textsubscript{j}^\text{3:0} \geq RA\textsubscript{j}^\text{3:0}) then RT\textsubscript{j}^\text{3:0} \leftarrow 1
    else RT\textsubscript{j}^\text{3:0} \leftarrow 0
  else
    if (RB\textsubscript{j}^\text{3:0} > RA\textsubscript{j}^\text{3:0}) then RT\textsubscript{j}^\text{3:0} \leftarrow 1
    else RT\textsubscript{j}^\text{3:0} \leftarrow 0
\end{verbatim}
Multiply

mpy rt,ra,rb

For each of four word slots:

- The value in the rightmost 16 bits of register RA is multiplied by the value in the rightmost 16 bits of register RB.
- The 32-bit product is placed in register RT.
- The leftmost 16 bits of each operand are ignored.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RT0:3</td>
<td>← RA2:3 * RB2:3</td>
<td></td>
</tr>
<tr>
<td>RT4:7</td>
<td>← RA6:7 * RB6:7</td>
<td></td>
</tr>
<tr>
<td>RT8:11</td>
<td>← RA10:11 * RB10:11</td>
<td></td>
</tr>
<tr>
<td>RT12:15</td>
<td>← RA14:15 * RB14:15</td>
<td></td>
</tr>
</tbody>
</table>
Multiply Unsigned

**mpyu**

rt, ra, rb

| RT0:3 | ← RA2:3 |*| RB2:3 |
|-------|----------------------------------|
| RT4:7 | ← RA6:7 |*| RB6:7 |
| RT8:11| ← RA10:11 |*| RB10:11 |
| RT12:15| ← RA14:15 |*| RB14:15 |

For each of four word slots:

- The rightmost 16 bits of register RA are multiplied by the rightmost 16 bits of register RB, treating both operands as unsigned.
- The 32-bit product is placed in register RT.
Multiply Immediate

\[ \text{mpyi } \quad \text{rt,ra,value} \]

For each of four word slots:
- The signed value in the I10 field is multiplied by the value in the rightmost 16 bits of register RA.
- The resulting product is placed in register RT.

\[
\begin{array}{c|c|c|c}
\text{t} & \leftarrow \text{RepLeftBit(I10,16)} \\
\text{RT}^{0:3} & \leftarrow \text{RA}^{2:3} \times t \\
\text{RT}^{4:7} & \leftarrow \text{RA}^{6:7} \times t \\
\text{RT}^{8:11} & \leftarrow \text{RA}^{10:11} \times t \\
\text{RT}^{12:15} & \leftarrow \text{RA}^{14:15} \times t \\
\end{array}
\]
Multiply Unsigned Immediate

mpyui $rt, ra, value$

For each of four word slots:

- The signed value in the I10 field is extended to 16 bits by replicating the leftmost bit. The resulting value is multiplied by the rightmost 16 bits of register RA, treating both operands as unsigned.
- The resulting product is placed in register RT.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\leftarrow$ RepLeftBit(I10,16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$RT^{0:3}$</td>
<td>$\leftarrow$ RA$^{2:3}$</td>
</tr>
<tr>
<td>$RT^{4:7}$</td>
<td>$\leftarrow$ RA$^{6:7}$</td>
</tr>
<tr>
<td>$RT^{8:11}$</td>
<td>$\leftarrow$ RA$^{10:11}$</td>
</tr>
<tr>
<td>$RT^{12:15}$</td>
<td>$\leftarrow$ RA$^{14:15}$</td>
</tr>
</tbody>
</table>
Multiply and Add

Required v 1.0

mpya rt,ra,rb,rc

For each of four word slots:
- The value in register RA is treated as a 16-bit signed integer and multiplied by the 16-bit signed value in register RB. The resulting product is added to the value in register RC.
- The result is placed in register RT.
- Overflows and carries are not detected.

Programming Note: The operands are right-aligned within the 32-bit field.

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|  t0 | ← | RA^{2:3} \times RB^{2:3} |
|  t1 | ← | RA^{6:7} \times RB^{6:7} |
|  t2 | ← | RA^{10:11} \times RB^{10:11} |
|  t3 | ← | RA^{14:15} \times RB^{14:15} |
| RTE:3 | ← | t0 + RC^{0:3} |
| RTE:7 | ← | t1 + RC^{4:7} |
| RTE:11 | ← | t2 + RC^{8:11} |
| RTE:15 | ← | t3 + RC^{12:15} |
### Multiply High

**Syntax:**

```
mpyh rt, ra, rb
```

**Instruction Set:**

```
0 1 1 1 1 0 0 1 0 1 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

For each of four word slots:

- The leftmost 16 bits of the value in register RA are shifted right by 16 bits and multiplied by the 16-bit value in register RB.
- The product is shifted left by 16 bits and placed in register RT. Bits shifted out at the left are discarded. Zeros are shifted in at the right.

<table>
<thead>
<tr>
<th>t0</th>
<th>← RA&lt;sup&gt;0:1&lt;/sup&gt; • RB&lt;sup&gt;2:3&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>← RA&lt;sup&gt;4:5&lt;/sup&gt; • RB&lt;sup&gt;6:7&lt;/sup&gt;</td>
</tr>
<tr>
<td>t2</td>
<td>← RA&lt;sup&gt;8:9&lt;/sup&gt; • RB&lt;sup&gt;10:11&lt;/sup&gt;</td>
</tr>
<tr>
<td>t3</td>
<td>← RA&lt;sup&gt;12:13&lt;/sup&gt; • RB&lt;sup&gt;14:15&lt;/sup&gt;</td>
</tr>
<tr>
<td>RT&lt;sup&gt;0:3&lt;/sup&gt;</td>
<td>← t&lt;sup&gt;0:3&lt;/sup&gt;</td>
</tr>
<tr>
<td>RT&lt;sup&gt;4:7&lt;/sup&gt;</td>
<td>← t&lt;sup&gt;1:3&lt;/sup&gt;</td>
</tr>
<tr>
<td>RT&lt;sup&gt;8:11&lt;/sup&gt;</td>
<td>← t&lt;sup&gt;2:3&lt;/sup&gt;</td>
</tr>
<tr>
<td>RT&lt;sup&gt;12:15&lt;/sup&gt;</td>
<td>← t&lt;sup&gt;3:3&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

**Programming Note:** This instruction can be used in conjunction with `mpyu` and Add Word (a) to perform a 32-bit multiply. A 32-bit multiply instruction, `mpy32` rt, ra, rb, can be emulated with the following instruction sequence:

```
mpyh t1, ra, rb
mpyh t2, rb, ra
mpyu t3, ra, rb
a rt, t1, t2
a rt, rt, t3
```
Multiply and Shift Right

Required v 1.0

mpys rt,ra,rb

<table>
<thead>
<tr>
<th></th>
<th>RA0:3</th>
<th>RB0:3</th>
<th>RA4:7</th>
<th>RB4:7</th>
<th>RA10:11</th>
<th>RB10:11</th>
<th>RA14:15</th>
<th>RB14:15</th>
<th>RT0:3</th>
<th>RT4:7</th>
<th>RT8:11</th>
<th>RT12:15</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>← RA2:3 • RB2:3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>← RA6:7 • RB6:7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>← RA10:11 • RB10:11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>← RA14:15 • RB14:15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT0:3</td>
<td>← RepLeftBit(t00:1,32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT4:7</td>
<td>← RepLeftBit(t10:1,32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT8:11</td>
<td>← RepLeftBit(t20:1,32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT12:15</td>
<td>← RepLeftBit(t30:1,32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of four word slots:

- The value in the rightmost 16 bits of register RA is multiplied by the value in the rightmost 16 bits of register RB.
- The leftmost 16 bits of the 32-bit product are placed in the rightmost 16 bits of register RT, with the sign bit replicated into the left 16 bits of the register.
Multiply High High

\[
\text{mpyhh} \quad rt, ra, rb
\]

For each of four word slots:

- The leftmost 16 bits in register RA are multiplied by the leftmost 16 bits in register RB.
- The 32-bit product is placed in register RT.

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
& & & \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
\hline
RB & RA & RT
\end{array}
\]

\[
\begin{array}{cccccccccccccccccc}
\hline
\end{array}
\]

\[
\begin{array}{ll}
RT^{0:3} & \leftarrow RA^{0:1} \times RB^{0:1} \\
RT^{4:7} & \leftarrow RA^{4:5} \times RB^{4:5} \\
RT^{8:11} & \leftarrow RA^{8:9} \times RB^{8:9} \\
RT^{12:15} & \leftarrow RA^{12:13} \times RB^{12:13} \\
\end{array}
\]
Multiply High High and Add

mpyhha rt,ra,rb

For each of four word slots:

- The leftmost 16 bits in register RA are multiplied by the leftmost 16 bits in register RB. The product is added to the value in register RT.
- The sum is placed in register RT.

\[
\begin{array}{ll}
RT_{0:3} & \leftarrow RA_{0:1} \times RB_{0:1} + RT_{0:3} \\
RT_{4:7} & \leftarrow RA_{4:5} \times RB_{4:5} + RT_{4:7} \\
RT_{8:11} & \leftarrow RA_{8:9} \times RB_{8:9} + RT_{8:11} \\
RT_{12:15} & \leftarrow RA_{12:13} \times RB_{12:13} + RT_{12:15}
\end{array}
\]
Multiply High High Unsigned

Required v 1.0

\[
\text{mpyhhu } \text{rt,ra,rb}
\]

For each of four word slots:

- The leftmost 16 bits in register RA are multiplied by the leftmost 16 bits in register RB, treating both operands as unsigned.
- The 32-bit product is placed in register RT.

\[
\begin{array}{cccccccccccccccccccccccccccc}
RT^{0:3} & \leftarrow & RA^{0:1} & \text{||} & RB^{0:1} \\
RT^{4:7} & \leftarrow & RA^{4:5} & \text{||} & RB^{4:5} \\
RT^{8:11} & \leftarrow & RA^{8:9} & \text{||} & RB^{8:9} \\
RT^{12:15} & \leftarrow & RA^{12:13} & \text{||} & RB^{12:13}
\end{array}
\]
Multiply High High Unsigned and Add

**mpyhhau** rt,ra,rb

For each of four word slots:

- The leftmost 16 bits in register RA are multiplied by the leftmost 16 bits in register RB, treating both operands as unsigned. The product is added to the value in register RT.
- The sum is placed in register RT.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;0:3</td>
<td>← RA&lt;0:1&gt;</td>
<td>*</td>
</tr>
<tr>
<td>RT&lt;4:7</td>
<td>← RA&lt;4:5&gt;</td>
<td>*</td>
</tr>
<tr>
<td>RT&lt;8:11</td>
<td>← RA&lt;8:9&gt;</td>
<td>*</td>
</tr>
<tr>
<td>RT&lt;12:15</td>
<td>← RA&lt;12:13&gt;</td>
<td>*</td>
</tr>
</tbody>
</table>
Count Leading Zeros

\[ \text{clz} \quad \text{rt,ra} \]

For each of four word slots:

- The number of zero bits to the left of the first ‘1’ bit in the operand in register RA is computed.
- The result is placed in register RT. If register RA is zero, the result is 32.

**Programming Note:** The result placed in register RT satisfies \( 0 \leq RT \leq 32 \). The value in register RT is zero, for example, if the corresponding slot in RA is a negative integer. The value in register RT is 32 if the corresponding slot in register RA is zero.

for j = 0 to 15 by 4
    t ← 0
    u ← RA[j:4]
    For m = 0 to 31
        If \( u_m = 1 \) then leave
        t ← t + 1
    end
    RT[j:4] ← t
end
Count Ones in Bytes

\( \text{cntb \ rt,ra} \)

For each of 16 byte slots:

- The number of bits in register RA whose value is ‘1’ is computed.
- The result is placed in register RT.

**Programming Note:** The result placed in register RT satisfies \( 0 \leq RT \leq 8 \). The value in register RT is zero, for example, if the value in RA is zero. The value in RT is 8 if the value in RA is -1.

```plaintext
for j = 0 to 15
  c = 0
  b ← RA_j
  For m = 0 to 7
    If b_m = 1 then c ← c + 1
  end
  RT_j ← c
end
```

(See also the *Form Select Mask for Bytes* instruction on page 85.)
The rightmost 16 bits of the preferred slot of register RA are used to create a mask in register RT by replicating each bit eight times. Bits in the operand are related to bytes in the result in a left-to-right correspondence.

```
s ← RA^{2:3}
for j = 0 to 15
    if sj = 0 then
        r_j ← 0x00
    else
        r_j ← 0xFF
end
RT ← r
```
Form Select Mask for Halfwords

The rightmost 8 bits of the preferred slot of register RA are used to create a mask in register RT by replicating each bit 16 times. Bits in the operand are related to halfwords in the result, in a left-to-right correspondence.

```
s ← RA^3
k = 0
for j = 0 to 7
    if s_j = 0 then
        r^k:2 ← 0x0000
    else
        r^k:2 ← 0xFFFF
    k = k + 2
end
RT ← r
```
Form Select Mask for Words

fsm rt,ra

0 0 1 1 0 1 1 0 0

RA

RT

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

The rightmost 4 bits of the preferred slot of register RA are used to create a mask in register RT by replicating each bit 32 times. Bits in the operand are related to words in the result in a left-to-right correspondence.

s ← RA_{28:31}
k = 0
for j = 0 to 3
  If sj = 0 then
    r_{k:4} ← 0
  else
    r_{k:4} ← F
  k = k + 4
end
RT ← r
Gather Bits from Bytes

Required v 1.0

gbb rt,ra

A 16-bit quantity is formed in the right half of the preferred slot of register RT by concatenating the rightmost bit in each byte of register RA. The leftmost 16 bits of register RT are set to zero, as are the remaining slots of register RT.

```
k = 0
s = 0
for j = 7 to 128 by 8
    s_k ← RA_j
    k = k + 1
end
RT_0:3 ← 0x0000 || s
RT_4:7 ← 0
RT_8:11 ← 0
RT_12:15 ← 0
```
Gather Bits from Halfwords

A 8-bit quantity is formed in the rightmost byte of the preferred slot of register RT by concatenating the rightmost bit in each halfword of register RA. The leftmost 24 bits of the preferred slot of register RT are set to zero, as are the remaining slots of register RT.

```
k ← 0
s ← 0x00
for j = 15 to 128 by 16
    s_k ← RA_j
    k ← k + 1
end
RT0:3 ← 0x000000 || s
RT4:7 ← 0
RT8:11 ← 0
RT12:15 ← 0
```
Gather Bits from Words

A 4-bit quantity is formed in the rightmost 4 bits of register RT by concatenating the rightmost bit in each word of register RA. The leftmost 28 bits of register RT are set to zero, as are the remaining slots of register RT.

\[
\begin{align*}
&k = 0 \\
&s = 0x0 \\
&\text{for } j = 31 \text{ to } 128 \text{ by } 32 \\
&\quad s_k \leftarrow RA_j \\
&\quad k \leftarrow k + 1 \\
&\text{end} \\
&RT^{0:3} \leftarrow 0x00000000 \parallel s \\
&RT^{4:7} \leftarrow 0 \\
&RT^{8:11} \leftarrow 0 \\
&RT^{12:15} \leftarrow 0
\end{align*}
\]
For each of 16 byte slots:

- The operand from register RA is added to the operand from register RB, and ‘1’ is added to the result. These additions are done without loss of precision.
- That result is shifted to the right by 1 bit and placed in register RT.

\[
\text{avgb } \text{rt,ra,rb}
\]

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \text{RB} & \text{RA} & \text{RT} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow
\end{array}
\]

\[
\text{for } j = 0 \text{ to } 15
\]

\[
R_{Tj} \leftarrow ((0x00 \| \text{RA}) + (0x00 \| \text{RB}) + 1)_{7:14}
\]

end
Absolute Differences of Bytes

\texttt{absdb rt,ra,rb}

\begin{center}
\begin{tabular}{cccc}
\hline
0 & 0 & 0 & 0  \\
0 & 0 & 1 & 1  \\
\hline
RB & RA & RT
\end{tabular}
\end{center}

For each of 16 byte slots:

- The operand in register RA is subtracted from the operand in register RB.
- The absolute value of the result is placed in register RT.

\textbf{Programming Note:} The operands are unsigned.

\begin{verbatim}
for j = 0 to 15
    if (RB\textsuperscript{j} >u RA\textsuperscript{j})
        RT\textsuperscript{j} ← RB\textsuperscript{j} - RA\textsuperscript{j}
    else
        RT\textsuperscript{j} ← RA\textsuperscript{j} - RB\textsuperscript{j}
end
\end{verbatim}
Sum Bytes into Halfwords

For each of four word slots:

- The 4 bytes in register RB are added, and the 16-bit result is placed in bytes 0 and 1 of register RT.
- The 4 bytes in register RA are added, and the 16-bit result is placed in bytes 2 and 3 of register RT.

**Programming Note:** The operands are unsigned.

| RT<sub>0:1</sub> | ← RB<sub>0</sub> + RB<sub>1</sub> + RB<sub>2</sub> + RB<sub>3</sub> |
| RT<sub>2:3</sub> | ← RA<sub>0</sub> + RA<sub>1</sub> + RA<sub>2</sub> + RA<sub>3</sub> |
| RT<sub>4:5</sub> | ← RB<sub>4</sub> + RB<sub>5</sub> + RB<sub>6</sub> + RB<sub>7</sub> |
| RT<sub>6:7</sub> | ← RA<sub>4</sub> + RA<sub>5</sub> + RA<sub>6</sub> + RA<sub>7</sub> |
| RT<sub>8:9</sub> | ← RB<sub>8</sub> + RB<sub>9</sub> + RB<sub>10</sub> + RB<sub>11</sub> |
| RT<sub>10:11</sub> | ← RA<sub>8</sub> + RA<sub>9</sub> + RA<sub>10</sub> + RA<sub>11</sub> |
| RT<sub>12:13</sub> | ← RB<sub>12</sub> + RB<sub>13</sub> + RB<sub>14</sub> + RB<sub>15</sub> |
| RT<sub>14:15</sub> | ← RA<sub>12</sub> + RA<sub>13</sub> + RA<sub>14</sub> + RA<sub>15</sub> |
### Extend Sign Byte to Halfword

**xsbh**  
**rt,ra**

For each of eight halfword slots:
- The sign of the byte in the right byte of the operand in register RA is propagated to the left byte.
- The resulting 16-bit integer is stored in register RT.

**Programming Note:** This is the only instruction that treats bytes as signed.

<table>
<thead>
<tr>
<th>RT</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>R^{0:1}</td>
<td>← RepLeftBit(RA^{1},16)</td>
</tr>
<tr>
<td>R^{2:3}</td>
<td>← RepLeftBit(RA^{3},16)</td>
</tr>
<tr>
<td>R^{4:5}</td>
<td>← RepLeftBit(RA^{5},16)</td>
</tr>
<tr>
<td>R^{6:7}</td>
<td>← RepLeftBit(RA^{7},16)</td>
</tr>
<tr>
<td>R^{8:9}</td>
<td>← RepLeftBit(RA^{9},16)</td>
</tr>
<tr>
<td>R^{10:11}</td>
<td>← RepLeftBit(RA^{11},16)</td>
</tr>
<tr>
<td>R^{12:13}</td>
<td>← RepLeftBit(RA^{13},16)</td>
</tr>
<tr>
<td>R^{14:15}</td>
<td>← RepLeftBit(RA^{15},16)</td>
</tr>
</tbody>
</table>
### Extend Sign Halfword to Word

**Required** v 1.0

The **xshw** instruction takes the sign of the right halfword of the operand in register RA and propagates it to the left halfword. The result is placed in register RT.

**Syntax:**

```
xshw rt, ra
```

**Example:**

```
0 1 0 1 0 1 1 0
```

- The sign of the right halfword in RA is propagated to the left halfword.
- The resulting 32-bit integer is placed in RT.

<table>
<thead>
<tr>
<th>RT0:3</th>
<th>← RepLeftBit(RA2:3,32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT4:7</td>
<td>← RepLeftBit(RA6:7,32)</td>
</tr>
<tr>
<td>RT8:11</td>
<td>← RepLeftBit(RA10:11,32)</td>
</tr>
<tr>
<td>RT12:15</td>
<td>← RepLeftBit(RA14:15,32)</td>
</tr>
</tbody>
</table>
Extend Sign Word to Doubleword

For each of two doubleword slots:

- The sign of the word in the right slot is propagated to the left word.
- The resulting 64-bit integer is stored in register RT.

\[
\begin{array}{c|c|c}
\text{RT}^0:7 & \leftarrow \text{RepLeftBit(RA}^4:7,64) \\
\text{RT}^8:15 & \leftarrow \text{RepLeftBit(RA}^{12:15},64)
\end{array}
\]
And

and rt, ra, rb

0 0 0 1 1 0 0 0 0 1

The values in register RA and register RB are logically ANDed. The result is placed in register RT.

\[
\begin{array}{c|c|c|c}
\hline
& RT^0:3 & RA^0:3 \& RB^0:3 \\
\hline
RT^4:7 & RA^4:7 \& RB^4:7 \\
\hline
RT^8:11 & RA^8:11 \& RB^8:11 \\
\hline
RT^{12:15} & RA^{12:15} \& RB^{12:15} \\
\hline
\end{array}
\]
And with Complement

**andc**  \( \text{rt,ra,rb} \)

<table>
<thead>
<tr>
<th>RT[0:3]</th>
<th>RA[0:3] &amp; (¬RB[0:3])</th>
</tr>
</thead>
</table>

The value in register RA is logically ANDed with the complement of the value in register RB. The result is placed in register RT.
And Byte Immediate

andbi rt, ra, value

For each of 16 byte slots, the rightmost 8 bits of the I10 field are ANDed with the value in register RA. The result is placed in register RT.

| b | ← I10 & 0x00FF |
| bbbb | ← b || b || b || b |
| RT<0:3> | ← RA<0:3> & bbbb |
| RT<4:7> | ← RA<4:7> & bbbb |
| RT<8:11> | ← RA<8:11> & bbbb |
| RT<12:15> | ← RA<12:15> & bbbb |
And Halfword Immediate

\texttt{andhi rt,ra,value}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The I10 field is extended to 16 bits by replicating its leftmost bit. The result is ANDed with the value in register RA.
- The 16-bit result is placed in register RT.

\[
\begin{array}{l}
\text{t} & \leftarrow \text{RepLeftBit(I10,16)} \\
\text{RT}^{0:1} & \leftarrow \text{RA}^{0:1} \& \text{t} \\
\text{RT}^{2:3} & \leftarrow \text{RA}^{2:3} \& \text{t} \\
\text{RT}^{4:5} & \leftarrow \text{RA}^{4:5} \& \text{t} \\
\text{RT}^{6:7} & \leftarrow \text{RA}^{6:7} \& \text{t} \\
\text{RT}^{8:9} & \leftarrow \text{RA}^{8:9} \& \text{t} \\
\text{RT}^{10:11} & \leftarrow \text{RA}^{10:11} \& \text{t} \\
\text{RT}^{12:13} & \leftarrow \text{RA}^{12:13} \& \text{t} \\
\text{RT}^{14:15} & \leftarrow \text{RA}^{14:15} \& \text{t} \\
\end{array}
\]
### And Word Immediate

**Syntax:**

```plaintext
andi rt, ra, value
```

#### Description:

- For each of four word slots:
  - The value of the I10 field is extended to 32 bits by replicating its leftmost bit. The result is ANDed with the contents of register RA.
  - The result is placed in register `RT`.

#### Table:

<table>
<thead>
<tr>
<th></th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t</code></td>
<td>← <code>RepLeftBit(I10, 32)</code></td>
</tr>
<tr>
<td><code>RT^0:3</code></td>
<td>← <code>RA^0:3 &amp; t</code></td>
</tr>
<tr>
<td><code>RT^4:7</code></td>
<td>← <code>RA^4:7 &amp; t</code></td>
</tr>
<tr>
<td><code>RT^8:11</code></td>
<td>← <code>RA^8:11 &amp; t</code></td>
</tr>
<tr>
<td><code>RT^12:15</code></td>
<td>← <code>RA^12:15 &amp; t</code></td>
</tr>
</tbody>
</table>
The values in register RA and register RB are logically ORed. The result is placed in register RT.

<table>
<thead>
<tr>
<th></th>
<th>RT&lt;sup&gt;0:3&lt;/sup&gt;</th>
<th>← RA&lt;sup&gt;0:3&lt;/sup&gt;</th>
<th>RB&lt;sup&gt;0:3&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RT&lt;sup&gt;4:7&lt;/sup&gt;</td>
<td>← RA&lt;sup&gt;4:7&lt;/sup&gt;</td>
<td>RB&lt;sup&gt;4:7&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>RT&lt;sup&gt;8:11&lt;/sup&gt;</td>
<td>← RA&lt;sup&gt;8:11&lt;/sup&gt;</td>
<td>RB&lt;sup&gt;8:11&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>RT&lt;sup&gt;12:15&lt;/sup&gt;</td>
<td>← RA&lt;sup&gt;12:15&lt;/sup&gt;</td>
<td>RB&lt;sup&gt;12:15&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
The value in register RA is ORed with the complement of the value in register RB. The result is placed in register RT.

\[
\begin{align*}
\text{RT}_0:3 &\leftarrow \text{RA}_0:3 \mid (\neg \text{RB}_0:3) \\
\text{RT}_4:7 &\leftarrow \text{RA}_4:7 \mid (\neg \text{RB}_4:7) \\
\text{RT}_8:11 &\leftarrow \text{RA}_8:11 \mid (\neg \text{RB}_8:11) \\
\text{RT}_{12:15} &\leftarrow \text{RA}_{12:15} \mid (\neg \text{RB}_{12:15})
\end{align*}
\]
Or Byte Immediate

<table>
<thead>
<tr>
<th>orbi</th>
<th>rt, ra, value</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbi</td>
<td>rt, ra, value</td>
</tr>
</tbody>
</table>

For each of 16 byte slots:
- The rightmost 8 bits of the I10 field are ORed with the value in register RA.
- The result is placed in register RT.

<table>
<thead>
<tr>
<th>b</th>
<th>← I10 &amp; 0x00FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbbb</td>
<td>← b</td>
</tr>
<tr>
<td>RT0:3</td>
<td>← RA0:3</td>
</tr>
<tr>
<td>RT4:7</td>
<td>← RA4:7</td>
</tr>
<tr>
<td>RT8:11</td>
<td>← RA8:11</td>
</tr>
<tr>
<td>RT12:15</td>
<td>← RA12:15</td>
</tr>
</tbody>
</table>
**Or Halfword Immediate**

**orhi**  \( rt, ra, value \)

<table>
<thead>
<tr>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The I10 field is extended to 16 bits by replicating its leftmost bit. The result is ORed with the value in register RA.
- The result is placed in register RT.

|     | \( t \) ← RepLeftBit(I10,16) | \( RT^{0:1} \) ← RA^{0:1} \( | t \) |
|-----|--------------------------------|----------------------------------|
|     | \( RT^{2:3} \) ← RA^{2:3} \( | t \) | \( RT^{4:5} \) ← RA^{4:5} \( | t \) |
|     | \( RT^{6:7} \) ← RA^{6:7} \( | t \) | \( RT^{8:9} \) ← RA^{8:9} \( | t \) |
|     | \( RT^{10:11} \) ← RA^{10:11} \( | t \) | \( RT^{12:13} \) ← RA^{12:13} \( | t \) |
|     | \( RT^{14:15} \) ← RA^{14:15} \( | t \) |
For each of four word slots:

- The I10 field is sign-extended to 32 bits and ORed with the contents of register RA.
- The result is placed in register RT.

\[
\begin{array}{c|c}
\text{t} & \leftarrow \text{RepLeftBit}(I_{10}, 32) \\
\text{RT}_{0:3} & \leftarrow \text{RA}_{0:3} \mid t \\
\text{RT}_{4:7} & \leftarrow \text{RA}_{4:7} \mid t \\
\text{RT}_{8:11} & \leftarrow \text{RA}_{8:11} \mid t \\
\text{RT}_{12:15} & \leftarrow \text{RA}_{12:15} \mid t \\
\end{array}
\]
The four words of RA are logically ORed. The result is placed in the preferred slot of register RT. The other three slots of the register are written with zeros.

<table>
<thead>
<tr>
<th>RT&lt;sup&gt;0:3&lt;/sup&gt;</th>
<th>← RA&lt;sup&gt;0:3&lt;/sup&gt;</th>
<th>RA&lt;sup&gt;4:7&lt;/sup&gt;</th>
<th>RA&lt;sup&gt;8:11&lt;/sup&gt;</th>
<th>RA&lt;sup&gt;12:15&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;sup&gt;4:15&lt;/sup&gt;</td>
<td>← 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The values in register RA and register RB are logically XORed. The result is placed in register RT.

<table>
<thead>
<tr>
<th>RT^0:3</th>
<th>← RA^0:3 ⊕ RB^0:3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT^4:7</td>
<td>← RA^4:7 ⊕ RB^4:7</td>
</tr>
<tr>
<td>RT^8:11</td>
<td>← RA^8:11 ⊕ RB^8:11</td>
</tr>
<tr>
<td>RT^12:15</td>
<td>← RA^12:15 ⊕ RB^12:15</td>
</tr>
</tbody>
</table>
### Exclusive Or Byte Immediate

**xorbi**

\[ \text{rt}, \text{ra}, \text{value} \]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>R10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

For each of 16 byte slots:
- The rightmost 8 bits of the R10 field are XORed with the value in register RA.
- The result is placed in register RT.

<table>
<thead>
<tr>
<th>b</th>
<th>← I10 &amp; 0x00FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbbb</td>
<td>← b</td>
</tr>
<tr>
<td>RT^0:3</td>
<td>← RA^0:3 ⊕ bbbb</td>
</tr>
<tr>
<td>RT^4:7</td>
<td>← RA^4:7 ⊕ bbbb</td>
</tr>
<tr>
<td>RT^8:11</td>
<td>← RA^8:11 ⊕ bbbb</td>
</tr>
<tr>
<td>RT^12:15</td>
<td>← RA^12:15 ⊕ bbbb</td>
</tr>
</tbody>
</table>
For each of eight halfword slots:

- The I10 field is extended to 16 bits by replicating the leftmost bit. The resulting value is XORed with the value in register RA.
- The 16-bit result is placed in register RT.

```
| t     | ← RepLeftBit(I10,16) |
| RT^0:1| ← RA^0:1 ⊕ t         |
| RT^2:3| ← RA^2:3 ⊕ t         |
| RT^4:5| ← RA^4:5 ⊕ t         |
| RT^6:7| ← RA^6:7 ⊕ t         |
| RT^8:9| ← RA^8:9 ⊕ t         |
| RT^10:11| ← RA^10:11 ⊕ t    |
| RT^12:13| ← RA^12:13 ⊕ t     |
| RT^14:15| ← RA^14:15 ⊕ t     |
```
Exclusive Or Word Immediate

\text{exclusive or word immediate}

\text{xori} \quad \text{rt,ra,value}

\begin{center}
\begin{tabular}{ccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{cccccccccc}
8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{cccccccccc}
28 & 29 & 30 & 31 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
RT0:3 & RT4:7 & RT8:11 & RT12:15 \\
\end{tabular}
\end{center}

For each of four word slots:

- The I10 field is sign-extended to 32 bits and XORed with the contents of register RA.
- The 32-bit result is placed in register RT.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{t} & \text{\textless{}\text{\textasciicircum} RepLeftBit(I10,32)} \\
\hline
\text{RT}^{0:3} & \text{\textless{} RA}^{0:3} \oplus \text{t} \\
\hline
\text{RT}^{4:7} & \text{\textless{} RA}^{4:7} \oplus \text{t} \\
\hline
\text{RT}^{8:11} & \text{\textless{} RA}^{8:11} \oplus \text{t} \\
\hline
\text{RT}^{12:15} & \text{\textless{} RA}^{12:15} \oplus \text{t} \\
\hline
\end{tabular}
\end{center}
Nand

For each of four word slots:

- The complement of the AND of the bit in register RA and the bit in register RB is placed in register RT.

<table>
<thead>
<tr>
<th>RT&lt;sup&gt;0:3&lt;/sup&gt;</th>
<th>← ¬(RA&lt;sup&gt;0:3&lt;/sup&gt; &amp; RB&lt;sup&gt;0:3&lt;/sup&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;sup&gt;4:7&lt;/sup&gt;</td>
<td>← ¬(RA&lt;sup&gt;4:7&lt;/sup&gt; &amp; RB&lt;sup&gt;4:7&lt;/sup&gt;)</td>
</tr>
<tr>
<td>RT&lt;sup&gt;8:11&lt;/sup&gt;</td>
<td>← ¬(RA&lt;sup&gt;8:11&lt;/sup&gt; &amp; RB&lt;sup&gt;8:11&lt;/sup&gt;)</td>
</tr>
<tr>
<td>RT&lt;sup&gt;12:15&lt;/sup&gt;</td>
<td>← ¬(RA&lt;sup&gt;12:15&lt;/sup&gt; &amp; RB&lt;sup&gt;12:15&lt;/sup&gt;)</td>
</tr>
</tbody>
</table>
For each of four word slots:

- The values in register RA and register RB are logically ORed.
- The result is complemented and placed in register RT.

| RT<0:3> | ← ¬(RA<0:3> | RB<0:3>) |
| RT<4:7> | ← ¬(RA<4:7> | RB<4:7>) |
| RT<8:11> | ← ¬(RA<8:11> | RB<8:11>) |
| RT<12:15> | ← ¬(RA<12:15> | RB<12:15>) |
For each of four word slots:

- If the bit in register RA and register RB are the same, the result is ‘1’; otherwise, the result is ‘0’.
- The result is placed in register RT.

<table>
<thead>
<tr>
<th>RT&lt;sub&gt;0:3&lt;/sub&gt;</th>
<th>← RA&lt;sub&gt;0:3&lt;/sub&gt; ⊕ (¬RB&lt;sub&gt;0:3&lt;/sub&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;sub&gt;4:7&lt;/sub&gt;</td>
<td>← RA&lt;sub&gt;4:7&lt;/sub&gt; ⊕ (¬RB&lt;sub&gt;4:7&lt;/sub&gt;)</td>
</tr>
<tr>
<td>RT&lt;sub&gt;8:11&lt;/sub&gt;</td>
<td>← RA&lt;sub&gt;8:11&lt;/sub&gt; ⊕ (¬RB&lt;sub&gt;8:11&lt;/sub&gt;)</td>
</tr>
<tr>
<td>RT&lt;sub&gt;12:15&lt;/sub&gt;</td>
<td>← RA&lt;sub&gt;12:15&lt;/sub&gt; ⊕ (¬RB&lt;sub&gt;12:15&lt;/sub&gt;)</td>
</tr>
</tbody>
</table>
A result is formed by using bits from RC to choose corresponding bits either from RA or RB.

- If the bit in register RC is ‘0’, then select the bit from register RA; otherwise, select the bit from register RB.
- The selected bits are placed in register RT.

\[
\text{RT}^{0:15} \leftarrow \text{RC}^{0:15} \& \text{RB}^{0:15} \mid (\neg \text{RC}^{0:15}) \& \text{RA}^{0:15}
\]
Shuffle Bytes

`shufb rt, ra, rb, rc`

```
 1 0 1 1
↓ ↓ ↓ ↓
0 1 2 3 4 5 6 7 8 9
```

Registers RA and RB are logically concatenated with the least-significant bit of RA adjacent to the most-significant bit of RB. The bytes of the resulting value are considered to be numbered from 0 to 31.

For each byte slot in registers RC and RT:
- The value in register RC is examined, and a result byte is produced as shown in Table 5-1.
- The result byte is inserted into register RT.

**Table 5-1. Binary Values in Register RC and Byte Results**

<table>
<thead>
<tr>
<th>Value in Register RC (Expressed in Binary)</th>
<th>Result Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>10xxxxxx</td>
<td>0x00</td>
</tr>
<tr>
<td>110xxxxx</td>
<td>0xFF</td>
</tr>
<tr>
<td>111xxxxx</td>
<td>0x80</td>
</tr>
<tr>
<td>Otherwise</td>
<td>The byte of the concatenated register addressed by the rightmost 5 bits of register RC</td>
</tr>
</tbody>
</table>

```
Rconcat ← RA || RB
for j = 0 to 15
    b ← RCj
    if b0:1 = 0b10 then c ← 0x00
    else if b0:2 = 0b110 then c ← 0xFF
    else if b0:2 = 0b111 then c ← 0x80
    else
        b ← b & 0x1F;
        c ← Rconcatb;
    end
    RTj ← c
end
```
6. Shift and Rotate Instructions

This section describes the SPU shift and rotate instructions.
Shift Left Halfword

```
shlh rt,ra,rb
```

For each of eight halfword slots:

- The contents of register RA are shifted to the left according to the count in bits 11 to 15 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.
- Bits shifted out of the left end of the halfword are discarded; zeros are shifted in at the right.

**Note:** Each halfword slot has its own independent shift amount.

```
for j = 0 to 15 by 2
    s ← RBj/2 & 0x001F
    t ← RAj/2
    for b = 0 to 15
        if b + s < 16 then r_b ← t_b + s
        else r_b ← 0
    end
    RTj/2 ← r
end
```
Shift Left Halfword Immediate

For each of eight halfword slots:

- The contents of register RA are shifted to the left according to the count in bits 13 to 17 of the I7 field.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.
- Bits shifted out of the left end of the halfword are discarded; zeros are shifted in at the right.

```
s ← RepLeftBit(I7, 16) & 0x001F
for j = 0 to 15 by 2
    t ← RA::2
    for b = 0 to 15
        if b + s < 16 then
            fb ← tb + s
        else
            fb ← 0
        end
    end
    RT::2 ← r
end
```
Shift Left Word

For each of four word slots:

- The contents of register RA are shifted to the left according to the count in bits 26 to 31 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 31, the result is zero.
- Bits shifted out of the left end of the word are discarded; zeros are shifted in at the right.

Note: Each word slot has its own independent shift amount.

```
for j = 0 to 15 by 4
    s ← RBj::4 & 0x0000003F
    t ← RAj::4
    for b = 0 to 31
        if b + s < 32 then
            rb ← tb + s
        else
            rb ← 0
        end
    end
    RTj::4 ← r
end
```
Shift Left Word Immediate

required v 1.0

<table>
<thead>
<tr>
<th>shli</th>
<th>rt,ra,value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0111 011 011</td>
</tr>
</tbody>
</table>

For each of four word slots:

- The contents of register RA are shifted to the left according to the count in bits 12 to 17 of the I7 field.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 31, the result is zero.
- Bits shifted out of the left end of the word are discarded; zeros are shifted in at the right.

```
s ← RepLeftBit(I7,32) & 0x0000003F
for j = 0 to 15 by 4
    t ← RAj::4
    for b = 0 to 31
        if b + s < 32 then
            rb ← tb + s
        else
            rb ← 0
        end
    end
    RTj::4 ← r
end
```
Shift Left Quadword by Bits

\[
\text{shlqbi \hspace{1cm} rt,ra,rb}
\]

The contents of register RA are shifted to the left according to the count in bits 29 to 31 of the preferred slot of register RB. The result is placed in register RT. A shift of up to 7 bit positions is possible.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bits shifted out of the left end of the register are discarded, and zeros are shifted in at the right.

\[
s \leftarrow \text{RB}_{29:31} \\
\text{for } b = 0 \text{ to } 127 \\
\quad \text{if } b + s < 128 \text{ then } r_b \leftarrow \text{RA}_b + s \\
\quad \text{else } r_b \leftarrow 0 \\
\text{end} \\
\text{RT} \leftarrow r
\]
Shift Left Quadword by Bits Immediate

\[ \text{shlqbii} \quad \text{rt,ra,value} \]

The contents of register RA are shifted to the left according to the count in bits 15 to 17 of the I7 field. The result is placed in register RT. A shift of up to 7 bit positions is possible.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bits shifted out of the left end of the register are discarded, and zeros are shifted in at the right.

\[
\begin{align*}
  s & \leftarrow I7 \& 0x07 \\
  \text{for } b = 0 \text{ to } 127 & \\
  & \quad \text{if } b + s < 128 \text{ then } r_b \leftarrow RA_{b+s} \\
  & \quad \text{else} \quad r_b \leftarrow 0 \\
  \text{end} \\
  \text{RT} & \leftarrow r
\end{align*}
\]
Shift Left Quadword by Bytes

shlqby rt,ra,rb

The bytes of register RA are shifted to the left according to the count in bits 27 to 31 of the preferred slot of register RB. The result is placed in register RT.

If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.

Bytes shifted out of the left end of the register are discarded, and bytes of zeros are shifted in at the right.

```
s ← RB[27:31]
for b = 0 to 15
    if b + s < 16 then rb ← RA[b] + s
    else rb ← 0
end
RT ← r
```
Shift Left Quadword by Bytes Immediate

\texttt{shlqbyi rt,ra,value}

The bytes of register RA are shifted to the left according to the count in bits 13 to 17 of the I7 field. The result is placed in register RT.

If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.

Bytes shifted out of the left end of the register are discarded, and zero bytes are shifted in at the right.

\begin{verbatim}
s ← I7 & 0x1F
for b = 0 to 15
    if b + s < 16 then \( r^b \leftarrow RA^b + s \)
    else \( r^b \leftarrow 0 \)
end
RT ← r
\end{verbatim}
Shift Left Quadword by Bytes from Bit Shift Count

```
shlqbybi rt,ra,rb
```

```
  0 0 1 1 1 0 0 1 1 1 1
  ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
  0 1 2 3 4 5 6 7 8 9

RB      RA      RT
0123456789 1011121314 1516171819 2021222324 2526272829 3031
```

The bytes of register RA are shifted to the left according to the count in bits 24 to 28 of the preferred slot of register RB. The result is placed in register RT.

If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.

Bytes shifted out of the left end of the register are discarded, and bytes of zeros are shifted in at the right.

```
s ← RB_{24:28}
for b = 0 to 15
    if b + s < 16 then rb ← RA_b + s
    else r_b ← 0x00
end
RT ← r
```
Rotate Halfword

```
roth rt,ra,rb
```

For each of eight halfword slots:

- The contents of register RA are rotated to the left according to the count in bits 12 to 15 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT.
- Bits rotated out of the left end of the halfword are rotated in at the right end.

**Note:** Each halfword slot has its own independent rotate amount.

```
for j = 0 to 15 by 2
    s ← RBj::2 & 0x000F
    t ← RAj::2
    for b = 0 to 15
        if b + s < 16 then
            fb ← tb + s
        else
            fb ← tb + s - 16
        end
    end
    RTj::2 ← r
end
```
Rotate Halfword Immediate

rothi rt,ra,value

For each of eight halfword slots:

- The contents of register RA are rotated to the left according to the count in bits 14 to 17 of the I7 field.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT.
- Bits rotated out of the left end of the halfword are rotated in at the right end.

```
s ← RepLeftShift(I7,16) & 0x000F
for j = 0 to 15 by 2
    t ← RA[j]::2
    for b = 0 to 15
        if b + s < 16 then
            rb ← tb + s
        else
            rb ← tb + s - 16
        end
    end
    RT[j]::2 ← r
end
```
Rotate Word

rot rt,ra,rb

For each of four word slots:

- The contents of register RA are rotated to the left according to the count in bits 27 to 31 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT.
- Bits rotated out of the left end of the word are rotated in at the right end.

\[
\text{for } j = 0 \text{ to } 15 \text{ by } 4
\]

\[
\begin{align*}
    s &\leftarrow \text{RB}_{j+4} \& 0x0000001F \\
    t &\leftarrow \text{RA}_{j+4} \\
    \text{for } b = 0 \text{ to } 31 \\
    &\quad \text{if } b + s < 32 \text{ then } f_b \leftarrow t_b + s \\
    &\quad \text{else } f_b \leftarrow t_b + s - 32 \\
    \end{align*}
\]

\[
\text{RT}_{j+4} \leftarrow f
\]

end
Rotate Word Immediate

roti rt,ra,value

For each of four word slots:

- The contents of register RA are rotated to the left according to the count in bits 13 to 17 of the I7 field.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT.
- Bits rotated out of the left end of the word are rotated in at the right end.

\[
S \leftarrow \text{RepLeftBit}(I7,32) \& 0x0000001F
\]

for \( j = 0 \) to \( 15 \) by \( 4 \)

\[
t \leftarrow RA^{j:4}
\]

for \( b = 0 \) to \( 31 \)

if \( b + s < 32 \) then

\[
f_b \leftarrow t_b + s
\]

else

\[
f_b \leftarrow t_b + s - 32
\]

end

RT\(^{j:4}\) \(\leftarrow r\)

end
Rotate Quadword by Bytes

```
rotqby rt,ra,rb
```

The bytes in register RA are rotated to the left according to the count in the rightmost 4 bits of the preferred slot of register RB. The result is placed in register RT. Rotation of up to 15 byte positions is possible.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bytes rotated out of the left end of the register are rotated in at the right.

```
s ← RB28:31
for b = 0 to 15
    if b + s < 16 then rb ← RAb + s
    else rb ← RAb + s - 16
end
RT ← r
```
Rotate Quadword by Bytes Immediate

\[ \text{rotqbyi rt,ra,value} \]

The bytes in register RA are rotated to the left according to the count in the rightmost 4 bits of the I7 field. The result is placed in register RT. Rotation of up to 15 byte positions is possible.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bytes rotated out of the left end of the register are rotated in at the right.

\[
\begin{align*}
  s &\leftarrow I_7_{14:17} \\
  \text{for } b = 0 \text{ to } 15 & \\
  &\quad \text{if } b + s < 16 \text{ then } r^b \leftarrow RA^b + s \\
  &\quad \text{else } r^b \leftarrow RA^b + s - 16 \\
  \text{end} & \\
  RT &\leftarrow r
\end{align*}
\]
### Rotate Quadword by Bytes from Bit Shift Count

**Required** v 1.0

#### rotqbybi

```markdown
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
```

#### rb | ra | rt

| 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 |

The bytes of register RA are rotated to the left according to the count in bits 25 to 28 of the preferred slot of register RB. The result is placed in register RT.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bytes rotated out of the left end of the register are rotated in at the right.

```plaintext
s ← RB_{24:28}
for b = 0 to 15
    if b + s < 16 then
        r^b ← RA^b + s
    else
        r^b ← RA^b + (s - 16)
end
RT ← r
```
### Rotate Quadword by Bits

**Syntax:**

\[ \text{rotqbi \ rt,ra,rb} \]

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>RB</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

The contents of register RA are rotated to the left according to the count in bits 29 to 31 of the preferred slot of register RB. The result is placed in register RT. Rotation of up to 7 bit positions is possible.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bits rotated out at the left end of the register are rotated in at the right.

```plaintext
s ← RB_{29:31}
for b = 0 to 127
    if b + s < 128 then \( r_b ← RA_b + s \)
    else \( r_b ← RA_b + s - 128 \)
end
RT ← r
```
Rotate Quadword by Bits Immediate

```
rotqbi
rt,ra,value
```

The contents of register RA are rotated to the left according to the count in bits 15 to 17 of the I7 field. The result is placed in register RT. Rotation of up to 7 bit positions is possible.

If the count is zero, the contents of register RA are copied unchanged into register RT.

Bits rotated out at the left end of the register are rotated in at the right.

```
s ← l_{4:6}
for b = 0 to 127
  if b + s < 128 then rb ← RA_b + s
  else r_b ← RA_b + s - 128
end
RT ← r
```
Rotate and Mask Halfword

For each of eight halfword slots:

- The shift_count is (0 - RB) modulo 32.
- If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bits, with zero fill at the left.
- Otherwise, RT is set to zero.

Note: Each halfword slot has its own independent rotate amount.

```
for j = 0 to 15 by 2
    s ← (0 - RBj::2) & 0x001F
    t ← RAj::2
    for b = 0 to 15
        if b ≥ s then
            rb ← tb - s
        else
            rb ← 0
        end
    end
    RTj::2 ← rb
end
```

Programming Note: The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions is the two’s complement of the right shift amount. Thus, to shift right logically the contents of R2 by the number of bits given in R1, the following sequence could be used:

```
sfi r3,r1,0    Form two's complement
rothm r4,r2,r3 Rotate, then mask
```

For the immediate forms of these instructions, the formation of the two’s complement shift quantity can be performed during assembly or compilation.
Rotate and Mask Halfword Immediate

\textbf{rothmi} \quad \textbf{rt,ra,value}

\begin{center}
\begin{tabular}{ccccccccccccccccccccccc}
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & I7 & RA & RT \\
\hline
\hline
\end{tabular}
\end{center}

For each of eight halfword slots:

- The \textit{shift\_count} is \((0 - I7) \mod 32\).
- If the \textit{shift\_count} is less than 16, then RT is set to the contents of RA shifted right \textit{shift\_count} bits, with zero fill at the left.
- Otherwise, RT is set to zero.

\begin{verbatim}
s ← (0 - RepLeftBit(I7,32)) & 0x0000001F
for j = 0 to 15 by 2
  t ← RAj::2
  for b = 0 to 15
    if b>s then r_b ← t_b-s
    else r_b ← 0
  end
  RTj::2 ← r
end
\end{verbatim}

\textbf{Programming Note:} The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions is the two’s complement of the right shift amount. Thus, to shift right logically the contents of R2 by the number of bits given in R1, the following sequence could be used:

\begin{verbatim}
 sfi r3,r1,0 Form two's complement
 rotm r4,r2,r3 Rotate, then mask
\end{verbatim}

For the immediate forms of these instructions, the formation of the two’s complement shift quantity can be performed during assembly or compilation.
Rotate and Mask Word

### Required v 1.0

#### rotm rt,ra,rb

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB</td>
<td>RA</td>
<td>RT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of four word slots:

- The shift_count is \((0 - RB) \ mod \ 64\).  
- If the shift_count is less than 32, then RT is set to the contents of RA shifted right shift_count bits, with zero fill at the left.  
- Otherwise, RT is set to zero.

```plaintext
for j = 0 to 15 by 4
    s ← \((0 - RB^j) \ mod \ 64\) & 0x0000003F  
    t ← RA^j
    for b = 0 to 31
        if b ≥ s then r_b ← t_b - s
        else r_b ← 0
    end
    RT^j ← r
end
```

**Programming Note:** The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions is the two’s complement of the right shift amount. Thus, to shift right logically the contents of R2 by the number of bits given in R1, the following sequence could be used:

```plaintext
sfi r3,r1,0 Form two's complement
rotm r4,r2,r3 Rotate, then mask
```

For the immediate forms of these instructions, the formation of the two’s complement shift quantity can be performed during assembly or compilation.
Rotate and Mask Word Immediate

For each of four word slots:

- The shift_count is $(0 - I7)$ modulo 64.
- If the shift_count is less than 32, then RT is set to the contents of RA shifted right shift_count bits, with zero fill at the left.
- Otherwise, RT is set to zero.

```
s ← (0 - RepLeftBit(I7,32)) & 0x0000003F
for j = 0 to 15 by 4
    t ← RAj::4
    for b = 0 to 31
        if b ≥ s then
            rb ← tb - s
        else
            rb ← 0
        end
    end
    RTj::4 ← r
end
```

Programming Note: The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions is the two’s complement of the right shift amount. Thus, to shift right logically the contents of R2 by the number of bits given in R1, the following sequence could be used:

```
sfi r3,r1,0    Form two’s complement
rotm r4,r2,r3 Rotate, then mask
```

For the immediate forms of these instructions, the formation of the two’s complement shift quantity can be performed during assembly or compilation.
Rotate and Mask Quadword by Bytes

rotqmby rt, ra, rb

The shift_count is \( 0 - \) the preferred word of RB modulo 32. If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bytes, filling at the left with 0x00 bytes. Otherwise, RT is set to zero.

\[
s \leftarrow (0 - \text{RB}_{27:31}) \& 0x1F
\]

for \( b = 0 \) to 15

\[
\begin{align*}
\text{if } b & \geq s \text{ then } r^b \leftarrow t^b - s \\
\text{else } r^b & \leftarrow 0x00
\end{align*}
\]

end

RT \leftarrow r
The shift_count is \((0 - I7)\) modulo 32. If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bytes, filling at the left with 0x00 bytes. Otherwise, all bytes of RT are set to 0x00.

\[
\begin{align*}
s &\leftarrow (0 - I7) & \& 0x1F \\
\text{for } b = 0 \text{ to } 15 \\
&\quad \text{if } b \geq s \text{ then } \quad r_b &\leftarrow t^b \cdot s \\
&\quad \text{else } &\leftarrow 0x00 \\
\text{end}
\end{align*}
\]

\[
RT \leftarrow r
\]
The shift_count is (0 minus bits 24 to 28 of RB) modulo 32. If the shift_count is less than 16, then RT is set to the contents of RA, which is shifted right shift_count bytes, and filled at the left with 0x00 bytes. Otherwise, all bytes of RT are set to 0x00.

\[
s \leftarrow (0 - RB_{24:28}) \& 0x1F
\]

for b = 0 to 15

\[
\text{if } b \geq s \text{ then } r^b \leftarrow RA^b - s
\]

\[
\text{else } r^b \leftarrow 0x00
\]

end
The shift_count is \((0 - \text{the preferred word of RB}) \mod 8\). RT is set to the contents of RA, shifted right by shift_count bits, filling at the left with zero bits.

```
s ← (0 · RB_{29:31}) \& 0x07
for b = 0 to 127
  if b ≥ s then  r_b ← r_b - s
  else          r_b ← 0
end
RT ← r
```
The shift_count is \((0 - I7)\) modulo 8. RT is set to the contents of RA, shifted right by shift_count bits, filling at the left with zero bits.

\[
s \leftarrow (0 - I7) \& 0x07
\]

for \(b = 0\) to \(127\)

    if \(b \geq s\) then
        \(r_b \leftarrow t_b - s\)
    else
        \(r_b \leftarrow 0\)

end

RT \leftarrow r
Rotate and Mask Algebraic Halfword

For each of eight halfword slots:

- The shift_count is \((0 - RB)\) modulo 32.
- If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bits, replicating bit 0 (of the halfword) at the left.
- Otherwise, all bits of this halfword of RT are set to bit 0 of this halfword of RA.

**Note:** Each halfword slot has its own independent rotate amount.

```plaintext
for j = 0 to 15 by 2
    s ← \((0 - RB^{j/2})\) & 0x001F
    t ← RA^{j/2}
    for b = 0 to 15
        if b ≥ s then
            rb ← tb - s
        else
            rb ← t0
        end
    end
    RT^{j/2} ← r
end
```
Rotate and Mask Algebraic Halfword Immediate  Required  v 1.0

`rotmahi rt,ra,value`

For each of eight halfword slots:

- The `shift_count` is $(0 - I7) \mod 32$.
- If the `shift_count` is less than 16, then `RT` is set to the contents of `RA` shifted right `shift_count` bits, replicating bit 0 (of the halfword) at the left.
- Otherwise, all bits of this halfword of `RT` are set to bit 0 of this halfword of `RA`.

```
s ← (0 - RepLeftBit(I7,16)) & 0x001F
for j = 0 to 15 by 2
    t ← RA^j/2
    for b = 0 to 15
        if b ≥ s then
            rb ← tb - s
        else
            rb ← t0
        end
    end
    RT^j/2 ← r
end
```
Rotate and Mask Algebraic Word

\texttt{rotma \ rt,ra,rb}

\begin{tabular}{cccc}
0 & 0 & 0 & 0  \\
0 & 1 & 1 & 0  \\
0 & 1 & 1 & 0  \\
0 & 1 & 1 & 1  \\
\end{tabular}

For each of four word slots:

- The shift\_count is (0 - RB) modulo 64.
- If the shift\_count is less than 32, then RT is set to the contents of RA shifted right shift\_count bits, replicating bit 0 (of the word) at the left.
- Otherwise, all bits of this word of RT are set to bit 0 of this word of RA.

\begin{Verbatim}
for j = 0 to 15 by 4
  s ← (0 - RB\textsubscript{j}::4) & 0x0000003F
  t ← RA\textsubscript{j}::4
  for b = 0 to 31
    if b ≥ s then r\_b ← t\_b - s
    else r\_b ← t\_0
  end
  RT\textsubscript{j}::4 ← r
end
\end{Verbatim}
Rotate and Mask Algebraic Word Immediate

Required v 1.0

For each of four word slots:

- The shift_count is \((0 - I7)\) modulo 64.
- If the shift_count is less than 32, then RT is set to the contents of RA shifted right shift_count bits, replicating bit 0 (of the word) at the left.
- Otherwise, all bits of this word of RT are set to bit 0 of this word of RA.

s ← \((0 - \text{RepLeftBit}(I7, 32))\) & \(0x0000003F\)
for j = 0 to 15 by 4
    t ← RA\(_{j:4}\)
    for b = 0 to 31
        if \(b \geq s\) then \(r_b ← t_b - s\)
        else \(r_b ← t_0\)
    end
    RT\(_{j:4} ← r\)
end
7. Compare, Branch, and Halt Instructions

This section lists and describes the SPU compare, branch, and halt instructions. For more information about the SPU interrupt facility, see Section 12 on page 251.

Conditional branch instructions operate by examining a value in a register, rather than by accessing a specialized condition code register. The value is taken from the preferred slot. It is usually set by a compare instruction.

Compare instructions perform a comparison of the values in two registers or a value in a register and an immediate value. The result is indicated by setting into the target register a result value that is the same width as the register operands. If the comparison condition is met, the value is all one bits; if not, the value is all zero bits.

Logical comparison instructions treat the operands as unsigned integers. Other compare instructions treat the operands as two’s complement signed integers.

A set of halt instructions is provided that stops execution when the tested condition is met. These are intended to be used, for example, to check addresses or subscript ranges in situations where failure to meet the condition is regarded as a serious error. The stop that occurs is not precise; as a result, execution can generally not be restarted.

Floating-point compare instructions are listed in Section 9 Floating-Point Instructions on page 195 with the other floating-point instructions.
Halt If Equal

The value in the preferred slot of register RA is compared with the value in the preferred slot of register RB. If the values are equal, execution of the program stops at or after the halt.

**Programming Note:** RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.

\[
\text{heq ra,rb}
\]

If \( RA^{0:3} = RB^{0:3} \) then

Stop after executing zero or more instructions after the halt.
Halt If Equal Immediate

**heqi** ra,symbol

The value in the I10 field is extended to 32 bits by replicating the leftmost bit. The result is compared to the value in the preferred slot of register RA. If the value from register RA is equal to the immediate value, execution of the SPU program stops at or after the halt instruction.

**Programming Note:** RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.

If RA0:3 = RepLeftBit(I10,32) then

Stop after executing zero or more instructions after the halt.
Halt If Greater Than

Required v 1.0

The value in the preferred slot of register RA is algebraically compared with the value in the preferred slot of register RB. If the value from register RA is greater than the RB value, execution of the SPU program stops at or after the halt instruction.

**Programming Note:** RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.

If RA\(^{0:3}\) > RB\(^{0:3}\) then
Stop after executing zero or more instructions after the halt.
Halt If Greater Than Immediate

Required v 1.0

The value in the I10 field is extended to 32 bits by replicating the leftmost bit. The result is algebraically compared to the value in the preferred slot of register RA. If the value from register RA is greater than the immediate value, execution of the SPU program stops at or after the halt instruction.

**Programming Note:** RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.

If $RA^{0:3} > \text{RepLeftBit}(I10,32)$ then
Stop after executing zero or more instructions after the halt.
Halt If Logically Greater Than

hlgt ra,rb

The value in the preferred slot of register RA is logically compared with the value in the preferred slot of register RB. If the value from register RA is greater than the value from register RB, execution of the SPU program stops at or after the halt instruction.

**Programming Note:** RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.

If \( \text{RA}_{0:3} > \text{u} \text{RB}_{0:3} \) then

Stop after executing zero or more instructions after the halt.
Halt If Logically Greater Than Immediate

`hlgti ra,symbol`

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value in the I10 field is extended to 32 bits by replicating the leftmost bit. The result is logically compared to the value in the preferred slot of register RA. If the value from register RA is logically greater than the immediate value, execution of the SPU program stops at or after the halt instruction.

**Programming Note:** RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.

If $RA_{0:3} >\text{RepLeftBit}(I10,32)$ then

Stop after executing zero or more instructions after the halt.
Compare Equal Byte

**ceqb**

```
rt, ra, rb
```

For each of 16 byte slots:

- The operand from register RA is compared with the operand from register RB. If the operands are equal, a result of all one bits (true) is produced. If they are unequal, a result of all zero bits (false) is produced.
- The 8-bit result is placed in register RT.

```
for i = 0 to 15
  If RA^i = RB^i then RT^i ← 0xFF
  else RT^i ← 0x00
end
```
Compare Equal Byte Immediate

\textbf{ceqbi} \quad \text{rt,ra,value}

\begin{align*}
0 & 1 & 1 & 1 & 1 & 1 & 0 & I10 & RA & RT \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow
\end{align*}

\begin{align*}
\end{align*}

For each of 16 byte slots:

- The value in the rightmost 8 bits of the I10 field is compared with the value in register RA. If the two values are equal, a result of all one bits (true) is produced. If they are unequal, a result of all zero bits (false) is produced.
- The 8-bit result is placed in register RT.

\begin{verbatim}
for i = 0 to 15
    If RA[i] = I10[2:9] then
        RT[i] ← 0xFF
    else
        RT[i] ← 0x00
end
\end{verbatim}
**Compare Equal Halfword**

**ceqh**  \( \text{rt,ra,rb} \)

For each of 8 halfword slots:

- The operand from register RA is compared with the operand from register RB. If the operands are equal, a result of all one bits (true) is produced. If they are unequal, a result of all zero bits (false) is produced.
- The 16-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 2
  If RA\(^{i-2}\) = RB\(^{i-2}\) then RT\(^{i-2}\) ← 0xFFFF
  else RT\(^{i-2}\) ← 0x0000
end
```
Compare Equal Halfword Immediate

**ceqhi**

```
rt, ra, value
```

<table>
<thead>
<tr>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The value in the I10 field is extended to 16 bits by replicating its leftmost bit and compared with the value in register RA. If the two values are equal, a result of all one bits (true) is produced. If they are unequal, a result of all zero bits (false) is produced.
- The 16-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 2
    if RA^i = RepLeftBit(I10,16) then RT^i ← 0xFFF
    else RT^i ← 0x0000
end
```
**Compare Equal Word**

```plaintext
ceq rt,ra,rb
```

For each of four word slots:

- The operand from register RA is compared with the operand from register RB. If the operands are equal, a result of all one bits (true) is produced. If they are unequal, a result of all zero bits (false) is produced.

- The 32-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 4
    if RA^i\_4 = RB^i\_4 then
        RT^i\_4 ← 0xFFFFFFFF
    else
        RT^i\_4 ← 0x00000000
end
```
Compare Equal Word Immediate

c eqi rt, ra, value

For each of four word slots:

- The I10 field is extended to 32 bits by replicating its leftmost bit and comparing it with the value in register RA. If the two values are equal, a result of all one bits (true) is produced. If they are unequal, a result of all zero bits (false) is produced.

- The 32-bit result is placed in register RT.

```
for i = 0 to 15 by 4
    if RA[i-4] = RepLeftBit(I10,32) then RT[i-4] ← 0xFFFFFFFF
    else RT[i-4] ← 0x00000000
end
```
Compare Greater Than Byte

### Required v 1.0

cgtb rt,ra,rb

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>RB</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

For each of 16 byte slots:

- The operand from register RA is algebraically compared with the operand from register RB. If the operand in register RA is greater than the operand in register RB, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 8-bit result is placed in register RT.

```
for i = 0 to 15
    If RA[i] > RB[i] then RT[i] ← 0xFF
    else RT[i] ← 0x00
end
```
**Compare Greater Than Byte Immediate**

**cgtbi**

rt,ra,value

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

For each of 16 byte slots:

- The value in the rightmost 8 bits of the I10 field is algebraically compared with the value in register RA. If the value in register RA is greater, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 8-bit result is placed in register RT.

```plaintext
for i = 0 to 15
    if RA^i > I10_{2:9} then
        RT^i ← 0xFF
    else
        RT^i ← 0x00
end
```
**Compare Greater Than Halfword**

**cgth**

```
0 1 0 0 1 0 0 0
```

**rt, ra, rb**

```
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
```

For each of 8 halfword slots:

- The operand from register RA is algebraically compared with the operand from register RB. If the operand in register RA is greater than the operand in register RB, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.
- The 16-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 2
    If RA^i > RB^i then
        RT^i ← 0xFFFF
    else
        RT^i ← 0x0000
end
```
Compare Greater Than Halfword Immediate

**cgthi**

```
rt,ra,value
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The value in the I10 field is extended to 16 bits and algebraically compared with the value in register RA. If the value in register RA is greater than the I10 value, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 16-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 2
    If RA^{i/2} > RepLeftBit(I10,16) then RT^{i/2} ← 0xFFFF
    else RT^{i/2} ← 0x0000
end
```
For each of four word slots:

- The operand from register RA is algebraically compared with the operand from register RB. If the operand in register RA is greater than the operand in register RB, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.
- The 32-bit result is placed in register RT.

```
for i = 0 to 15 by 4
    If RA[i:4] > RB[i:4] then
        RT[i:4] ← 0xFFFFFFFF
    else
        RT[i:4] ← 0x00000000
end
```
### Compare Greater Than Word Immediate

**Instruction**: cgti rt,ra,value

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

For each of four word slots:

- The value in the I10 field is extended to 32 bits by sign extension and algebraically compared with the value in register RA. If the value in register RA is greater than the I10 value, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.
- The 32-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 4
    If RA[i:4] > RepLeftBit(I10,32) then RT[i:4] ← 0xFFFFFFFF
    else RT[i:4] ← 0x00000000
end
```
Compare Logical Greater Than Byte

**clgtb**  **rt,ra,rb**

| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RB | RA | RT |

For each of 16 byte slots:

- The operand from register RA is logically compared with the operand from register RB. If the operand in register RA is logically greater than the operand in register RB, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 8-bit result is placed in register RT.

```plaintext
for i = 0 to 15
    If RA_i >^u RB_i then RT_i ← 0xFF
    else RT_i ← 0x00
end
```
Compare Logical Greater Than Byte Immediate

clgtbi rt, ra, value

for i = 0 to 15
    if RA<sub>i</sub> ><sub>u</sub> I10<sub>2:9</sub> then
        RT<sub>i</sub> ← 0xFF
    else
        RT<sub>i</sub> ← 0x00
end

For each of 16 byte slots:
- The value in the rightmost 8 bits of the I10 field is logically compared with the value in register RA. If the value in register RA is logically greater, a result of all one bits (true) is produced. Otherwise, a result of all zero (false) bits is produced.
- The 8-bit result is placed in register RT.
Compare Logical Greater Than Halfword

clgth rt,ra,rb

For each of eight halfword slots:

- The operand from register RA is logically compared with the operand from register RB. If the operand in register RA is logically greater than the operand in register RB, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 16-bit result is placed in register RT.

for i = 0 to 15 by 2
    If RA\_{i:2} >u RB\_{i:2} then RT\_{i:2} ← 0xFFFF
    else RT\_{i:2} ← 0x0000
end
Compare Logical Greater Than Halfword Immediate  

\[ \text{clgthi } \text{rt,ra,value} \]

<table>
<thead>
<tr>
<th></th>
<th>I10</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

For each of eight halfword slots:

- The value in the I10 field is extended to 16 bits by replicating the leftmost bit and logically compared with the value in register RA. If the value in register RA is logically greater than the I10 value, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 16-bit result is placed in register RT.

\[
\text{for } i = 0 \text{ to } 15 \text{ by } 2
\]
\[
\begin{align*}
\text{if } \text{RA}_i^{16} > \text{RepLeftBit(I10,16)} \text{ then } & \text{RT}_i^{16} \leftarrow 0xFFFF \\
\text{else } & \text{RT}_i^{16} \leftarrow 0x0000
\end{align*}
\]
Compare Logical Greater Than Word

\texttt{clgt rt,ra,rb}

For each of four word slots:

- The operand from register RA is logically compared with the operand from register RB. If the operand in register RA is logically greater than the operand in register RB, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.
- The 32-bit result is placed in register RT.

\begin{verbatim}
for i = 0 to 15 by 4
  if RA\text{\textsubscript{i+4}} >\text{u} RB\text{\textsubscript{i+4}} then
    RT\text{\textsubscript{i+4}} ← 0xFFFFFFFF
  else
    RT\text{\textsubscript{i+4}} ← 0x00000000
end
\end{verbatim}
The value in the I10 field is extended to 32 bits by sign extension and logically compared with the value in register RA. If the value in register RA is logically greater than the I10 value, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

The 32-bit result is placed in register RT.

For each of four word slots:

- The value in the I10 field is extended to 32 bits by sign extension and logically compared with the value in register RA. If the value in register RA is logically greater than the I10 value, a result of all one bits (true) is produced. Otherwise, a result of all zero bits (false) is produced.

- The 32-bit result is placed in register RT.

```plaintext
for i = 0 to 15 by 4
    If RA[i:i+3] > RepLeftBit(I10,32) then
        RT[i:i+4] ← 0xFFFFFFFF
    else
        RT[i:i+4] ← 0x00000000
end
```
Execution proceeds with the target instruction. The address of the target instruction is computed by adding the value of the I16 field, extended on the right with two zero bits with the result treated as a signed quantity, to the address of the Branch Relative instruction.

**Programming Note:** If the value of the I16 field is zero, an infinite one instruction loop is executed.

\[
\text{PC} \leftarrow (\text{PC} + \text{RepLeftBlit} (\text{I16} || 0b00, 32)) \& \text{LSLR}
\]
Branch Absolute

Execution proceeds with the target instruction. The address of the target instruction is the value of the I16 field, extended on the right with two zero bits and extended on the left with copies of the most-significant bit.

\[
\text{PC} \leftarrow \text{RepLeftBit}(I16 \ll 0b00,32) \& \text{LSLR}
\]
### Branch Relative and Set Link

**Required v 1.0**

**Instruction**

<table>
<thead>
<tr>
<th>brsl</th>
<th>rt,symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0 0 1 1 0 0 1 1 0</code></td>
<td></td>
</tr>
</tbody>
</table>

**Symbols**

<table>
<thead>
<tr>
<th>I16</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution proceeds with the target instruction. In addition, a link register is set.

The address of the target instruction is computed by adding the value of the I16 field, extended on the right with two zero bits with the result treated as a signed quantity, to the address of the Branch Relative and Set Link instruction.

The preferred slot of register RT is set to the address of the byte following the Branch Relative and Set Link instruction. The remaining slots of register RT are set to zero.

**Programming Note:** If the value of the I16 field is zero, an infinite one instruction loop is executed.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RT&lt;0:3</td>
<td>← (PC + 4) &amp; LSLR</td>
<td></td>
</tr>
<tr>
<td>RT&lt;4:15</td>
<td>← 0</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>← (PC + RepLeftBit(I16</td>
<td></td>
</tr>
</tbody>
</table>
Branch Absolute and Set Link

Execution proceeds with the target instruction. In addition, a link register is set.

The address of the target instruction is the value of the I16 field, extended on the right with two zero bits and extended on the left with copies of the most-significant bit.

The preferred slot of register RT is set to the address of the byte following the Branch Absolute and Set Link instruction. The remaining slots of register RT are set to zero.

| RT^0:3 | ← (PC + 4) & LSLR |
| RT^4:15 | ← 0 |
| PC | ← RepLeftBit(I16 || 0b00,32) & LSLR |
Execution proceeds with the instruction addressed by the preferred slot of register RA. The rightmost 2 bits of the value in register RA are ignored and assumed to be zero. Interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

PC ← RA^0.3 & LSLR & 0xFFFFFFFFC
if (E = 0 and D = 0) then interrupt enable status is not modified
else if (E = 1 and D = 0) then enable interrupts at target
else if (E = 0 and D = 1) then disable interrupts at target
else if (E = 1 and D = 1) then reserved
Interrupt Return

`iret ra`

```
0 0 1 1 0 1 0 1 0 1 0 / D E / / / / RA ///
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
```

Execution proceeds with the instruction addressed by SRR0. RA is considered to be a valid source whose value is ignored. Interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

PC ← SRR0

- if (E = 0 and D = 0) then interrupt enable status is not modified
- else if (E = 1 and D = 0) then enable interrupts at target
- else if (E = 0 and D = 1) then disable interrupts at target
- else if (E = 1 and D = 1) then reserved
Branch Indirect and Set Link if External Data

The external condition is examined. If it is false, execution continues with the next sequential instruction. If the external condition is true, the effective address of the next instruction is taken from the preferred word slot of register RA.

The address of the instruction following the `bisled` instruction is placed into the preferred word slot of register RT; the remainder of register RT is set to zero.

If the branch is taken, interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

```
u ← LSLR & (PC + 4)
t ← RA[0:3] & LSLR & 0xFFFFFFFF
RT[0:3] ← u
RT[4:15] ← 0

if (external event) then
    PC ← t
    if (E = 0 and D = 0) then interrupt enable status is not modified
    else if (E = 1 and D = 0) then enable interrupts at target
    else if (E = 0 and D = 1) then disable interrupts at target
    else if (E = 1 and D = 1) then reserved
    else
        PC ← u
```
### Branch Indirect and Set Link

**bisl**

```
0 0 1 1 0 1 0 0 0 1 / D E / / / RA RT
```

The effective address of the next instruction is taken from the preferred word slot of register RA, with the rightmost 2 bits assumed to be zero. The address of the instruction following the `bisl` instruction is placed into the preferred word slot of register RT. The remainder of register RT is set to zero. Interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

| t ← RA[0..3] & LSLR & 0xFFFFFFFC |
| u ← LSLR & (PC + 4) |
| RT[0..3] ← u |
| RT[4..15] ← 0x00 |

PC ← t

if (E = 0 and D = 0) then interrupt enable status is not modified
else if (E = 1 and D = 0) then enable interrupts at target
else if (E = 0 and D = 1) then disable interrupts at target
else if (E = 1 and D = 1) then reserved
Branch If Not Zero Word

Required v 1.0

brnz rt,symbol

Examine the preferred slot; if it is not zero, proceed with the branch target. Otherwise, proceed with the next instruction.

The address of the branch target is computed by appending two zero bits to the value of the I16 field, extending it on the left with copies of the most-significant bit, and adding it to the value of the instruction counter.

If $RT_{0:3} \neq 0$

\[
PC \leftarrow (PC + \text{RepLeftBit}(I16 || 0b00)) \& \text{LSLR} \& 0xFFFFFFFC
\]

else

\[
PC \leftarrow (PC + 4) \& \text{LSLR}
\]
Branch If Zero Word

**brz**

**rt,symbol**

Examine the preferred slot. If it is zero, proceed with the branch target. Otherwise, proceed with the next instruction.

The address of the branch target is computed by appending two zero bits to the value of the I16 field, extending it on the left with copies of the most-significant bit, and adding it to the value of the instruction counter.

\[
\begin{align*}
\text{If } \text{RT}^{0:3} = 0 & \text{ then} \\
& \quad \text{PC} \leftarrow (\text{PC} + \text{RepLeftBit}(I16 \parallel 0b00)) \& \text{LSLR} \& 0xFFFFFFFFC \\
\text{else} & \\
& \quad \text{PC} \leftarrow (\text{PC} + 4) \& \text{LSLR}
\end{align*}
\]
Branch If Not Zero Halfword

`brhnz rt, symbol`

Examine the preferred slot. If the rightmost halfword is not zero, proceed with the branch target. Otherwise, proceed with the next instruction.

The address of the branch target is computed by appending two zero bits to the value of the I16 field, extending it on the left with copies of the most-significant bit, and adding it to the value of the instruction counter.

```
If RT[2:3] ≠ 0 then
    PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
    PC ← (PC + 4) & LSLR
```
Branch If Zero Halfword

```
brhz rt,symbol
```

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examine the preferred slot. If the rightmost halfword is zero, proceed with the branch target. Otherwise, proceed with the next instruction.

The address of the branch target is computed by appending two zero bits to the value of the I16 field, extending it on the left with copies of the most-significant bit, and adding it to the value of the instruction counter.

If \( RT^{2:3} = 0 \) then
\[
\text{PC} \leftarrow (\text{PC} + \text{RepLeftBit}(I16 || 0b00)) \& \text{LSLR} \& 0xFFFFFFFC
\]
else
\[
\text{PC} \leftarrow (\text{PC} + 4) \& \text{LSLR}
\]
Branch Indirect If Zero

If the preferred slot of register RT is not zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

\[ t \leftarrow RA^{0,3} \& \text{LSLR} \& 0xFFFFFFFF \]
\[ u \leftarrow \text{LSLR} \& (PC + 4) \]

If \( RT^{0,3} = 0 \) then
\[ PC \leftarrow t \& \text{LSLR} \& 0xFFFFFFFF \]
- if \( E = 0 \) and \( D = 0 \) then interrupt enable status is not modified
- else if \( E = 1 \) and \( D = 0 \) then enable interrupts at target
- else if \( E = 0 \) and \( D = 1 \) then disable interrupts at target
- else if \( E = 1 \) and \( D = 1 \) then reserved

else
\[ PC \leftarrow u \]
Branch Indirect If Not Zero

Required v 1.0

If the preferred slot of register RT is zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

\[
t ← RA_{0:3} \& LSLR \& 0xFFFFFFFFC
\]
\[
u ← LSLR \& (PC + 4)
\]

If \(RT^{0:3} \neq 0\) then
\[
PC ← t \& LSLR \& 0xFFFFFFFFC
\]
- If \((E = 0 \text{ and } D = 0)\) then interrupt enable status is not modified
- else if \((E = 1 \text{ and } D = 0)\) then enable interrupts at target
- else if \((E = 0 \text{ and } D = 1)\) then disable interrupts at target
- else if \((E = 1 \text{ and } D = 1)\) then reserved
else
\[
PC ← u
\]
Branch Indirect If Zero Halfword

```
bihz     rt, ra
```

```
0 0 1 0 0 1 0 1 0 / D E / / / /       RA       RT

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
```

If the rightmost halfword of the preferred slot of register RT is not zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).

```
t ← RA[0:3] & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)

If RT[2:3] = 0 then do
	PC ← t & LSLR & 0xFFFFFFFC
	If (E = 0 and D = 0) then interrupt enable status is not modified
	else if (E = 1 and D = 0) then enable interrupts at target
	else if (E = 0 and D = 1) then disable interrupts at target
	else if (E = 1 and D = 1) then reserved

else
	PC ← u
```
Branch Indirect If Not Zero Halfword Required v 1.0

\[
t \leftarrow RA_{0:3} & \text{LSLR} & 0xFFFFFFFC
\]

\[
u \leftarrow \text{LSLR} & (PC + 4)
\]

If \( RT^{2:3} \neq 0 \) then

\[
\text{PC} \leftarrow t & \text{LSLR} & 0xFFFFFFFC
\]

- If \( E = 0 \) and \( D = 0 \) then interrupt enable status is not modified
- else if \( E = 1 \) and \( D = 0 \) then enable interrupts at target
- else if \( E = 0 \) and \( D = 1 \) then disable interrupts at target
- else if \( E = 1 \) and \( D = 1 \) then reserved

else

\[
\text{PC} \leftarrow u
\]

If the rightmost halfword of the preferred slot of register RT is zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled or disabled with the E or D feature bits (see Section 12 SPU Interrupt Facility on page 251).
8. Hint-for-Branch Instructions

This section lists and describes the SPU hint-for-branch instructions.

These instructions have no semantics. They provide a hint to the implementation about a future branch instruction, with the intention that the information be used to improve performance by either prefetching the branch target or by other means.

Each of the hint-for-branch instructions specifies the address of a branch instruction and the address of the expected branch target address. If the expectation is that the branch is not taken, the target address is the address of the instruction following the branch.

The instructions in this section use the variables `brinst` and `brtarg`, which are defined as follows:

- `brinst` = RO
- `brtarg` = I16
Hint for Branch (r-form)

Required v 1.0

The address of the branch target is given by the contents of the preferred slot of register RA. The RO field gives the signed word offset from the \texttt{hbr} instruction to the branch instruction.

If the P feature bit is set, \texttt{hbr} does not hint a branch. Instead, it hints that this is the proper implementation-specific moment to perform inline prefetching. Inline prefetching is the instruction fetch function necessary to run linearly sequential program text. To obtain optimal performance, some implementations of the SPU may require help scheduling these inline prefetches of local storage when the program is also doing loads and stores. See the implementation-specific SPU documentation for information about when this might be beneficial. When the P feature bit is set, the instruction ignores the value of RA. The relative offset (RO) field, formed by concatenating ROH (high) and ROL (low), must be set to zero.

\begin{verbatim}
branch target address ← RA[0:3] & LSLR & 0xFFFFFFFFC
branch instruction address ← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & LSLR
\end{verbatim}
Hint for Branch (a-form)

The address of the branch target is specified by an address in the I16 field. The value has 2 bits of zero appended on the right before it is used.

The RO field, formed by concatenating ROH (high) and ROL (low), gives the signed word offset from the \texttt{hbra} instruction to the branch instruction.

\begin{align*}
\text{branch target address} & \leftarrow \text{RepLeftBit}(I16 \| 0b00,32) \& \text{LSLR} \\
\text{branch instruction address} & \leftarrow (\text{RepLeftBit}(ROH \| ROL \| 0b00,32) + PC) \& \text{LSLR}
\end{align*}
The address of the branch target is specified by a word offset given in the I16 field. The signed I16 field is added to the address of the `hbrr` instruction to determine the absolute address of the branch target.

The RO field, formed by concatenating ROH (high) and ROL (low), gives the signed word offset from the `hbrr` instruction to the branch instruction.

\[
\text{branch target address} \leftarrow (\text{RepLeftBit}(I16 \| 0b00,32) + \text{PC}) \& \text{LSLR}
\]

\[
\text{branch instruction address} \leftarrow (\text{RepLeftBit}(ROH \| ROL \| 0b00,32) + \text{PC}) \& \text{LSLR}
\]
9. Floating-Point Instructions

This section describes the SPU floating-point instructions. This section also describes the differences between SPU floating-point calculations and IEEE standard floating-point calculations. The single-precision, floating-point instructions do not calculate results compliant with IEEE Standard 754. However, the data formats for single-precision and double-precision floating-point numbers used in the SPU are the same as the IEEE Standard 754.

Implementation Note: The architecture allows implementations to produce different results for floating-point instructions. See the implementation-specific documentation for information about the results produced by an implementation. To achieve the same results between implementations requires more than architectural compliance.

9.1 Single Precision (Extended-Range Mode)

For single-precision operations, the range of normalized numbers is extended. However, the full range defined in the standard is not implemented. The range of nonzero numbers that can be represented and operated on in the SPU is between the minimum and maximum listed in Table 9-1. Table 9-1 also demonstrates converting from a register value to a decimal value.

Table 9-1. Single-Precision (Extended-Range Mode) Minimum and Maximum Values

<table>
<thead>
<tr>
<th>Number Format</th>
<th>Minimum Positive Magnitude (Smin)</th>
<th>Maximum Positive Magnitude (Smax)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Value</td>
<td>0x00800000</td>
<td>0xFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>Bit Fields</td>
<td>Sign 8-Bit Biased Exponent Fraction (implied [1] and 23 bits)</td>
<td>Sign 8-Bit Biased Exponent Fraction (implied [1] and 23 bits)</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>00000001 [1.000...000]</td>
<td>0</td>
<td>11111111 [1.111...111]</td>
</tr>
<tr>
<td>Value in Powers of 2</td>
<td>+ 2^{1 - 127} 1</td>
<td>+ 2^{255 - 127} 2 - 2^-23</td>
<td>2</td>
</tr>
<tr>
<td>Combined Exponent and Fraction</td>
<td>2^{-126} * (+1)</td>
<td></td>
<td>2^{128} * (+2 - 2^{-23})</td>
</tr>
<tr>
<td>Value of Register in Decimal</td>
<td>1.2 * 10^{-38}</td>
<td></td>
<td>6.8 * 10^{38}</td>
</tr>
</tbody>
</table>

Notes:
1. The exponent field is biased by +127.
2. The value 2 - 2^{-23} is one least significant bit (LSb) less than 2.

Zero has two representations:
- For a positive zero, all bits are zero; that is, the sign, exponent, and fraction are zero.
- For a negative zero, the sign is one; that is, the exponent and fraction are zero.

As inputs, both kinds of zero are supported; however, a zero result is always a positive zero.

Single-precision operations in the SPU have the following characteristics:
- Not a Number (NaN) is not supported as an operand and is not produced as a result.
- Infinity (Inf) is not supported. An operation that produces a magnitude greater than the largest number representable in the target floating-point format instead produces a number with the appropriate sign, the largest biased exponent, and a magnitude of all (binary) ones. It is important to note that the representa-
tion of Inf, which conforms to the IEEE standard, is interpreted by the SPU as a number that is smaller than the largest number used on the SPU.

- Denorms are not supported and are treated as zero. Thus, an operation that would generate a denorm under IEEE rules instead generates a positive zero. If a denorm is used as an operand, it is treated as a zero.
- The only supported rounding mode is truncation (toward zero).

For single-precision extended-range arithmetic, four kinds of exception conditions are tested: overflow, underflow, divide-by-zero, and IEEE noncompliant result.

- **Overflow (OVF)**
  
  An overflow exception occurs when the magnitude of the result before rounding is bigger than the largest positive representable number, Smax. If the operation in slice $k$ produces an overflow, the OVF flag for slice $k$ in the Floating-Point Status and Control Register (FPSCR) is set, and the result is saturated to Smax with the appropriate sign.

- **Underflow (UNF)**
  
  An underflow exception occurs when the magnitude of the result before rounding is smaller than the smallest positive representable number, Smin. If the operation in slice $k$ produces an underflow, the UNF flag for slice $k$ in the FPSCR is set, and the result is saturated to a positive zero.

- **Divide-by-Zero (DBZ)**
  
  A divide-by-zero exception occurs when the input of an estimate instruction has a zero exponent. If the operation in slice $k$ produces a divide-by-zero exception, the DBZ flag for slice $k$ in the FPSCR is set.

- **IEEE noncompliant result (DIFF)**

  A different-from-IEEE exception indicates that the result produced with extended-range arithmetic could be different from the IEEE result. This occurs when one of the following conditions exists:
  
  - Any of the inputs or the result has a maximal exponent (IEEE arithmetic treats such an operand as NaN or Infinity; extended-range arithmetic treats them as normalized values.)
  - Any of the inputs has a zero exponent and a nonzero fraction (IEEE arithmetic treats such an operand as a denormal number; extended-range arithmetic treats them as a zero.)
  - An underflow occurs; that is, the result before rounding is different from zero and the result after rounding is zero.

  If this happens for the operation in slice $k$, the DIFF flag for slice $k$ in the FPSCR is set.

These exceptions can be set only by extended-range floating-point instructions. *Table 9-2* lists the instructions for which exceptions can be set.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Set OVF</th>
<th>Set UNF</th>
<th>Set DBZ</th>
<th>Set DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>fa, fs, fm, fma, fnms, fl</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>frest, frsqest</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>csflt, cuflt</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>cfllts, cflltu, fceq, fcneq, fcgt, fcmgt</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
9.2 Double Precision

SPU double-precision instructions process 128-bit values as two SIMD double-precision operations. SIMD slice 0 processes doubleword 0, and slice 1 processes doubleword 1. For double-precision operations, normal IEEE semantics and definitions apply. The range of the nonzero numbers supported by this format is between the minimum and the maximum listed in Table 9-3. Table 9-3 also demonstrates converting from a register value to a decimal value.

Table 9-3. Double-Precision (IEEE Mode) Minimum and Maximum Values

<table>
<thead>
<tr>
<th>Number Format</th>
<th>Minimum Positive Denormalized Magnitude (Dmin)</th>
<th>Maximum Positive Normalized Magnitude (Dmax)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Value</td>
<td>0x0000000000000001</td>
<td>0x7FEFFFFFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>Bit Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11-Bit Biased Exponent</td>
<td>[0.000...000]</td>
<td>[1.111...111]</td>
<td>2</td>
</tr>
<tr>
<td>Fraction (implied [0] and 52 bits for denormalized number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11-Bit Biased Exponent</td>
<td>[0.000...001]</td>
<td>[1.111...111]</td>
<td>2</td>
</tr>
<tr>
<td>Fraction (implied [1] and 52 bits for normalized number)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value in Powers of 2</td>
<td>+ 2^{(0 - 1023)}</td>
<td>+ 2^{-52}</td>
<td>3, 4</td>
</tr>
<tr>
<td>Combined Exponent and Fraction</td>
<td>2^{-1022} * (+2^{-52})</td>
<td>2^{1023} * (+[2 - 2^{-52}])</td>
<td></td>
</tr>
<tr>
<td>Value of Register in Decimal</td>
<td>4.9 \times 10^{-324}</td>
<td>1.8 \times 10^{308}</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. The exponent is biased by +1023.
2. An exponent field of all ones is reserved for not-a-number (NaN) and infinity.
3. The value 2^{-52} is one LSb less than 2.
4. An extra 1 is added to the exponent for denormalized numbers.

Double-precision operations in the SPU have the following characteristics:
- Only a subset of the operations required by the IEEE standard is supported in hardware.
- All four rounding modes are supported.
- The rounding modes for the two slices can be controlled independently. The RN0 field (bits 20 - 21) in the FPSCR specifies the current rounding mode for slice 0; the RN1 field (bits 22 -23) in the FPSCR specifies the current rounding mode for slice 1.
- The IEEE exceptions are detected and accumulated in the FPSCR. Trapping is not supported.
- The IEEE standard recognizes two kind of NaNs. These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored. If the high-order bit of the fraction field is 0b0, then the NaN is a Signaling NaN (SNaN); otherwise, it is a Quiet NaN (QNaN). When a QNaN is the result of a floating-point operation that has no NaN inputs, the result is always the default QNaN. That is, the high-order bit of the fraction field is 0b1, all the other bits of the fraction field are zero, and the sign bit is zero.
- The IEEE standard has very strict rules on the propagation of NaNs. When a QNaN is the result of a floating-point operation that has at least one NaN input, an SPU implementation can either produce the default QNaN or one of the input NaN values. If an implementation produces a QNaN result rather than propagating the proper input NaN, QNaN, or SNaN; the NaN flag in the FPSCR is set to signal a possibly noncompliant result.
Some implementations might support denorms only as results. Such an implementation treats denormal operands as zeros (this also applies to the setting of the IEEE flags); the sign of the operand is preserved. Whenever a denormal operand is forced to zero, the DENORM flag in the FPSCR is set to signal a possibly noncompliant result.

### 9.2.1 Conversions Between Single-Precision and Double-Precision Format

There are two types of conversions: one rounds a double-precision number to a single-precision number (`frds`); the other extends a single-precision number to a double-precision number (`fesd`). Both operations comply with the IEEE standard, except for the handling of denormal inputs. Some implementations may force denormal values to zero. When an implementation forces a denormal input to zero, it sets the DENORM flag rather than the Underflow flag in the FPSCR. Thus, for these two operations, NaNs, infinities, and denormal results are supported in double-precision format as well as in single-precision format. The range of nonzero IEEE single-precision numbers supported is between the minimum and the maximum listed in *Table 9-4*. *Table 9-4* also demonstrates converting from a register value to a decimal value.

**Table 9-4. Single-Precision (IEEE Mode) Minimum and Maximum Values**

<table>
<thead>
<tr>
<th>Number Format</th>
<th>Minimum Positive Denormalized Magnitude (Smin)</th>
<th>Maximum Positive Magnitude (Smax)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Value</td>
<td>0x00000001</td>
<td>0x7F7FFFFF</td>
<td></td>
</tr>
<tr>
<td>Bit Fields</td>
<td>Sign 8-Bit Biased Exponent Fraction (implied)</td>
<td>Sign 8-Bit Biased Exponent Fraction (implied)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0 00000000 [0.]000..001</td>
<td>0 11111110 [1.]111...111</td>
<td></td>
</tr>
<tr>
<td>Value in Powers of 2</td>
<td>+ $2^{(0+1-127)}$ $2^{-23}$</td>
<td>+ $2^{(254-127)}$ $2^{-23}$</td>
<td>2</td>
</tr>
<tr>
<td>Combined Exponent and Fraction</td>
<td>$2^{-126} * 2^{-23}$</td>
<td>$2^{127} * (2 - 2^{-23})$</td>
<td></td>
</tr>
<tr>
<td>Value of Register in Decimal</td>
<td>1.4 * $10^{-45}$</td>
<td>3.4 * $10^{38}$</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
1. The exponent field is biased by +127.
2. The value $2 - 2^{-23}$ is 1 LSb less than 2.

### 9.2.2 Exception Conditions

This architecture only supports nontrap exception handling; that is, exception conditions are detected and reported in the appropriate fields of the FPSCR. These flags are sticky; once set, they remain set until they are cleared by an FPSCR-write instruction. These exception flags are not set by the single-precision operations executed in the extended range. Because the double-precision operations are 2-way SIMD, there are two sets of these flags.

*Inexact Result (INX)*
An inexact result is detected when the delivered result value differs from what would have been computed if both the exponent range and precision were unbounded.

*Overflow (OVF)*
An overflow occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.
Underflow (UNF)
For nontrap exception handling, the IEEE 754 standard defines the underflow (UNF) as the following:

\[
\text{UNF} = \text{tiny} \text{ AND } \text{loss_of_accuracy}
\]

Where there are two definitions each for tiny and loss of accuracy, and the implementation is free to choose any of the four combinations. This architecture implements tiny-before-rounding and inexact result (INX), thus:

\[
\text{UNF} = \text{tiny\_before\_rounding} \text{ AND } \text{inexact\_result}
\]

**Note:** Tiny before rounding is detected when a nonzero result value, computed as though the exponent range were unbounded, would be less in magnitude than the smallest normalized number.

Invalid Operation (INV)
An invalid operation exception occurs whenever an operand is invalid for the specified operation. For operations implemented in hardware, the following operations give rise to an invalid operation exception condition:

- Any floating-point operation on a signaling NaN (SNaN)
- For add, subtract, and fused multiply add operations on magnitude subtraction of infinities; that is, infinity - infinity
- Multiplication of infinity by zero.

**Note:** Some implementations may treat denormal inputs as zeros and set both the DENORM flag and the Invalid Operation flag.

Not Propagated NaN (NaN)
The IEEE standard requires special handling of input NaNs, but SPU implementations can deliver the default QNaN as a result of double-precision operations. When at least one of the inputs is a NaN, the resulting QNaN can differ from the result delivered by a design that is fully compliant with the IEEE standard. This is flagged in the NaN field.

Denormal Input Forced to Zero (DENORM)
SPU implementations can force certain double-precision denormal operands to zeros before the processing of double-precision operations. If an implementation forces these operands to zeros, the zero will preserve the sign of the original denormal value. When a denormal input is forced to zero, the DENORM exception flag is set in the FPSCR to signal that the result could differ from an IEEE-compliant result.

**Programming Note:** Applications that require IEEE-compliant double-precision results can use the NaN and DENORM flags in the FPSCR to detect noncompliant results. This allows the code to be re-executed in a less efficient but compliant manner. Both flags are sticky, so that large blocks of code can be guarded, minimizing the overhead of the code checking. For example,

```c
clear fpscr
fast code block
if (NaN || DENORM)
{
    compliant code block
}
```
On SPUs within CBEA-compliant processors, the SPU can stop and signal the PPE to request that the PPE perform the calculation and then restart the SPU.

Table 9-5 lists the instructions for which exceptions can be set.

### Table 9-5. Instructions and Exception Settings

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Set OVF</th>
<th>Set UNF</th>
<th>Set INX</th>
<th>Set INV</th>
<th>Set NAN</th>
<th>Set DENORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfa, dfs, dfm, dfma, dfms, dfnms, dfnma</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>fesd</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>frds</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 9.3 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) records the status resulting from the floating-point operations and controls the rounding mode for double-precision operations. The FPSCR is read by the FPSCR read instruction (fscrrd) and written with the FPSCR write instruction (fscrwr). Bits [20:23] are control bits; the remaining bits are either status bits or unused. All the status bits in the FPSCR are sticky. That is, once set, the sticky bits remain set until they are cleared by an fscrwr instruction.

The format of the FPSCR is as follows.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:19</td>
<td>Unused</td>
</tr>
<tr>
<td>20:21</td>
<td>Rounding control for slice 0 of the 2-way SIMD double-precision operations (RN0)</td>
</tr>
<tr>
<td>00</td>
<td>Round to nearest even</td>
</tr>
<tr>
<td>01</td>
<td>Round towards zero (truncate)</td>
</tr>
<tr>
<td>10</td>
<td>Round towards +infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round towards -infinity</td>
</tr>
<tr>
<td>22:23</td>
<td>Rounding control for slice 1 of the 2-way SIMD double-precision operations (RN1)</td>
</tr>
<tr>
<td>00</td>
<td>Round to nearest even</td>
</tr>
<tr>
<td>01</td>
<td>Round towards zero (truncate)</td>
</tr>
<tr>
<td>10</td>
<td>Round towards +infinity</td>
</tr>
<tr>
<td>11</td>
<td>Round towards -infinity</td>
</tr>
<tr>
<td>24:28</td>
<td>Unused</td>
</tr>
<tr>
<td>29:31</td>
<td>Single-precision exception flags for slice 0</td>
</tr>
<tr>
<td>29</td>
<td>Overflow (OVF)</td>
</tr>
<tr>
<td>30</td>
<td>Underflow (UNF)</td>
</tr>
<tr>
<td>31</td>
<td>Result produced with extended-range arithmetic could be different from the IEEE compliant result (DIFF)</td>
</tr>
<tr>
<td>32:49</td>
<td>Unused</td>
</tr>
<tr>
<td>50:55</td>
<td>IEEE exception flags for slice 0 of the 2-way SIMD double-precision operations</td>
</tr>
<tr>
<td>50</td>
<td>Overflow (OVF)</td>
</tr>
<tr>
<td>51</td>
<td>Underflow (UNF)</td>
</tr>
<tr>
<td>52</td>
<td>Inexact result (INX)</td>
</tr>
<tr>
<td>53</td>
<td>Invalid operation (INV)</td>
</tr>
<tr>
<td>54</td>
<td>Possibly noncompliant result because of QNaN propagation (NaN)</td>
</tr>
<tr>
<td>55</td>
<td>Possibly noncompliant result because of denormal operand (DENORM)</td>
</tr>
<tr>
<td>56:60</td>
<td>Unused</td>
</tr>
<tr>
<td>Bits</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>61:63</td>
<td>Single-precision exception flags for slice 1 (OVF, UNF, DIFF)</td>
</tr>
<tr>
<td>64:81</td>
<td>Unused</td>
</tr>
<tr>
<td>82:87</td>
<td>IEEE exception flags for slice 1 of the 2-way SIMD double-precision operations (OVF, UNF, INX, INV, NAN, DENORM)</td>
</tr>
<tr>
<td>88:92</td>
<td>Unused</td>
</tr>
<tr>
<td>93:95</td>
<td>Single-precision exception flags for slice 2 (OVF, UNF, DIFF)</td>
</tr>
<tr>
<td>96:115</td>
<td>Unused</td>
</tr>
<tr>
<td>116:119</td>
<td>Single-precision divide-by-zero flags for each of the four slices</td>
</tr>
<tr>
<td>116</td>
<td>DBZ for slice 0</td>
</tr>
<tr>
<td>117</td>
<td>DBZ for slice 1</td>
</tr>
<tr>
<td>118</td>
<td>DBZ for slice 2</td>
</tr>
<tr>
<td>119</td>
<td>DBZ for slice 3</td>
</tr>
<tr>
<td>120:124</td>
<td>Unused</td>
</tr>
<tr>
<td>125:127</td>
<td>Single-precision exception flags for slice 3 (OVF, UNF, DIFF)</td>
</tr>
</tbody>
</table>
Floating Add

For each of the four word slots:

- The operand from register RA is added to the operand from register RB.
- The result is placed in register RT.
- If the magnitude of the result is greater than Smax, then Smax (with the correct sign) is produced as the result. If the magnitude of the result is less than Smin, then zero is produced.
Double Floating Add

```
<table>
<thead>
<tr>
<th>dfa</th>
<th>rt,ra,rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011001100</td>
<td>RB RA RT</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
<td>11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>
```

For each of two doubleword slots:

- The operand from register RA is added to the operand from register RB.
- The result is placed in register RT.
Floating Subtract

For each of the four word slots:

- The operand from register RB is subtracted from the operand from register RA.
- The result is placed in register RT.
- If the magnitude of the result is greater than Smax, then Smax (with the correct sign) is produced as the result. If the magnitude of the result is less than Smin, then zero is produced.
Double Floating Subtract

For each of two doubleword slots:

- The operand from register RB is subtracted from the operand from register RA.
- The result is placed in register RT.
Floating Multiply

For each of the four word slots:

- The operand from register RA is multiplied by the operand from register RB.
- The result is placed in register RT.
- If the magnitude of the result is greater than Smax, then Smax (with the correct sign) is produced. If the magnitude of the result is less than Smin, then zero is produced.
Double Floating Multiply

```
dfm          rt,ra,rb
0 1 0 1 1 0 0 1 1 1 0
```

For each of two doubleword slots:

- The operand from register RA is multiplied by the operand from register RB.
- The result is placed in register RT.
Floating Multiply and Add  

```
fma rt, ra, rb, rc
```

1 1 1 0

For each of the four word slots:

- The operand from register RA is multiplied by the operand from register RB and added to the operand from register RC. The multiplication is exact and not subject to limits on its range.
- The result is placed in register RT.
- If the magnitude of the result of the addition is greater than Smax, then Smax (with the correct sign) is produced. If the magnitude of the result is less than Smin, then zero is produced.
Double Floating Multiply and Add

\texttt{dfma \ rt,ra,rb}

\begin{verbatim}
0 1 1 0 1 1 1 0 0
\end{verbatim}

For each of two doubleword slots:

- The operand from register RA is multiplied by the operand from register RB and added to the operand from register RT. The multiplication is exact and not subject to limits on its range.
- The result is placed in register RT.
Floating Negative Multiply and Subtract  

Required  

fnms  
rt,ra,rb,rc  

For each of the four word slots:

- The operand from register RA is multiplied by the operand from register RB, and the product is subtracted from the operand from register RC. The result of the multiplication is exact and not subject to limits on its range.
- The result is placed in register RT.
- If the magnitude of the result of the subtraction is greater than Smax, then Smax (with the correct sign) is produced. If the magnitude of the result of the subtraction is less than Smin, then zero is produced.
Double Floating Negative Multiply and Subtract

dfnms rt,ra,rb

0 1 1 0 1 0 1 1 1 0 RB RA RT

For each of two doubleword slots:

- The operand from register RA is multiplied by the operand from register RB. The operand from register RT is subtracted from the product. The result, which is placed in register RT, is usually obtained by negating the rounded result of this multiply subtract operation. There is one exception: If the result is a QNaN, the sign bit of the result is zero.

- This instruction produces the same result as would be obtained by using the Double Floating Multiply and Subtract instruction and then negates any result that is not a NaN.

- The multiplication is exact and not subject to limits on its range.
Floating Multiply and Subtract

required v 1.0

fms rt,ra,rb,rc

For each of the four word slots:

- The operand from register RA is multiplied by the operand from register RB. The result of the multiplication is exact and not subject to limits on its range. The operand from register RC is subtracted from the product.
- The result is placed in register RT.
- If the magnitude of the result of the subtraction is greater than Smax, then Smax (with the correct sign) is produced. If the magnitude of the result of the subtraction is less than Smin, then zero is produced.
Double Floating Multiply and Subtract

For each of two doubleword slots:

- The operand from register RA is multiplied by the operand from register RB. The multiplication is exact and not subject to limits on its range. The operand from register RT is subtracted from the product.
- The result is placed in register RT.
Double Floating Negative Multiply and Add

\[ \text{dfnma} \quad \text{rt}, \text{ra}, \text{rb} \]

For each of two doubleword slots:

- The operand from register RA is multiplied by the operand from register RB and added to the operand from register RT. The multiplication is exact and not subject to limits on its range. The result, which is placed in register RT, is usually obtained by negating the rounded result of this multiply add operation. There is one exception: If the result is a QNaN, the sign bit of the result is 0.

- This instruction produces the same result as would be obtained by using the Double Floating Multiply and Add instruction and then negating any result that is not a NaN.
Floating Reciprocal Estimate

**frest**  
**rt,ra**

For each of four word slots:

- The operand in register RA is used to compute a base and a step for estimating the reciprocal of the operand. The result, in the form shown below, is placed in register RT. S is the sign bit of the base result.

<table>
<thead>
<tr>
<th>S</th>
<th>Biased Exponent</th>
<th>BaseFraction</th>
<th>StepFraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16</td>
<td>17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</td>
<td></td>
</tr>
</tbody>
</table>

- The base result is expressed as a floating-point number with 13 bits in the fraction, rather than the usual 23 bits. The remaining 10 bits of the fraction are used to encode the magnitude of the step as a 10-bit denormal fraction; the exponent is that of the base.

- The step fraction differs from the base fraction (and any normalized IEEE fraction) in that there is a ‘0’ in front of the binary point and three additional bits of ‘0’ between the binary point and the fraction. The represented numbers are as follows:

<table>
<thead>
<tr>
<th>Base</th>
<th>S 1.(\text{BaseFraction} \times 2^{\text{BiasedExponent}} - 127)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>0.000 (\text{StepFraction} \times 2^{\text{BiasedExponent}} - 127)</td>
</tr>
</tbody>
</table>

- Let \(x\) be the initial value in register RA. The result placed in RT, which is interpreted as a regular IEEE number, provides an estimate of the reciprocal of a nonzero \(x\).

- If the operand in register RA has a zero exponent, a divide-by-zero exception is flagged.

**Programming Note:** The result returned by this instruction is intended as an operand for the Floating Interpolate instruction.

The quality of the estimate produced by the Floating Reciprocal Estimate instruction is sufficient to produce a result within 1 ulp of the IEEE single-precision reciprocal after interpolation and a single step of Newton-Raphson. Consider this code sequence:

```
FREST y0,x  // table-lookup
FI y1,x,y0  // interpolation
FNMS t1,x,y1,ONE  // t1 = -(x * y1 - 1.0)
FMA y2,t1,y1,y1  // y2 = t1 * y1 + y1
```

Three ranges of input must be described separately:

**Zeros**  
1/0 is defined to give the maximum SPU single-precision extended-range floating point (sfp) number:

\[ y2 = x'7FFFF FFFF' (1.999 \times 2^{128}) \]
Big

If $|x| \geq 2^{126}$, then $1/x$ underflows to zero, $y_2 = 0$.

**Note:** This underflows for one value of $x$ that IEEE single-precision reciprocal would not. If this is a concern, the following code sequence produces the IEEE answer:

```plaintext
maxnounderflow = 0x7e800000
min = 0x00800000
msb = 0x80000000
FCMEQ selmask,x,maxnounderflow
AND s1,x,msb
OR smin,s1,min
SELB y3,selmask,y2,smin
```

Normal

$1/x = Y$ where $x \cdot Y < 1.0$ and $x \cdot \text{INC}(Y) \geq 1.0$.

INC($y$) gives the sfp number with the same sign as $y$ and next larger magnitude. The absolute error bound is:

$$| Y - y_2 | \leq 1 \text{ ulp} \quad (\text{either } y_2 = Y, \text{ or } \text{INC}(y_2) = Y)$$
Floating Reciprocal Absolute Square Root Estimate

frsqest rt,ra

For each of four word slots:

- The operand in register RA is used to compute a base and step for estimating the reciprocal of the square root of the absolute value of the operand. The result is placed in register RT. The sign bit (S) will be zero.

- Let \( x \) be the initial value of register RA. The result placed in register RT, interpreted as a regular IEEE number, provides an estimate of the reciprocal square root of \( \text{abs}(x) \).

- If the operand in register RA has a zero exponent, a divide-by-zero exception is flagged.

**Programming Note:** The result returned by this instruction is intended as an operand for the Floating Interpolate instruction.

The quality of the estimate produced by the Floating Reciprocal Absolute Square Root Estimate instruction is sufficient to produce an IEEE single-precision reciprocal after interpolation and a single step of Newton-Raphson. Consider the following code sequence:

```assembly
mask=0x7fffffff
half=0.5
one=1.0
FRSQEST y0,x // table-lookup
AND ax,x,mask // ax = ABS(x)
FI y1,ax,y0 // interpolation
FM t1,ax,y1 // t1 = ax * y1
FM t2,y1,HALF // t2 = y1 * 0.5
FNMS t1,t1,y1,ONE // t1 = -(t1 * y1 - 1.0)
FMA y2,t1,t2,y1 // y2 = t1 * t2 + y1
```

Three ranges of input must be described separately:

**Zeros, where:** \( x \) fraction \( \leq 0x000ff53c \) then \( y2 = 0x7fffffff \) \((1.999 \times 2^{128})\)

**Zeros where:** \( x \) fraction > \( 0x000ff53c \), \( y2 \approx 0x7fc00000 \)

The following sequence could be used to correct the answer:

```assembly
zero = 0.0
mask = 0x7fffffff
FCMEQ z,x,zero
AND zmask,z,mask
OR y3,zmask,y2
```
Normal

\[ \frac{1}{\sqrt{x}} = Y \text{ where } x \cdot Y^2 < 1.0 \text{ and } x \cdot \text{INC}(Y)^2 \geq 1.0 \]

INC(y) gives the sfp number with the same sign as y and next larger magnitude. The absolute error bound is:

\[ | Y - y^2 | \leq 1 \text{ ulp} \quad (0 \text{ and } \pm 1 \text{ are all possible}) \]
Floating Interpolate

fi rt, ra, rb

For each of four word slots:

- The operand in register RB is disassembled to produce a floating-point base and step according to the format described in *Floating Reciprocal Estimate* on page 215; that is, a sign, biased exponent, base fraction, and step fraction.

- Bits 13 to 31 of register RA are taken to represent a fraction, Y, whose binary point is to the left of bit 13; that is, $Y \leftarrow 0.RA_{13:31}$.

The result is computed by the following equation:

$$RT \leftarrow (-1)^S \times (1.\text{BaseFraction} - 0.000\text{StepFraction} \times Y) \times 2^{(\text{BiasedExponent} - 127)}$$

**Programming Note:** If the operand in register RB is the result of an *frest* or *frsqest* instruction with the operand from register RA, then the result of the fi instruction placed in register RT provides a more accurate estimation.
Convert Signed Integer to Floating

csflt rt,ra,scale

For each of four word slots:

- The signed 32-bit integer value in register RA is converted to an extended-range, single-precision, floating-point value.

- The result is divided by $2^{scale}$ and placed in register RT. The factor scale is an 8-bit unsigned integer provided by 155 minus the unsigned value from the I8 field. If the value scale is not in the range of 0 to 127, the result of the operation is undefined.

- The scale factor describes the number of bit positions between the binary point of the magnitude and the right end of register RA. A scale factor of zero means that the register RA value is an unscaled integer.
Convert Floating to Signed Integer

\[ \text{cflts } rt, ra, scale \]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>18</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of four word slots:

- The extended-range, single-precision, floating-point value in register RA is multiplied by \(2^{\text{scale}}\). The factor scale is an 8-bit unsigned integer provided by 173 minus the unsigned value from the I8 field. If the value scale is not in the range of 0 to 127, the result of the operation is undefined.

- The product is converted to a signed 32-bit integer. If the intermediate result is greater than \((2^{31} - 1)\), it saturates to \((2^{31} - 1)\); if it is less than \(-2^{31}\), it saturates to \(-2^{31}\). The resulting signed integer is placed in register RT.

- The scale factor is the location of the binary point of the result, expressed as the number of bit positions from the right end of the register RT. A scale factor of zero means that the value in register RT is an unscaled integer.
For each of four word slots:

- The unsigned 32-bit integer value in register RA is converted to an extended-range, single-precision, floating-point value.

- The result is divided by $2^{\text{scale}}$ and placed in register RT. The factor scale is an 8-bit unsigned integer provided by 155 minus the unsigned value from the I8 field. If the value scale is not in the range of 0 to 127, the result of the operation is undefined.

- The scale factor describes the number of bit positions between the binary point of the magnitude and the right end of register RA. A scale factor of zero means that the register RA value is an unscaled integer.
Convert Floating to Unsigned Integer

**cfltu**

rt,ra,\scale

For each of four word slots:

- The extended-range, single-precision, floating-point value in register RA is multiplied by \(2^{\scale}\). The factor \(\scale\) is an 8-bit unsigned integer provided by 173 minus the unsigned value from the I8 field. If the value \(\scale\) is not in the range of 0 to 127, the result of the operation is undefined.

- The product is converted to an unsigned 32-bit integer. If the intermediate result is greater than \((2^{32} - 1)\) it saturates to \((2^{32} - 1)\). If the product is negative, it saturates to zero. The resulting unsigned integer is placed in register RT.

- The scale factor is the location of the binary point of the result, expressed as the number of bit positions from the right end of the register RT. A scale factor of zero means that the value in RT is an unscaled integer.
Floating Round Double to Single

For each of two doubleword slots:

- The double-precision value in register RA is rounded to a single-precision, floating-point value and placed in the left word slot. The conversions are done as described in Section 9.2.1 Conversions Between Single-Precision and Double-Precision Format on page 198. Zeros are placed in the right word slot.

- The rounding is performed in accordance with the rounding mode specified in the Floating-Point Status Register. Double-precision exceptions are detected and accumulated in the Floating-Point Unit (FPU) Status Register.
Floating Extend Single to Double

**fesd**  
**rt,ra**

For each of two doubleword slots:

- The single-precision value in the left slot of register RA is converted to a double-precision, floating-point value and placed in register RT. The conversions are done as described in *Section 9.2.1 Conversions Between Single-Precision and Double-Precision Format* on page 198. The contents of the right word slot are ignored.

- Double-precision exceptions are detected and accumulated in the FPU Status Register.
Double Floating Compare Equal

Optional v 1.2

dfceq rt,ra,rb

For each of the two doubleword slots:

- The double-precision floating-point value from register RA is compared with the double-precision floating-point value from register RB. If the values are equal, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT.
- Two zeros always compare equal independent of their signs.
- A NaN compares false against all other operands. Even two NaNs with identical bit patterns generate false.
- When accessing a NaN, the corresponding INV exception bit in the FPSCR is set.
Double Floating Compare Magnitude Equal

Optional v 1.2

dfcmeq rt,ra,rb

For each of the two doubleword slots:

- The absolute value of the double-precision floating-point number in register RA is compared with the absolute value of the double-precision floating-point number in register RB. If the absolute values are equal, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT.

- Two zeros always compare equal independent of their signs.

- A NaN compares false against all other operands. Even two NaNs with identical bit patterns generate false.

- When accessing a NaN, the corresponding INV exception bit in the FPSCR is set.
Double Floating Compare Greater Than

```
dfcgt rt,ra,rb
```

For each of the two doubleword slots:

- The double-precision floating-point value in register RA is compared with the double-precision floating-point value in register RB. If the value in RA is greater than the value in RB, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT.
- Two zeros never compare greater than, independent of their sign bits.
- A NaN compares false against all other operands. Even two NaNs with identical bit patterns generate false.
- When accessing a NaN, the corresponding INV exception bit in the FPSCR is set.
Double Floating Compare Magnitude Greater Than

```
  dfcmgt
  rt,ra,rb
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>RB</th>
<th>RA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

For each of the two doubleword slots:

- The absolute value of the double-precision floating-point number in register RA is compared with the absolute value of the double-precision floating-point number in register RB. If the absolute value of the value from register RA is greater than the absolute value of the value from register RB, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT.
- Two zeros never compare greater than, independent of their signs.
- A NaN compares false against all other operands. Even two NaNs with identical bit patterns generate false.
- When accessing a NaN, the corresponding INV exception bit in the FPSCR is set.
Double Floating Test Special Value

Optional v 1.2

dftsv rt,ra,value

For each of two doubleword slots:

- The double-precision floating-point value in register RA is tested for special values. The bits of I7 enable the following seven checks:

<table>
<thead>
<tr>
<th>I7</th>
<th>RA Value Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>NaN</td>
</tr>
<tr>
<td>0100000</td>
<td>+Infinity</td>
</tr>
<tr>
<td>0010000</td>
<td>-Infinity</td>
</tr>
<tr>
<td>0001000</td>
<td>+0</td>
</tr>
<tr>
<td>0000100</td>
<td>-0</td>
</tr>
<tr>
<td>0000010</td>
<td>Positive Denorm</td>
</tr>
<tr>
<td>0000001</td>
<td>Negative Denorm</td>
</tr>
</tbody>
</table>

- If one or more of the enabled checks is true, a result of all ones is produced in register RT. When none of the enabled checks is met, a result of all zeros is produced in register RT.
Floating Compare Equal

fceq rt,ra,rb

For each of four word slots:

- The floating-point value from register RA is compared with the floating-point value from register RB. If the values are equal, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT. Two zeros always compare equal independent of their fractions and signs.
- This instruction is always executed in extended-range mode and ignores the setting of the mode bit.
Floating Compare Magnitude Equal

\texttt{fcmeq \quad rt,ra,rb}

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

\[ \text{RB} \quad \text{RA} \quad \text{RT} \]

For each of four word slots:

- The absolute value of the floating-point number in register RA is compared with the absolute value of the floating-point number in register RB. If the absolute values are equal, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT. Two zeros always compare equal independent of their fractions and signs.

- This instruction is always executed in extended-range mode and ignores the setting of the mode bit.
Floating Compare Greater Than

```
fcgt  rt, ra, rb
```

For each of four word slots:

- The floating-point value in register RA is compared with the floating-point value in register RB. If the value in RA is greater than the value in RB, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT. Two zeros never compare greater than independent of their sign bits and fractions.

- This instruction is always executed in extended-range mode, and ignores the setting of the mode bit.
Floating Compare Magnitude Greater Than

fcmgt rt, ra, rb

For each of four word slots:

- The absolute value of the floating-point number in register RA is compared with the absolute value of the floating-point number in register RB. If the absolute value of the value from register RA is greater than the absolute value of the value from register RB, a result of all ones (true) is produced in register RT. Otherwise, a result of zero (false) is produced in register RT. Two zeros never compare greater than, independent of their fractions and signs.
- This instruction is always executed in extended-range mode, and ignores the setting of the mode bit.
Floating-Point Status and Control Register Write

fscrwr | ra
---|---
0 1 1 1 0 1 1 0 | RA
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ | RT
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

The 128-bit value of register RA is written into the FPSCR. The value of the unused bits in the FPSCR is undefined. RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.
Floating-Point Status and Control Register Read

This instruction reads the value of the FPSCR. In the result, the unused bits of the FPSCR are forced to zero. The result is placed in the register RT.

```
fscrrd      rt
  0 1 1 1 0 0 1 1 0 0  ///  ///  ///  RT
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ lenders
  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
```
10. Control Instructions

This section lists and describes the SPU control instructions.
Stop and Signal

**stop**

![Stop and Signal Type]

Execution of the program in the SPU stops, and the external environment is signaled. No further instructions are executed.

PC ← PC + 4 & LSLR
precise stop
Stop and Signal with Dependencies

stopd

Execution of the program in the SPU stops.

Programming Note: This instruction differs from stop only in that, in typical implementations, instructions with dependencies can be replaced with stopd to create a breakpoint without affecting the instruction timings.
No Operation (Load)

Inop

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

This instruction has no effect on the execution of the program. It exists to provide implementation-defined control of instruction issuance.
No Operation (Execute)

This instruction has no effect on the execution of the program. It exists to provide implementation-defined control of instruction issuance. RT is a false target. Implementations can schedule instructions as though this instruction produces a value into RT. Programs can avoid unnecessary delay by programming RT so as not to appear to source data for nearby subsequent instructions. False targets are not written.
This instruction has no effect on the execution of the program other than to cause the processor to wait until all pending store instructions have completed before fetching the next sequential instruction. This instruction must be used following a store instruction that modifies the instruction stream.

The C feature bit causes channel synchronization to occur before instruction synchronization occurs. Channel synchronization allows an SPU state modified through channel instructions to affect execution. Synchronization is discussed in more detail in Section 13 Synchronization and Ordering on page 253.
Synchronize Data

**dsync**

This instruction forces all earlier load, store, and channel instructions to complete before proceeding. No subsequent load, store, or channel instructions can start until the previous instructions complete. The `dsync` instruction allows SPU software to ensure that the local storage data would be consistent if it were observed by another entity. This instruction does not affect any prefetching of instructions that the processor might have done. Synchronization is discussed in more detail in *Section 13 Synchronization and Ordering* on page 253.
Move from Special-Purpose Register

Synergistic Processor Unit

mfspr rt, sa

<table>
<thead>
<tr>
<th>opcode</th>
<th>SA</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1 1 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special-Purpose Register SA is copied into register RT. If SPR SA is not defined, zeros are supplied.

Note: The SPU ISA defines the mtspr and mfspr instructions as 128-bit operations. An implementation might define 32-bit wide registers. In that case, the 32-bit value occupies the preferred slot; the other slots return zeros.

```
if defined(SPR(SA)) then RT ← SPR(SA)
else RT ← 0
```
The contents of register RT is written to Special-Purpose Register SA. If SPR SA is not defined, no operation is performed.

**Note:** The SPU ISA defines the `mtspr` and `mfspr` instructions as 128-bit operations. An implementation might define 32-bit wide registers. In that case, the 32-bit value of the preferred slot is used; values in the other slots are ignored.

```plaintext
if defined(SPR(SA)) then
    SPR(SA) ← RT
else
    do nothing
```
11. Channel Instructions

The SPU provides an input/output interface based on message passing called the *channel interface*. This section describes the instructions used to communicate between the SPU and external devices through the channel interface.

Channels are 128-bit wide communication paths between the SPU and external devices. Each channel operates in one direction only, and is called either a read channel or a write channel, according to the operation that the SPU can perform on the channel. Instructions are provided that allow the SPU program to read from or write to a channel; the operations performed must match the type of channel addressed.

An implementation can implement any number of channels up to 128. Each channel has a channel number in the range 0-127. Channel numbers have no particular significance, and there is no relationship between the direction of a channel and its number.

The channels and the external devices have capacity. Channel capacity is the minimum number of reads or writes that can be performed without delay. Attempts to access a channel without capacity cause instruction processing to cease until capacity becomes available and the access can complete. The SPU maintains counters to measure channel capacity and provides an instruction to read channel capacity.

As long as capacity is available, the channels and external devices can service a burst of SPU accesses without requiring the SPU to delay execution. An attempt to write to a channel beyond its capacity causes the SPU to hang until the external device empties the channel. An attempt to read from a channel when it is empty also causes the SPU to hang until the device inserts data into the channel.
Read Channel

**rdch** rt,ca

```plaintext
0 0 0 0 0 0 1 1 0 1
\[\]
CA
RT
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
```

The SPU waits for data to become available in channel CA (capacity is available). When data is available to the channel, it is moved from the channel and placed into register RT.

If the channel designated by the CA field is not a valid, readable channel, the SPU will stop on or after the `rdch` instruction.

**Note:** The SPU ISA defines the `rdch` and `wrch` instructions as 128-bit operations. An implementation might define 32-bit wide channels. In that case, the 32-bit value occupies the preferred slot; the other slots return zeros.

```plaintext
if readable(\text{Channel(CA)}) \text{ then}
   \text{RT} \leftarrow \text{Channel(CA)}
else
   \text{Stop after executing zero or more instructions after the \text{rdch}.}
```
The channel capacity of channel CA is placed into the preferred slot of register RT. The channel capacity of unimplemented channels is zero.

\[
\text{RT}^{5:3} \leftarrow \text{Channel Capacity(CA)}
\]
\[
\text{RT}^{4:15} \leftarrow 0
\]
The SPU waits for capacity to become available in channel CA before executing the `wrch` instruction. When capacity is available in the channel, the contents of register RT are placed into channel CA. Channel writes targeting channels that are not valid writable channels cause the SPU to stop on or after the `wrch` instruction.

**Note:** The SPU ISA defines the `rdch` and `wrch` instructions as 128-bit operations. An implementation might define 32-bit wide channels. In that case, the 32-bit value of the preferred slot is used; values of the other slots are ignored.

```plaintext
if writable (Channel(CA)) then
    Channel(CA) ← RT
else
    Stop after executing zero or more instructions after the `wrch`.
```
12. SPU Interrupt Facility

This section describes the SPU interrupt facility.

External conditions are monitored and managed through external facilities that are controlled through the channel interface. External conditions can affect SPU instruction sequencing through the following facilities:

- **The bisled instruction**

  The bisled instruction tests for the existence of an external condition and branches to a target if it is present. The bisled instruction allows the SPU software to poll for external conditions and to call a handler subroutine, if one is present. When polling is not required, the SPU can be enabled to interrupt normal instruction processing and to vector to a handler subroutine when an external condition appears.

- **The interrupt facility**

The following indirect branch instructions allow software to enable and disable the interrupt facility during critical subroutines:

- bi
- bisl
- bisled
- biz
- binz
- bihz
- bihnz

All of these branch instructions provide the [D] and [E] feature bits. When one of these branches is taken, the interrupt-enable status changes before the target instruction is executed. Table 12-1 describes the feature bit settings and their results.

### Table 12-1. Feature Bits [D] and [E] Settings and Results

<table>
<thead>
<tr>
<th>Feature Bit Setting</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[D] [E]</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>Status does not change.</td>
</tr>
<tr>
<td>0 1</td>
<td>Interrupt processing is enabled.</td>
</tr>
<tr>
<td>1 0</td>
<td>Interrupt processing is disabled.</td>
</tr>
<tr>
<td>1 1</td>
<td>Causes undefined behavior.</td>
</tr>
</tbody>
</table>

12.1 SPU Interrupt Handler

The SPU supports a single interrupt handler. The entry point for this handler is address 0 in local storage. When a condition is present and interrupts are enabled, the SPU branches to address 0 and disables the interrupt facility. The address of the next instruction to be executed is saved in the SRR0 register. The iret instruction can be used to return from the handler. iret branches indirectly to the address held in the SRR0 register. iret, like the other indirect branches, has an [E] feature bit that can be used to re-enable interrupts.
12.2 SPU Interrupt Facility Channels

The interrupt facility uses several channels for configuration, state observation, and state restoration. The current value of SRR0 can be read from the SPU_RdSRR0 channel, and the SPU_WrSRR0 channel provides write access to SRR0. When SRR0 is written by wrch 14, synchronization is required to ensure that this new value is available to the iret instruction. This synchronization is provided by executing the sync instruction with the [C], or Channel Sync, feature bit set. Without this synchronization, iret instructions executed after wrch 14 instructions branch to unpredictable addresses. The SPU_RdSRR0 and SPU_WrSRR0 support nested interrupts by allowing software to save and restore SRR0 to a save area in local storage.
13. Synchronization and Ordering

The SPU provides a sequentially ordered programming model so that, with a few exceptions, all previous instructions appear to be finished before the next instruction is started.

Systems including an SPU often feature external devices with direct local storage access. Figure 13-1 shows a common organization where the external devices also communicate with the SPU via the channel interface. These systems are shared memory multiprocessors with message passing.

Figure 13-1. Systems with Multiple Accesses to Local Storage

Table 13-1 defines five transactions serviced by local storage. The SPU ISA does not define the behavior of the external device or how the external device accesses local storage. When this document refers to an external write of local storage, it assumes the external device delivers data to local storage such that a subsequent SPU load from local storage can retrieve the data.

Table 13-1. Local Storage Accesses

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>SPU load instruction gets data from local storage read.</td>
</tr>
<tr>
<td>Store</td>
<td>SPU store instruction sends data to local storage write.</td>
</tr>
<tr>
<td>Fetch</td>
<td>SPU instruction fetch gets data from local storage read.</td>
</tr>
<tr>
<td>ExtWrite</td>
<td>External device sends data to local storage write.</td>
</tr>
<tr>
<td>ExtRead</td>
<td>External device gets data from local storage read.</td>
</tr>
</tbody>
</table>

Interaction between local storage accesses of the external devices and those of the SPU can expose effects of SPU implementation-specific reordering, speculation, buffering, and caching. This section discusses how to order sequences of these transactions to obtain consistent results.
13.1 Speculation, Reordering, and Caching SPU Local Storage Access

SPU local storage access is weakly consistent (see PowerPC Virtual Environment Architecture, Book II). Therefore, the sequential execution model, as applied to instructions that cause storage accesses, guarantees only that those accesses appear to be performed in program order with respect to the SPU executing the instructions. These accesses might not appear to be performed in program order with respect to external local storage accesses or with respect to the SPU instruction fetch. This means that, in the absence of external local storage writes, an SPU load from any particular address returns the data written by the most recent SPU store to that address. However, an instruction fetch from that address does not necessarily return that data.

The SPU is allowed to cache, buffer, and otherwise reorder its local storage accesses. SPU loads, stores, and instruction fetches might or might not access the local storage. The SPU can speculatively read the local storage. That is, the SPU can read the local storage on behalf of instructions that are not required by the program. The SPU does not speculatively write local storage. If and when the SPU stores access local storage, the SPU only writes local storage on behalf of stores required by the program. Instruction fetches, loads, and stores can access local storage in any order.

13.2 SPU Internal Execution State

The channel interface can be used to modify the SPU internal execution state. An internal execution state is any state within an SPU, but outside local storage, that is modified through the channel interface and that can affect the sequence or execution of instructions. For example, programs can change SRR0 by writing the SPU_WrSRR0 channel, and SRR0 is the internal execution state. State changes made through the channel interface might not be synchronized with SPU program execution.

13.3 Synchronization Primitives

The SPU provides three synchronization instructions: dsync, sync, and sync.c. These instructions have both consistency and instruction serializing effects, as shown in Table 13-2 Synchronization Instructions on page 255. Programs can use the consistency effects of these primitives to ensure that the local storage state is consistent with SPU loads and stores. The instruction serializing effects allow the SPU program to order its local storage access.

The dsync instruction orders loads, stores, and channel accesses but not instruction fetches. When a dsync completes, the SPU will have completed all prior loads, stores, and channel accesses and will not have begun execution of any subsequent loads, stores, or channel accesses. At this time, an external read from a local storage address returns the data stored by the most recent SPU store to that address. SPU loads after the dsync return the data externally written before the moment when the dsync completes. The dsync instruction affects only SPU instruction sequencing and the consistency of loads and stores with respect to actual local storage state. The SPU does not broadcast dsync notification to external devices that access local storage, and, therefore, does not affect the state of the external devices.

The sync instruction is much like dsync, but it also orders instruction fetches. Instruction fetches from a local storage address after a sync instruction return data stored by the most recent store instruction or external write to that address. The sync instruction builds upon the sync instruction. It ensures that the effects upon the internal state caused by prior wrch instructions are propagated and influence the execution of the following instructions. SPU execution begins with a start event and ends with a stop event. Both start and stop perform sync.c.
Table 13-2. Synchronization Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Consistency Effects</th>
<th>Instruction Serialization Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsync</td>
<td>Ensures that subsequent external reads access data written by prior stores. Ensures that subsequent loads access data written by external writes.</td>
<td>Forces load and store access of local storage because of instructions before the <code>dsync</code> to be completed before completion of <code>dsync</code>. Forces read channel operations because of instructions before the <code>dsync</code> to be completed before completion of the <code>dsync</code>. Forces load and store access of local storage because of instructions after the <code>dsync</code> to occur after completion of the <code>dsync</code>. Forces read and write channel operations because of instructions after the <code>dsync</code> to occur after completion of the <code>dsync</code>.</td>
</tr>
<tr>
<td>sync</td>
<td>Ensures that subsequent external reads access data written by prior stores. Ensures that subsequent instruction fetches access data written by prior stores and external writes. Ensures that subsequent loads access data written by external writes.</td>
<td>Forces all access of local storage and channels because of instructions before the <code>sync</code> to be completed before completion of <code>sync</code>. Forces all access of local storage and channels because of instructions after the <code>sync</code> to occur after completion of the <code>sync</code>.</td>
</tr>
<tr>
<td>sync.c</td>
<td>Ensures that subsequent external reads access data written by prior stores. Ensures that subsequent instruction fetches access data written by prior stores and external writes. Ensures that subsequent loads access data written by external writes. Ensures that subsequent instruction processing is influenced by all internal execution states modified by previous wrch instructions.</td>
<td>Forces all access of local storage and channels because of instructions before the <code>sync.c</code> to be completed before completion of <code>sync.c</code>. Forces all access of local storage and channels because of instructions after the <code>sync.c</code> to occur after completion of the <code>sync.c</code>.</td>
</tr>
</tbody>
</table>

Table 13-3 indicates which synchronization primitives are required between actions that modify local storage and other reads and writes of local storage. SPU programs do not require synchronization primitives between their own load and store instructions in order for load instructions to get the data stored by the last preceding store instruction.

However, a program that stores into the instruction stream must execute a `sync` instruction before it reaches the newly stored instructions. The `sync` instruction forces the instruction fetch to read the instructions after the last store before the `sync` instruction. Without the `sync` instruction, the SPU might or might not execute the newly stored instruction. The SPU might execute the instruction in local storage at the time of the last `sync` event.

When an external access of local storage occurs, and it is clear that the external access is before or after a particular SPU access of local storage, synchronization is required to force the data to move between the SPU and the external device. Without synchronization, the external device might see a local storage state that is inconsistent with any point of execution in the SPU program.

For example, if an SPU program is to send data through local storage to an external reader, it must store the data and then execute a `dsync` instruction. If the external read occurs after the `dsync` instruction, it will read the stored data. If an SPU program is to load data put into local storage by an external writer, it must first execute a `dsync` instruction before it executes the load instruction. If the `dsync` instruction executes after the external write, the subsequent load instructions will be able to read the data stored by the external writer.
13.4 Caching SPU Local Storage Access

Implementations of the SPU can feature caches of local storage data for either instructions, data, or both. These caches must reflect data to and from the local storage when synchronization requires the state of local storage to be consistent. The **dsync** instruction ensures that modified data is visible to external devices that access local storage, and that data modified by these external devices is visible to subsequent loads and stores. The **sync** instructions also ensure that data modified by either stores or external puts is visible to a subsequent instruction fetch. For example, an instruction cache that does not snoop must be invalidated when **sync** is executed, and a copy-back data cache that does not snoop must be flushed and invalidated when either **sync** or **dsync** is executed.

13.5 Self-Modifying Code

SPU programs can store instructions in local storage and execute them. If the SPU has already read the instructions from local storage, before the store, the new instructions are not seen by SPU execution. Self-modifying code should always execute a **sync** instruction before executing the stored code. The **sync** instruction ensures that all stores complete before the next instruction is fetched from local storage.

13.6 External Local Storage Access

Loads and stores do not necessarily access local storage in program order. Accesses from external devices can be interleaved in ways that are inconsistent with program order. The **dsync** instruction forces all preceding loads and stores to complete their local storage access before allowing any further loads or stores to be initiated, while **sync** ensures that the next instruction is fetched after the **sync** instruction is executed. An external device can synchronize with an SPU program through local storage access.

**Table 13-4. Synchronizing Multiple Accesses to Local Storage**

<table>
<thead>
<tr>
<th>Writer</th>
<th>Local Storage Access to be Synchronized with the Local Storage Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtWrite</td>
<td>Store: <strong>dsync</strong>, Load: <strong>dsync</strong>, Fetch: <strong>sync</strong>, ExtRead: N/A, ExtWrite: N/A</td>
</tr>
</tbody>
</table>

*Note: The SPU ISA does not define how external readers and writers should order their accesses to local storage. Table 13-3 shows entries that relate to external readers and writers as “N/A.”*
again to prevent the data loads from being performed before the marker load. If such reordering were to occur, it would be possible for the marker write to occur between the reordered data loads and the delayed marker load. In this case, the data loads would receive stale data.

Table 13-4. Sending Data and Synchronizing through Local Storage

<table>
<thead>
<tr>
<th>External Device</th>
<th>SPU</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store data to C</td>
<td></td>
<td>Force a subsequent store to follow the store to C; that is, there will be no view of local storage where the marker is present in D but the data is not yet in C.</td>
</tr>
<tr>
<td>dsync</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store marker to D</td>
<td>dsync</td>
<td>Force the store to D to be visible in local storage to external readers.</td>
</tr>
<tr>
<td>eloop: Read D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>If not marker, goto eloop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 13-5. Receiving Data and Synchronizing through Local Storage

<table>
<thead>
<tr>
<th>External Device</th>
<th>SPU</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write data to A</td>
<td></td>
<td>This is the order in which the external device modifies local storage. The ordering is not controlled by the SPU ISA.</td>
</tr>
<tr>
<td>Write marker to B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop: dsync</td>
<td></td>
<td>Force a subsequent load to access local storage, so that the load arriving from B will get new data from local storage.</td>
</tr>
<tr>
<td>Load from B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>If not marker, goto loop</td>
<td></td>
<td>Ensure A and B are both written to local storage.</td>
</tr>
<tr>
<td>dsync</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load from A</td>
<td></td>
<td>Must get data from the write to A.</td>
</tr>
</tbody>
</table>

13.7 Speculation and Reordering of Channel Reads and Channel Writes

The SPU does not reorder or speculatively execute channel reads or channel writes. All operations at the channel interface represent instructions in the order they occur in the program.
13.8 Channel Interface with External Device

The channel interface delivers channel reads and writes to the SPU interface in program order, but there are no ordering guarantees with respect to load and stores. It is possible that a message sent to an external device may trigger the external device to directly access local storage. SPU programs might want to use either `sync` or `dsync` instructions, or both, to order SPU loads and stores relative to the external accesses. Table 13-6 shows how an SPU program might reliably send and receive data from an external device synchronizing through the channel interface.

Table 13-6. Synchronizing through the Channel Interface

<table>
<thead>
<tr>
<th>External Device</th>
<th>SPU</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPU receives data through local storage address A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write data to A</td>
<td></td>
<td>The ordering is not controlled by the SPU ISA.</td>
</tr>
<tr>
<td>Send message to channel B</td>
<td><code>rdch B</code></td>
<td>Wait for message</td>
</tr>
<tr>
<td></td>
<td><code>dsync</code></td>
<td>Ensure load from A is executed after <code>rdch</code>, and access the data in local storage</td>
</tr>
<tr>
<td>load from A</td>
<td></td>
<td>Must get data</td>
</tr>
<tr>
<td>SPU sends data through local storage address C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store data to C</td>
<td><code>dsync</code></td>
<td>Ensure data is in local storage</td>
</tr>
<tr>
<td></td>
<td><code>wrch D</code></td>
<td>Send message</td>
</tr>
<tr>
<td>Receive message from channel D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read data from C</td>
<td></td>
<td>The ordering is not controlled by the SPU ISA.</td>
</tr>
</tbody>
</table>

Note: The SPU architecture does not specify what actions an external device can perform in response to a channel read or write. The SPU does not wait for those actions to complete, and it does not synchronize the state of local storage before or after the channel operation.

13.9 Execution State Set by an SPU Program through the Channel Interface

Some SPU channels can control aspects of SPU execution state; for example, SRR0. State changes made through channel writes might not affect subsequent instructions. Execution of the `sync.c` instruction ensures that the new state does affect the next instruction.

13.10 Execution State Set by an External Device

Execution state changes made by an external device are ordered with respect to other externally requested state changes but not with respect to SPU instruction execution. The external device can stop the SPU, make execution state changes, start the SPU, and be certain the new state is visible to program execution.
## Appendix A. Instruction Table Sorted by Instruction Mnemonic

Table A-1. Instructions Sorted by Mnemonic (Page 1 of 6)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Add Word</td>
<td>60</td>
</tr>
<tr>
<td>absdb</td>
<td>Absolute Differences of Bytes</td>
<td>92</td>
</tr>
<tr>
<td>addx</td>
<td>Add Extended</td>
<td>66</td>
</tr>
<tr>
<td>ah</td>
<td>Add Halfword</td>
<td>58</td>
</tr>
<tr>
<td>ah1</td>
<td>Add Halfword Immediate</td>
<td>59</td>
</tr>
<tr>
<td>ai</td>
<td>Add Word Immediate</td>
<td>61</td>
</tr>
<tr>
<td>and</td>
<td>And</td>
<td>97</td>
</tr>
<tr>
<td>andbi</td>
<td>And Byte Immediate</td>
<td>99</td>
</tr>
<tr>
<td>andc</td>
<td>And with Complement</td>
<td>98</td>
</tr>
<tr>
<td>andhi</td>
<td>And Halfword Immediate</td>
<td>100</td>
</tr>
<tr>
<td>andi</td>
<td>And Word Immediate</td>
<td>101</td>
</tr>
<tr>
<td>avgb</td>
<td>Average Bytes</td>
<td>91</td>
</tr>
<tr>
<td>bg</td>
<td>Borrow Generate</td>
<td>70</td>
</tr>
<tr>
<td>bgx</td>
<td>Borrow Generate Extended</td>
<td>71</td>
</tr>
<tr>
<td>bi</td>
<td>Branch Indirect</td>
<td>178</td>
</tr>
<tr>
<td>bihnz</td>
<td>Branch Indirect If Not Zero Halfword</td>
<td>189</td>
</tr>
<tr>
<td>bihz</td>
<td>Branch Indirect If Zero Halfword</td>
<td>188</td>
</tr>
<tr>
<td>binz</td>
<td>Branch Indirect If Not Zero</td>
<td>187</td>
</tr>
<tr>
<td>bisl</td>
<td>Branch Indirect and Set Link</td>
<td>181</td>
</tr>
<tr>
<td>bisled</td>
<td>Branch Indirect and Set Link if External Data</td>
<td>180</td>
</tr>
<tr>
<td>biz</td>
<td>Branch Indirect If Zero</td>
<td>186</td>
</tr>
<tr>
<td>br</td>
<td>Branch Relative</td>
<td>174</td>
</tr>
<tr>
<td>bra</td>
<td>Branch Absolute</td>
<td>175</td>
</tr>
<tr>
<td>brasi</td>
<td>Branch Absolute and Set Link</td>
<td>177</td>
</tr>
<tr>
<td>brhinz</td>
<td>Branch If Not Zero Halfword</td>
<td>184</td>
</tr>
<tr>
<td>brhz</td>
<td>Branch If Zero Halfword</td>
<td>185</td>
</tr>
<tr>
<td>brnz</td>
<td>Branch If Not Zero Word</td>
<td>182</td>
</tr>
<tr>
<td>brsli</td>
<td>Branch Relative and Set Link</td>
<td>176</td>
</tr>
<tr>
<td>brz</td>
<td>Branch If Zero Word</td>
<td>183</td>
</tr>
<tr>
<td>cbd</td>
<td>Generate Controls for Byte Insertion (d-form)</td>
<td>40</td>
</tr>
<tr>
<td>cbx</td>
<td>Generate Controls for Byte Insertion (x-form)</td>
<td>41</td>
</tr>
<tr>
<td>cdd</td>
<td>Generate Controls for Doubleword Insertion (d-form)</td>
<td>46</td>
</tr>
<tr>
<td>cdx</td>
<td>Generate Controls for Doubleword Insertion (x-form)</td>
<td>47</td>
</tr>
<tr>
<td>ceq</td>
<td>Compare Equal Word</td>
<td>160</td>
</tr>
<tr>
<td>ceqb</td>
<td>Compare Equal Byte</td>
<td>156</td>
</tr>
</tbody>
</table>
### Table A-1. Instructions Sorted by Mnemonic (Page 2 of 6)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceqbi</td>
<td>Compare Equal Byte Immediate</td>
<td>157</td>
</tr>
<tr>
<td>ceqh</td>
<td>Compare Equal Halfword</td>
<td>158</td>
</tr>
<tr>
<td>ceqhi</td>
<td>Compare Equal Halfword Immediate</td>
<td>159</td>
</tr>
<tr>
<td>ceqi</td>
<td>Compare Equal Word Immediate</td>
<td>161</td>
</tr>
<tr>
<td>cflts</td>
<td>Convert Floating to Signed Integer</td>
<td>221</td>
</tr>
<tr>
<td>cftu</td>
<td>Convert Floating to Unsigned Integer</td>
<td>223</td>
</tr>
<tr>
<td>cg</td>
<td>Carry Generate</td>
<td>67</td>
</tr>
<tr>
<td>cgt</td>
<td>Compare Greater Than Word</td>
<td>166</td>
</tr>
<tr>
<td>cgtb</td>
<td>Compare Greater Than Byte</td>
<td>162</td>
</tr>
<tr>
<td>cgthi</td>
<td>Compare Greater Than Halfword Immediate</td>
<td>163</td>
</tr>
<tr>
<td>cgthi</td>
<td>Compare Greater Than Halfword</td>
<td>164</td>
</tr>
<tr>
<td>cgti</td>
<td>Compare Greater Than Word Immediate</td>
<td>165</td>
</tr>
<tr>
<td>cgx</td>
<td>Carry Generate Extended</td>
<td>167</td>
</tr>
<tr>
<td>chd</td>
<td>Generate Controls for Halfword Insertion (d-form)</td>
<td>68</td>
</tr>
<tr>
<td>chx</td>
<td>Generate Controls for Halfword Insertion (x-form)</td>
<td>42</td>
</tr>
<tr>
<td>clgt</td>
<td>Compare Logical Greater Than Word</td>
<td>172</td>
</tr>
<tr>
<td>clgtb</td>
<td>Compare Logical Greater Than Byte</td>
<td>168</td>
</tr>
<tr>
<td>clgtbi</td>
<td>Compare Logical Greater Than Byte Immediate</td>
<td>169</td>
</tr>
<tr>
<td>clgth</td>
<td>Compare Logical Greater Than Halfword</td>
<td>170</td>
</tr>
<tr>
<td>clgthi</td>
<td>Compare Logical Greater Than Halfword Immediate</td>
<td>171</td>
</tr>
<tr>
<td>clgti</td>
<td>Compare Logical Greater Than Word Immediate</td>
<td>173</td>
</tr>
<tr>
<td>clz</td>
<td>Count Leading Zeros</td>
<td>83</td>
</tr>
<tr>
<td>cntb</td>
<td>Count Ones in Bytes</td>
<td>84</td>
</tr>
<tr>
<td>csflt</td>
<td>Convert Signed Integer to Floating</td>
<td>220</td>
</tr>
<tr>
<td>cuflt</td>
<td>Convert Unsigned Integer to Floating</td>
<td>222</td>
</tr>
<tr>
<td>cwd</td>
<td>Generate Controls for Word Insertion (d-form)</td>
<td>44</td>
</tr>
<tr>
<td>cwx</td>
<td>Generate Controls for Word Insertion (x-form)</td>
<td>45</td>
</tr>
<tr>
<td>dfa</td>
<td>Double Floating Add</td>
<td>203</td>
</tr>
<tr>
<td>dfceq</td>
<td>Double Floating Compare Equal</td>
<td>226</td>
</tr>
<tr>
<td>dfcg</td>
<td>Double Floating Compare Greater Than</td>
<td>228</td>
</tr>
<tr>
<td>dfcmeq</td>
<td>Double Floating Compare Magnitude Equal</td>
<td>227</td>
</tr>
<tr>
<td>dfcmg</td>
<td>Double Floating Compare Magnitude Greater Than</td>
<td>229</td>
</tr>
<tr>
<td>dfm</td>
<td>Double Floating Multiply</td>
<td>207</td>
</tr>
<tr>
<td>dfma</td>
<td>Double Floating Multiply and Add</td>
<td>209</td>
</tr>
<tr>
<td>dfms</td>
<td>Double Floating Multiply and Subtract</td>
<td>213</td>
</tr>
<tr>
<td>dfnma</td>
<td>Double Floating Negative Multiply and Add</td>
<td>214</td>
</tr>
</tbody>
</table>
### Table A-1. Instructions Sorted by Mnemonic (Page 3 of 6)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfnms</td>
<td>Double Floating Multiply and Subtract</td>
<td>213</td>
</tr>
<tr>
<td>dfs</td>
<td>Double Floating Subtract</td>
<td>205</td>
</tr>
<tr>
<td>dftsv</td>
<td>Double Floating Test Special Value</td>
<td>230</td>
</tr>
<tr>
<td>dsync</td>
<td>Synchronize Data</td>
<td>243</td>
</tr>
<tr>
<td>eqv</td>
<td>Equivalent</td>
<td>114</td>
</tr>
<tr>
<td>fa</td>
<td>Floating Add</td>
<td>202</td>
</tr>
<tr>
<td>fceq</td>
<td>Floating Compare Equal</td>
<td>231</td>
</tr>
<tr>
<td>fcgt</td>
<td>Floating Compare Greater Than</td>
<td>233</td>
</tr>
<tr>
<td>fcmpeq</td>
<td>Floating Compare Magnitude Equal</td>
<td>232</td>
</tr>
<tr>
<td>fcmgt</td>
<td>Floating Compare Magnitude Greater Than</td>
<td>234</td>
</tr>
<tr>
<td>fesd</td>
<td>Floating Extend Single to Double</td>
<td>225</td>
</tr>
<tr>
<td>fi</td>
<td>Floating Interpolate</td>
<td>219</td>
</tr>
<tr>
<td>fm</td>
<td>Floating Multiply</td>
<td>206</td>
</tr>
<tr>
<td>fma</td>
<td>Floating Multiply and Add</td>
<td>208</td>
</tr>
<tr>
<td>fms</td>
<td>Floating Multiply and Subtract</td>
<td>212</td>
</tr>
<tr>
<td>fnms</td>
<td>Floating Negative Multiply and Subtract</td>
<td>210</td>
</tr>
<tr>
<td>frds</td>
<td>Floating Round Double to Single</td>
<td>224</td>
</tr>
<tr>
<td>frest</td>
<td>Floating Reciprocal Estimate</td>
<td>215</td>
</tr>
<tr>
<td>frsqest</td>
<td>Floating Reciprocal Absolute Square Root Estimate</td>
<td>217</td>
</tr>
<tr>
<td>fs</td>
<td>Floating Subtract</td>
<td>204</td>
</tr>
<tr>
<td>fscrrd</td>
<td>Floating-Point Status and Control Register Write</td>
<td>235</td>
</tr>
<tr>
<td>fsclr</td>
<td>Floating-Point Status and Control Register Read</td>
<td>236</td>
</tr>
<tr>
<td>fsm</td>
<td>Form Select Mask for Words</td>
<td>87</td>
</tr>
<tr>
<td>fsmb</td>
<td>Form Select Mask for Bytes</td>
<td>85</td>
</tr>
<tr>
<td>fsmbi</td>
<td>Form Select Mask for Bytes Immediate</td>
<td>55</td>
</tr>
<tr>
<td>fsmh</td>
<td>Form Select Mask for Halfwords</td>
<td>86</td>
</tr>
<tr>
<td>gb</td>
<td>Gather Bits from Words</td>
<td>90</td>
</tr>
<tr>
<td>gbb</td>
<td>Gather Bits from Bytes</td>
<td>88</td>
</tr>
<tr>
<td>gbh</td>
<td>Gather Bits from Halfwords</td>
<td>89</td>
</tr>
<tr>
<td>hbr</td>
<td>Hint for Branch (r-form)</td>
<td>192</td>
</tr>
<tr>
<td>hbra</td>
<td>Hint for Branch (a-form)</td>
<td>193</td>
</tr>
<tr>
<td>hbrrr</td>
<td>Hint for Branch Relative</td>
<td>194</td>
</tr>
<tr>
<td>heq</td>
<td>Halt If Equal</td>
<td>150</td>
</tr>
<tr>
<td>heqi</td>
<td>Halt If Equal Immediate</td>
<td>151</td>
</tr>
<tr>
<td>hgt</td>
<td>Halt If Greater Than</td>
<td>152</td>
</tr>
<tr>
<td>hgti</td>
<td>Halt If Greater Than Immediate</td>
<td>153</td>
</tr>
<tr>
<td>hght</td>
<td>Halt If Logically Greater Than</td>
<td>154</td>
</tr>
</tbody>
</table>
Table A-1. Instructions Sorted by Mnemonic (Page 4 of 6)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hlgti</td>
<td>Halt If Logically Greater Than Immediate</td>
<td>155</td>
</tr>
<tr>
<td>il</td>
<td>Immediate Load Word</td>
<td>52</td>
</tr>
<tr>
<td>ila</td>
<td>Immediate Load Address</td>
<td>53</td>
</tr>
<tr>
<td>ilh</td>
<td>Immediate Load Halfword</td>
<td>50</td>
</tr>
<tr>
<td>ilhu</td>
<td>Immediate Load Halfword Upper</td>
<td>51</td>
</tr>
<tr>
<td>iohl</td>
<td>Immediate Or Halfword Lower</td>
<td>54</td>
</tr>
<tr>
<td>iret</td>
<td>Interrupt Return</td>
<td>179</td>
</tr>
<tr>
<td>lnop</td>
<td>No Operation (Load)</td>
<td>240</td>
</tr>
<tr>
<td>lqa</td>
<td>Load Quadword (a-form)</td>
<td>34</td>
</tr>
<tr>
<td>lqd</td>
<td>Load Quadword (d-form)</td>
<td>32</td>
</tr>
<tr>
<td>lqr</td>
<td>Load Quadword Instruction Relative (a-form)</td>
<td>35</td>
</tr>
<tr>
<td>lqx</td>
<td>Load Quadword (x-form)</td>
<td>33</td>
</tr>
<tr>
<td>mfspr</td>
<td>Move from Special-Purpose Register</td>
<td>244</td>
</tr>
<tr>
<td>mpy</td>
<td>Multiply</td>
<td>72</td>
</tr>
<tr>
<td>mpya</td>
<td>Multiply and Add</td>
<td>76</td>
</tr>
<tr>
<td>mpyh</td>
<td>Multiply High</td>
<td>77</td>
</tr>
<tr>
<td>mpyhh</td>
<td>Multiply High High</td>
<td>79</td>
</tr>
<tr>
<td>mpyhha</td>
<td>Multiply High High and Add</td>
<td>80</td>
</tr>
<tr>
<td>mpyhhau</td>
<td>Multiply High High Unsigned and Add</td>
<td>82</td>
</tr>
<tr>
<td>mpyhu</td>
<td>Multiply High High Unsigned</td>
<td>81</td>
</tr>
<tr>
<td>mpyi</td>
<td>Multiply Immediate</td>
<td>74</td>
</tr>
<tr>
<td>mpyxs</td>
<td>Multiply and Shift Right</td>
<td>78</td>
</tr>
<tr>
<td>mpyu</td>
<td>Multiply Unsigned</td>
<td>73</td>
</tr>
<tr>
<td>mpyui</td>
<td>Multiply Unsigned Immediate</td>
<td>75</td>
</tr>
<tr>
<td>mtspr</td>
<td>Move to Special-Purpose Register</td>
<td>245</td>
</tr>
<tr>
<td>nand</td>
<td>Nand</td>
<td>112</td>
</tr>
<tr>
<td>nop</td>
<td>No Operation (Execute)</td>
<td>241</td>
</tr>
<tr>
<td>nor</td>
<td>Nor</td>
<td>113</td>
</tr>
<tr>
<td>or</td>
<td>Or</td>
<td>102</td>
</tr>
<tr>
<td>orbi</td>
<td>Or Byte Immediate</td>
<td>104</td>
</tr>
<tr>
<td>orc</td>
<td>Or with Complement</td>
<td>103</td>
</tr>
<tr>
<td>orhi</td>
<td>Or Halfword Immediate</td>
<td>105</td>
</tr>
<tr>
<td>ori</td>
<td>Or Word Immediate</td>
<td>106</td>
</tr>
<tr>
<td>orx</td>
<td>Or Across</td>
<td>107</td>
</tr>
<tr>
<td>rchcnt</td>
<td>Read Channel Count</td>
<td>249</td>
</tr>
<tr>
<td>rdch</td>
<td>Read Channel</td>
<td>248</td>
</tr>
<tr>
<td>rot</td>
<td>Rotate Word</td>
<td>129</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>roth</td>
<td>Rotate Halfword</td>
<td>127</td>
</tr>
<tr>
<td>rothi</td>
<td>Rotate Halfword Immediate</td>
<td>128</td>
</tr>
<tr>
<td>rothm</td>
<td>Rotate and Mask Halfword</td>
<td>136</td>
</tr>
<tr>
<td>rothmi</td>
<td>Rotate and Mask Halfword Immediate</td>
<td>137</td>
</tr>
<tr>
<td>roti</td>
<td>Rotate Word Immediate</td>
<td>130</td>
</tr>
<tr>
<td>rotm</td>
<td>Rotate and Mask Word</td>
<td>138</td>
</tr>
<tr>
<td>rotma</td>
<td>Rotate and Mask Algebraic Word</td>
<td>147</td>
</tr>
<tr>
<td>rotmah</td>
<td>Rotate and Mask Algebraic Halfword</td>
<td>145</td>
</tr>
<tr>
<td>rotmahi</td>
<td>Rotate and Mask Algebraic Halfword Immediate</td>
<td>146</td>
</tr>
<tr>
<td>rotmai</td>
<td>Rotate and Mask Algebraic Word Immediate</td>
<td>148</td>
</tr>
<tr>
<td>rotmi</td>
<td>Rotate and Mask Word Immediate</td>
<td>139</td>
</tr>
<tr>
<td>rotqbi</td>
<td>Rotate Quadword by Bits</td>
<td>134</td>
</tr>
<tr>
<td>rotqibi</td>
<td>Rotate Quadword by Bits Immediate</td>
<td>135</td>
</tr>
<tr>
<td>rotqby</td>
<td>Rotate Quadword by Bytes</td>
<td>131</td>
</tr>
<tr>
<td>rotqbybi</td>
<td>Rotate Quadword by Bytes from Bit Shift Count</td>
<td>133</td>
</tr>
<tr>
<td>rotqbyi</td>
<td>Rotate Quadword by Bytes Immediate</td>
<td>132</td>
</tr>
<tr>
<td>rotqmbi</td>
<td>Rotate and Mask Quadword by Bits</td>
<td>143</td>
</tr>
<tr>
<td>rotqmahi</td>
<td>Rotate and Mask Quadword by Bits Immediate</td>
<td>144</td>
</tr>
<tr>
<td>rotqmbi</td>
<td>Rotate and Mask Quadword by Bytes</td>
<td>140</td>
</tr>
<tr>
<td>rotqmby</td>
<td>Rotate and Mask Quadword Bytes from Bit Shift Count</td>
<td>142</td>
</tr>
<tr>
<td>rotqmbyi</td>
<td>Rotate and Mask Quadword by Bytes Immediate</td>
<td>141</td>
</tr>
<tr>
<td>selb</td>
<td>Select Bits</td>
<td>115</td>
</tr>
<tr>
<td>sf</td>
<td>Subtract from Word</td>
<td>64</td>
</tr>
<tr>
<td>sfh</td>
<td>Subtract from Halfword</td>
<td>62</td>
</tr>
<tr>
<td>sfhi</td>
<td>Subtract from Halfword Immediate</td>
<td>63</td>
</tr>
<tr>
<td>sfi</td>
<td>Subtract from Word Immediate</td>
<td>65</td>
</tr>
<tr>
<td>sfx</td>
<td>Subtract from Extended</td>
<td>69</td>
</tr>
<tr>
<td>shl</td>
<td>Shift Left Word</td>
<td>120</td>
</tr>
<tr>
<td>shlh</td>
<td>Shift Left Halfword</td>
<td>118</td>
</tr>
<tr>
<td>shlhi</td>
<td>Shift Left Halfword Immediate</td>
<td>119</td>
</tr>
<tr>
<td>shli</td>
<td>Shift Left Word Immediate</td>
<td>121</td>
</tr>
<tr>
<td>shlqbi</td>
<td>Shift Left Quadword by Bits</td>
<td>122</td>
</tr>
<tr>
<td>shlqbi</td>
<td>Shift Left Quadword by Bits Immediate</td>
<td>123</td>
</tr>
<tr>
<td>shlqby</td>
<td>Shift Left Quadword by Bytes</td>
<td>124</td>
</tr>
<tr>
<td>shlqbybi</td>
<td>Shift Left Quadword by Bytes from Bit Shift Count</td>
<td>126</td>
</tr>
<tr>
<td>shlqbyi</td>
<td>Shift Left Quadword by Bytes Immediate</td>
<td>125</td>
</tr>
<tr>
<td>shufb</td>
<td>Shuffle Bytes</td>
<td>116</td>
</tr>
</tbody>
</table>
### Table A-1. Instructions Sorted by Mnemonic (Page 6 of 6)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop</td>
<td>Stop and Signal</td>
<td>238</td>
</tr>
<tr>
<td>stopd</td>
<td>Stop and Signal with Dependencies</td>
<td>239</td>
</tr>
<tr>
<td>stqa</td>
<td>Store Quadword (a-form)</td>
<td>38</td>
</tr>
<tr>
<td>stqd</td>
<td>Store Quadword (d-form)</td>
<td>36</td>
</tr>
<tr>
<td>stqr</td>
<td>Store Quadword Instruction Relative (a-form)</td>
<td>39</td>
</tr>
<tr>
<td>stqx</td>
<td>Store Quadword (x-form)</td>
<td>37</td>
</tr>
<tr>
<td>sumb</td>
<td>Sum Bytes into Halfwords</td>
<td>93</td>
</tr>
<tr>
<td>sync</td>
<td>Synchronize</td>
<td>242</td>
</tr>
<tr>
<td>wrch</td>
<td>Write Channel</td>
<td>250</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive Or</td>
<td>108</td>
</tr>
<tr>
<td>xorbi</td>
<td>Exclusive Or Byte Immediate</td>
<td>109</td>
</tr>
<tr>
<td>xorhi</td>
<td>Exclusive Or Halfword Immediate</td>
<td>110</td>
</tr>
<tr>
<td>xori</td>
<td>Exclusive Or Word Immediate</td>
<td>111</td>
</tr>
<tr>
<td>xsbh</td>
<td>Extend Sign Byte to Halfword</td>
<td>94</td>
</tr>
<tr>
<td>xshw</td>
<td>Extend Sign Halfword to Word</td>
<td>95</td>
</tr>
<tr>
<td>xswd</td>
<td>Extend Sign Word to Doubleword</td>
<td>96</td>
</tr>
</tbody>
</table>
Appendix B. Details of the Generate Controls Instructions

The tables in this section show the details of the masks that are generated by the eight generate controls instructions. The masks that are shown are intended for use as the RC operand of the shuffle bytes, shufb, instruction. Each row in a table shows the rightmost 4 bits of the effective address. An x in the first column indicates an ignored bit. Blanks within the “created mask” are shown only to improve clarity.

See the following tables, as applicable:

- For byte insertion, see Table B-1 Byte Insertion: Rightmost 4 Bits of the Effective Address and Created Mask on page 265.
- For halfword insertion, see Table B-2 Halfword Insertion: Rightmost 4 Bits of the Effective Address and Created Mask on page 266.
- For word insertion, see Table B-3 Word Insertion: Rightmost 4 Bits of the Effective Address and Created Mask on page 266.
- For doubleword insertion, see Table B-4 Doubleword Insertion: Rightmost 4 Bits of Effective Address and Created Mask on page 266.

Table B-1. Byte Insertion: Rightmost 4 Bits of the Effective Address and Created Mask

<table>
<thead>
<tr>
<th>Rightmost 4 Bits of the Effective Address</th>
<th>Created Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>03 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0001</td>
<td>10 03 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0010</td>
<td>10 11 03 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0011</td>
<td>10 11 12 03 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0100</td>
<td>10 11 12 13 03 15 16 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0101</td>
<td>10 11 12 13 14 03 16 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0110</td>
<td>10 11 12 13 14 15 03 17 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>0111</td>
<td>10 11 12 13 14 15 16 03 18 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>1000</td>
<td>10 11 12 13 14 15 16 17 03 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>1001</td>
<td>10 11 12 13 14 15 16 17 03 18 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>1010</td>
<td>10 11 12 13 14 15 16 17 18 03 19 1a 1b 1c 1d 1e 1f</td>
</tr>
<tr>
<td>1011</td>
<td>10 11 12 13 14 15 16 17 18 1a 03 1c 1d 1e 1f</td>
</tr>
<tr>
<td>1100</td>
<td>10 11 12 13 14 15 16 17 18 19 1a 1b 03 1d 1e 1f</td>
</tr>
<tr>
<td>1101</td>
<td>10 11 12 13 14 15 16 17 18 19 1a 1b 1c 03 1e 1f</td>
</tr>
<tr>
<td>1110</td>
<td>10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 03 1f</td>
</tr>
<tr>
<td>1111</td>
<td>10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 03</td>
</tr>
</tbody>
</table>
### Table B-2. Halfword Insertion: Rightmost 4 Bits of the Effective Address and Created Mask

<table>
<thead>
<tr>
<th>Rightmost 4 Bits of the Effective Address</th>
<th>Created Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>0203 1213 1415 1617 1819 1a1b 1c1d 1e1f</td>
</tr>
<tr>
<td>001x</td>
<td>1011 0203 1415 1617 1819 1a1b 1c1d 1e1f</td>
</tr>
<tr>
<td>010x</td>
<td>1011 1213 0203 1617 1819 1a1b 1c1d 1e1f</td>
</tr>
<tr>
<td>011x</td>
<td>1011 1213 1415 0203 1819 1a1b 1c1d 1e1f</td>
</tr>
<tr>
<td>100x</td>
<td>1011 1213 1415 1617 1819 1a1b 1c1d 1e1f</td>
</tr>
<tr>
<td>101x</td>
<td>1011 1213 1415 1617 1819 0203 1c1d 1e1f</td>
</tr>
<tr>
<td>110x</td>
<td>1011 1213 1415 1617 1819 1a1b 0203 1e1f</td>
</tr>
<tr>
<td>111x</td>
<td>1011 1213 1415 1617 1819 1a1b 1c1d 1e1f</td>
</tr>
</tbody>
</table>

### Table B-3. Word Insertion: Rightmost 4 Bits of the Effective Address and Created Mask

<table>
<thead>
<tr>
<th>Rightmost 4 Bits of the Effective Address</th>
<th>Created Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx</td>
<td>0010203 14151617 18191a1b 1c1d1e1f</td>
</tr>
<tr>
<td>01xx</td>
<td>10111213 000010203 18191a1b 1c1d1e1f</td>
</tr>
<tr>
<td>10xx</td>
<td>10111213 14151617 000010203 1c1d1e1f</td>
</tr>
<tr>
<td>11xx</td>
<td>10111213 14151617 18191a1b 000010203</td>
</tr>
</tbody>
</table>

### Table B-4. Doubleword Insertion: Rightmost 4 Bits of Effective Address and Created Mask

<table>
<thead>
<tr>
<th>Rightmost 4 Bits of the Effective Address</th>
<th>Created Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>00102030304050607 18191a1b1c1d1e1f</td>
</tr>
<tr>
<td>1xxx</td>
<td>101112130314151617 0001020304050607</td>
</tr>
</tbody>
</table>
Glossary

**architecture**
A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible implementations.

**big-endian**
A byte-ordering method in memory where the address \( n \) of a word corresponds to the most significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most significant byte. See little-endian.

**bisled**
branch indirect and set link if external data instruction

**cache**
High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.

**CBEA**

**Cell Broadband Engine Architecture**
Extends the PowerPC 64-bit architecture with loosely coupled cooperative off-load processors. The Cell Broadband Engine Architecture provides a basis for the development of microprocessors targeted at the game, multimedia, and real-time market segments. The Cell Broadband Engine is one implementation of the Cell Broadband Engine Architecture.

**channel**
Channels are unidirectional, function-specific registers or queues. They are the primary means of communication between an SPE’s SPU and its MFC, which in turn mediates communication with the PPEs, other SPEs, and other devices. These other devices use MMIO registers in the destination SPE to transfer information on the channel interface of that destination SPE.

Specific channels have read or write properties, and blocking or nonblocking properties. Software on the SPU uses channel commands to enqueue DMA commands, query DMA and processor status, perform MFC synchronization, access auxiliary resources such as the decrementer (timer), and perform interprocessor-communication via mailboxes and signal-notification.

**DBZ**
Divide by zero.

**DIFF**
IEEE noncompliant result.

**DMA**
Direct memory access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

**double precision**
The specification that causes a floating-point value to be stored (internally) in the long format (two computer words).

**effective address**
An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective-address space is \( 2^{64} \) bytes.

**exception**
An error, unusual condition, or external signal that may alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled.
**fetch**
Retrieving instructions from either the cache or system memory and placing them into the instruction queue.

**floating point**
A way of representing real numbers (that is, values with fractions or decimals) in 32 bits or 64 bits. Floating-point representation is useful to describe very small or very large numbers.

**FPU**
Floating-point unit.

**fscrrd**
Floating-Point Status and Control Register read instruction.

**fscrwr**
Floating-Point Status and Control Register write instruction.

**general purpose register**
An explicitly addressable register that can be used for a variety of purposes (for example, as an accumulator or an index register).

**GPR**
See general purpose register.

**guarded**
Prevented from responding to speculative loads and instruction fetches. The operating system typically implements guarding, for example, on all I/O devices.

**implementation**
A particular processor that conforms to the architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of optional features.

**Inf**
Infinity.

**instruction cache**
A cache for providing program instructions to the processor faster than they can be obtained from system memory.

**INV**
Invalid operation.

**INX**
Inexact result.

**iohl**
Immediate or halfword lower instruction.

**iret**
Interrupt return instruction.

**ISA**
Instruction set architecture.

**KB**
Kilobyte.

**least significant bit**
The bit of least value in an address, register, data element, or instruction encoding.

**least significant byte**
The byte of least value in an address, register, data element, or instruction encoding.

**little-endian**
A byte-ordering method in memory where the address $n$ of a word corresponds to the least significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the most significant byte. See big-endian.

**local storage**
The storage associated with each SPE. It holds both instructions and data.

**LSA**
Local Storage Address. An address in the LS of an SPU, by which programs running in the SPU and DMA transfers managed by the MFC access the LS.
LSb  See least significant bit.

mask  A pattern of bits used to accept or reject bit patterns in another set of data. Hardware interrupts are enabled and disabled by setting or clearing a string of bits, with each interrupt assigned a bit position in a mask register.

MFC  Memory flow controller. It is part of an SPE and provides two main functions: moves data using DMA between the SPE’s local storage (LS) and main storage, and synchronizes the SPU with the rest of the processing units in the system.

mfspr  Move from special-purpose register instruction.

most significant bit  The highest-order bit in an address, registers, data element, or instruction encoding.

most significant byte  The highest-order byte in an address, registers, data element, or instruction encoding.

MSb  See most significant bit.

MSB  See most significant byte.

MSR  Machine state register.

mtspr  Move to special-purpose register instruction.

NaN  Not a number.

OVF  Overflow.

PC  Program counter.

PowerPC  Of or relating to the PowerPC Architecture or the microprocessors that implement this architecture.

PowerPC Architecture  A computer architecture that is based on the third generation of reduced instruction set computer (RISC) processors. The PowerPC architecture was developed jointly by Apple, Motorola, and IBM.


QNaN  Quiet NaN.

rchcnt  Read channel counter instruction.

rdch  Read from channel instruction.

RN0  Rounding control for slice 0 of the 2-way SIMD double-precision operations.

RN1  Rounding control for slice 1 of the 2-way SIMD double-precision operations.

RO  Relative offset.

ROH  Relative offset high.
### Instruction Set Architecture

#### Synergistic Processor Unit

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROL</td>
<td>relative offset low</td>
</tr>
<tr>
<td>RTL</td>
<td>register transfer language</td>
</tr>
<tr>
<td>shufb</td>
<td>shuffle bytes instruction</td>
</tr>
<tr>
<td>signal</td>
<td>Information sent on a signal-notification channel. These channels are inbound (to an SPE) registers. They can be used by a PPE or other processor to send information to an SPE. Each SPE has two 32-bit signal-notification registers, each of which has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor. Unlike mailboxes, they can be configured for either one-to-one or many-to-one signalling.</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single instruction, multiple data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.</td>
</tr>
<tr>
<td>SNaN</td>
<td>Signalling NaN.</td>
</tr>
<tr>
<td>snoop</td>
<td>To compare an address on a bus with a tag in a cache, to detect operations that violate memory coherency.</td>
</tr>
<tr>
<td>SPR</td>
<td>Special-purpose register.</td>
</tr>
<tr>
<td>SPU</td>
<td>Synergistic Processor Unit. The part of an SPE that executes instructions from its local storage (LS).</td>
</tr>
<tr>
<td>SRAM</td>
<td>static random access memory</td>
</tr>
<tr>
<td>sync</td>
<td>Synchronize command.</td>
</tr>
<tr>
<td>synchronization</td>
<td>The process of arranging storage operations to complete in the order of occurrence.</td>
</tr>
<tr>
<td>UNF</td>
<td>Underflow</td>
</tr>
<tr>
<td>vector</td>
<td>An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.</td>
</tr>
<tr>
<td>word</td>
<td>Four bytes.</td>
</tr>
<tr>
<td>wrch</td>
<td>Write to channel instruction.</td>
</tr>
</tbody>
</table>
Symbols

&, defined, 20
*, defined, 20
+, defined, 20
-, defined, 20
//, //, ///, defined, 19, 20
=, defined, 20
l, defined, 20
←, defined, 20
≥, defined, 20
≠, defined, 20
⊕, defined, 20
¬, defined, 20

Numerics

10-bit immediate, 19
16-bit immediate, 19
32-bit channels, 248, 250
32-bit registers, 244, 245
7-bit immediate, 19
8-bit immediate, 19

A

a, 60
absdb, 92
absolute differences of bytes, 92
add extended, 66
add halfword, 58
add halfword immediate, 59
add word, 60
add word immediate, 61
addition, two’s complement, 20
additional resources, 13
addx, 66
ah, 58
ahi, 59
ai, 61
algebraic right shift, 136, 137, 138, 139
and, 97
and (mnemonic), 97
and byte immediate, 99
and halfword immediate, 100
and with complement, 98
and word immediate, 101
AND, defined, 20
andi, 100
andh, 101
architectural overview, 25
assignment symbol, 20
audience, for manual, 13
average bytes, 91
avgb, 91

B

bg, 70
bgx, 71
bi, 178
bi instruction, 251
bihnz, 189
bihnz instruction, 251
bihz, 188
bihz instruction, 251
binary values in register RC and byte results, 116
binz, 187
binz instruction, 251
bisl, 181
bisl instruction, 251
bisled, 180
bisled instruction, 251
bit and byte numbering, 26–27
bit encoding, conventions for, 16
bit ordering, conventions for, 16
bit ranges, conventions for, 17
biz, 186
biz instruction, 251
borrow generate, 70
borrow generate extended, 71
br, 174
bra, 175
branch absolute, 175
branch absolute and set link, 177
branch if not zero halfword, 184
branch if not zero word, 182
branch if zero halfword, 185
branch if zero word, 183
branch indirect, 178
branch indirect and set link, 181
branch indirect and set link if external data, 180
branch indirect if not zero, 187
branch indirect if not zero halfword, 189
branch indirect if zero, 186
branch indirect if zero halfword, 188
branch instructions, 149–189
branch relative, 174
branch relative and set link, 176
brasl, 177
brhnz, 184
brhz, 185
brinst variable, 191
Synergistic Processor Unit

bmrz, 182
bssl, 176
brtarg variable, 191
brz, 183
byte insertion, 265
byte ordering, conventions for, 16

C

caching, SPU local storage access, 254, 256
carry generate, 67
carry generate extended, 68
cbd, 40
cbx, 41
cdd, 46
cdx, 47
ceq, 160
ceqb, 156
ceqbi, 157
ceqh, 158
ceqhi, 159
ceqi, 161
cflts, 221
cfltu, 223
cg, 67
cgt, 166
cgtb, 162
cgtbi, 163
cght, 164
cghti, 165
cgtl, 167
cgx, 68
channel instructions, 247–250
channel interface, 258
channel reads and writes, 257
channels, conventions for, 17
chd, 42
chx, 43
cigt, 172
cigtb, 168
cigtbi, 169
cigth, 170
cigtbi, 171
cigtl, 173
ciz, 83
cntb, 84
compare equal byte, 156
compare equal byte immediate, 157
compare equal halfword, 158
compare equal halfword immediate, 159
compare equal word, 160
compare equal word immediate, 161
compare greater than byte, 162
compare greater than byte immediate, 163
compare greater than halfword, 164
compare greater than halfword immediate, 165
compare greater than word, 166
compare greater than word immediate, 167
compare instructions, 149–189
compare instructions for floating point. See floating-point instructions
compare logical greater than byte, 168
compare logical greater than byte immediate, 169
compare logical greater than halfword, 170
compare logical greater than halfword immediate, 171
compare logical greater than word, 172
compare logical greater than word immediate, 173
compute-mask instructions. See generate controls instructions
conditional branch instructions, described, 149
conditional execution, 20
constant-formation instructions, 49–55
control instructions, 237–245
conventions used in this manual, 16–17
convert floating to signed integer, 221
convert floating to unsigned integer, 223
convert signed integer to floating, 220
convert unsigned integer to floating, 222
converting between single and double-precision formats, 198
count leading zeros, 83
count ones in bytes, 84
csflt, 220
cuflt, 222
cwd, 44
cwx, 45

d

data layout in registers, 28
data representation, 25
DBZ. See divide-by-zero (DBZ) exception condition
DENORM. See denormal input forced to zero (DENORM) exception condition
denormal input forced to zero (DENORM) exception condition, 198, 199
denorms, support for, 196
dfa, 203
dtceq, 226
dtcgt, 228
dtcmeq, 227
dtcmgt, 229
dfm, 207
dfma, 209
dfms, 213
dfna, 214
dfnm, 211
dfs, 205
dftsv, 230
divide-by-zero (DBZ) exception condition, 196
do ... while (cond), 20
document organization, 13
documents, related, 13
double floating add, 203
double floating compare equal, 226
double floating compare greater than, 228
double floating compare magnitude equal, 227
double floating compare magnitude greater than, 229
double floating multiply, 207
double floating multiply and add, 209
double floating multiply and subtract, 213
double floating negative multiply and add, 214
double floating negative multiply and subtract, 211
double floating subtract, 205
double floating test special value, 230
double-precision (IEEE mode) minimum and maximum values, 197
double-precision format, converting, 198
double-precision operations, 197
doubleword insertion, 266
doublewords
  bit and byte numbering, 26
dsync, 243
dsync instruction, 254, 255
  caching SPU local storage access and, 256
  external local storage access and, 256
  SPU loads and stores and, 258

deals sign, 20
equivalent, 114
eqv, 114
example LSLR values and corresponding local storage sizes, 31
exception conditions, 198
  denormal input forced to zero (DENORM), 199
  divide-by-zero (DBZ), 196
  IEEE noncompliant result (DIFF), 196
  inexact result (INX), 198
  invalid operation (INV), 199
  not propagated NaN, 199
  overflow (OVF), 196, 198
  underflow (UNF), 196, 199
exception settings for instructions, 200
exclusive or, 108
exclusive or byte immediate, 109
exclusive or halfword immediate, 110
exclusive OR symbol, 20
exclusive or word immediate, 111
execution state
  set by an external device, 258
  set by an SPU program through the channel interface, 258
execution state set by an SPU program through the channel interface, 258
extend sign byte to halfword, 94
extend sign halfword to word, 95
extend sign word to doubleword, 96
extended-range mode operations, 195
extending numbers, 195, 198
external device
  behavior of, 253
  channel interface and, 258
  setting execution state, 258
external local storage access, 256
extread transaction, 253
extwrite transaction, 253

F
fa, 202
fceuq, 231
fcgt, 233
fceq, 232
fcmgt, 234
feature bits (d) and (e), settings and results, 251
features of SPU ISA, 23
fesd, 225
fetch transaction, 253
fi, 219
fields, conventions for, 17
floating add, 202
floating compare equal, 231
floating compare greater than, 233
floating compare magnitude equal, 232
floating compare magnitude greater than, 234
floating extend single to double, 225
floating interpolate, 219
floating multiply, 206
floating multiply and add, 208
floating multiply and subtract, 212
floating negative multiply and subtract, 210
floating reciprocal absolute square root estimate, 217
floating reciprocal estimate, 215
floating round double to single, 224
floating subtract, 204
floating-point instructions, 195–236
Floating-Point Status and Control Register, 200–201
floating-point status and control register read, 236
floating-point status and control register write, 235
fm, 206
fma, 208
fms, 212
fnms, 210
for ... end, 20
form select mask for bytes, 85
form select mask for bytes immediate, 55
form select mask for halfwords, 86

Version.1.2
January 27, 2007

Synergistic Processor Unit

Execution State Set by an External Device

Synergistic Processor Unit
form select mask for words, 87
format
 for instruction descriptions, 15
 of Floating-Point Status and Control Register, 200
FPSCR. See Floating-Point Status and Control Register
frds, 224
frest, 215
frsqest, 217
fs, 204
fsccrd, 236
fsccrd instruction, 200
fsccrw, 235
fsccrw instruction, 200
fsm, 87
fsmb, 85
fsmbi, 55
fsmh, 86

G
 gather bits from bytes, 88
gather bits from halfwords, 89
gather bits from words, 90
gb, 90
gbb, 88
gbh, 89
general-purpose register fields, 19
generate controls
 for byte insertion (d-form), 40
 for byte insertion (x-form), 41
 for doubleword insertion (d-form), 46
 for doubleword insertion (x-form), 47
 for halfword insertion (d-form), 42
 for halfword insertion (x-form), 43
 for insertion instructions, 40–47
 for word insertion (d-form), 44
 for word insertion (x-form), 45
generate controls instructions, details, 265
GPR fields, 19

H, I, J, K
 halfword insertion, 266
 halfwords
 bit and byte numbering, 26
 halt if equal, 150
 halt if equal immediate, 151
 halt if greater than, 152
 halt if greater than immediate, 153
 halt if logically greater than, 154
 halt if logically greater than immediate, 155
 halt instructions, 149–189
 hbr, 192
 hbra, 193

hbrr, 194
heq, 150
heqi, 151
hgt, 152
hgti, 153
hint for branch (a-form), 193
hint for branch (r-form), 192
hint for branch relative, 194
hint-for-branch instructions, 191
hlgt, 154
hlgti, 155
I10, defined, 19
I16, defined, 19
I7, defined, 19
I8, defined, 19
IEEE noncompliant result (DIFF) exception condition, 196
IEEE Standard 754, 195
IEEE standard floating point versus SPU floating point, 195
IEEE standard, compliance with, 198
if (cond) then ... else ..., 20
il, 52
ila, 53
ilh, 50
ilhu, 51
immediate load address, 53
immediate load halfword, 50
immediate load halfword upper, 51
immediate load word, 52
immediate or halfword lower, 54
inexact result (INX) exception condition, 198
Inf. See infinity (Inf), support for
infinity (Inf), support for, 195
inline prefetching, 192
inserting bytes, 265
inserting doublewords, 266
inserting halfwords, 266
inserting words, 266
 instruction descriptions
 format for, 15
 how to use, 15
 instruction fields, 19
 instruction formats, 28
 instruction mnemonics, 259–264
 instruction operation notations, 20
 instructions
 branch, 149–189
 channel, 247–250
 compare, 149–189
 constant-formation, 49–55
 control, 237–245
 described, 16
 exception settings and, 196, 200
 floating point, 195–236
 halt, 149–189
 integer, 57–116
logical, 57–116
memory—load and store, 31–41
reserved field, 19
rotate, 117–148
shift, 117–148
sorted by mnemonic, 259
integer instructions, 57–116
internal execution state, 254
interrupt facility, 251, 252
interrupt handler, 251
interrupt return, 179
INV. See invalid operation (INV) exception condition
invalid operation (INV) exception condition, 199
INX. See inexact result (INX) exception condition
iohl, 54
iret, 179
iret instruction, 251, 252
ISA support, 23, 24

L

legal notices, 2
inop, 240
load and store instructions, 31–41
load quadword (a-form), 34
load quadword (d-form), 32
load quadword (x-form), 33
load quadword instruction relative (a-form), 35
load transaction, 253
load/store architecture, 23
local storage
  synchronizing multiple accesses to, 256
  synchronizing through, 257
local storage access. See SPU local storage access
local storage address, 20
Local Storage Limit Register, 20, 31
local storage transactions, 253
LocStor(x,y), 18
LocStor, defined, 20
logical comparison instructions, described, 149
logical instructions, 57–116
logical right shift, 136, 137, 138, 139
loops, 20
lqa, 34
lqd, 32
lqr, 35
lxq, 33
LSA. See local storage address
LSLR values, 31
LSLR. See Local Storage Limit Register

M

manual
  conventions for, 16
  organization of, 13
  purpose of, 13, 23
memory instructions, 31–41
mispr, 244
minimum and maximum values
  double-precision (IEEE mode), 197
  single-precision (extended-range mode), 195, 198
mnemonics, 16, 259–264
move from special-purpose register, 244
move to special-purpose register, 245
mpy, 72
mpya, 76
mpyh, 77
mpyhh, 79
mpyhhha, 80
mpyhhhu, 81
mpyi, 74
mpys, 78
mpyu, 73
mpyui, 75
mtspr, 245
multiply, 72
multiply and add, 76
multiply and shift right, 78
multiply high, 77
multiply high high, 79
multiply high high and add, 80
multiply high high unsigned, 81
multiply high high unsigned and add, 82
multiply immediate, 74
multiply unsigned, 73
multiply unsigned immediate, 75

N

NaN. See not a number (NaN), support for
NaN. See not propagated NaN exception condition
nand, 112
nand (mnemonic), 112
negative zero, 195
nested interrupts, support for, 252
no operation (execute), 241
no operation (load), 240
nontrap exception handling, 198
nop, 241
nor, 113
nor (mnemonic), 113
not a number (NaN), support for, 195
not equals sign, 20
not propagated NaN exception condition, 199
notations used in this manual, 16
numbers
  extending, 195, 198
  rounding, 198

O
OP or OPCD, defined, 19
opcode instruction field, 19
operands, described, 16
operations
  double-precision, 197
  single precision (extended-range mode), 195
or, 102
or (mnemonic), 102
or across, 107
or byte immediate, 104
or halfword immediate, 105
or with complement, 103
or word immediate, 106
OR, defined, 20
orbi, 104
orc, 103
ordering of transactions, 253
organization of manual, 13
orhi, 105
ori, 106
orx, 107
overflow (OVF) exception condition, 196, 198
OVF. See overflow (OVF) exception condition

P
positive zero, 195
primitives, synchronization, 254
program counter, 18
propagation of NaNs, 197
purpose of manual, 13, 23

Q
quadwords
  bit and byte numbering, 27

R
RA field, described, 19
RB field, described, 19
RC field, described, 19
rchcnt, 249
rdch, 248
read channel, 248
read channel count, 249
reference documents, 13
reference materials, 13
register layout of data types, 28
register transfer language (RTL) instruction definitions, 18
registers
  conventions for, 17
  data layout in, 28
reordering of channel reads and channel writes, 257
reordering SPU local storage access, 254
RepLeftBit(x,y), 18
representation
  of data, 25
  of zeros, 195
reserved fields, 19, 20
RI10 instruction format, 29
RI16 instruction format, 29
RI18 instruction format, 29
RI7 instruction format, 28
rot, 129
rotate and mask algebraic halfword, 145
rotate and mask algebraic halfword immediate, 146
rotate and mask algebraic word, 147
rotate and mask algebraic word immediate, 148
rotate and mask halfword, 136
rotate and mask halfword immediate, 137
rotate and mask quadword by bits, 143
rotate and mask quadword by bits immediate, 144
rotate and mask quadword by bytes, 140
rotate and mask quadword by bytes immediate, 141
rotate and mask quadword bytes from bit shift count, 142
rotate and mask word, 138
rotate and mask word immediate, 139
rotate halfword, 127
rotate halfword immediate, 128
rotate instructions, 117–148
rotate quadword by bits, 134
rotate quadword by bits immediate, 135
rotate quadword by bytes, 131
rotate quadword by bytes immediate, 132
rotate word, 129
rotate word immediate, 130
roth, 127
rothi, 128
rothm, 136
rothmi, 137
roti, 130
rotm, 138
rotma, 147
rotmah, 145
rotmahi, 146
rotmai, 148
rotmi, 139
rotnbi, 134
rotnbii, 135
rotqby, 131
rotqbybi, 133
rotqbyi, 132
rotqmbyi, 143
rotqmbyii, 144
rotqmbyii, 140
rotqmbybi, 142
rotqmbyi, 141
rounding control, slice 0, 200, 269
rounding control, slice 1, 200, 269
rounding mode, support for, 196
rounding modes, independent control of, 197
rounding numbers, 198
RR instruction format, 28
RRR instruction format, 28
RT field, defined, 19
RTL instruction definitions, 18

S
selb, 115
select bits, 115
self-modifying code, 256
setting execution state, 258
sf, 64
sfh, 62
sfhi, 63
sfli, 65
sfx, 69
shift instructions, 117–148
shift left halfword, 118
shift left halfword immediate, 119
shift left quadword by bits, 122
shift left quadword by bits immediate, 123
shift left quadword by bytes, 124
shift left quadword by bytes from bit shift count, 126
shift left quadword by bytes immediate, 125
shift left word, 120
shift left word immediate, 121
shl, 120
shih, 118
shii, 119
shi, 121
shlqbi, 122
shlqbi, 123
shlqby, 124
shlqbyi, 126
shlqbyii, 125
shultb, 116
shuffle instruction, 265
shuffle bytes, 116
signed comparison signs, 20
signed multiplication symbol, 20
single precision (extended-range mode) operations, 195
single-precision (extended-range mode) minimum and maximum values, 195, 198
single-precision format, converting, 198
special-purpose register, move from, 244
special-purpose register, move to, 245
speculation
of channel reads and channel writes, 257
of SPU local storage access, 253, 254
SPU architecture, 23, 25
SPU floating point versus IEEE standard floating point, 195
SPU internal execution state, 254
SPU interrupt facility, 251
SPU interrupt facility channels, 252
SPU interrupt handler, 251
SPU ISA, reference materials for, 13
SPU loads and stores and dsync and sync instructions, 258
SPU local storage access, 253
caching, 254, 256
reordering, 254
speculation of, 253, 254
SPU_RdSRR0 channel and interrupt facility, 252
SPU_WrSRR0 channel and interrupt facility, 252
SRR0 Register and SPU interrupt handler, 251
stop, 238
stop and signal, 238
stop and signal with dependencies, 239
stopd, 239
store quadword (a-form), 38
store quadword (d-form), 36
store quadword (x-form), 37
store quadword instruction relative (a-form), 39
store transaction, 253
stqa, 38
stqd, 36
stqr, 39
stqx, 37
subtract from extended, 69
subtract from halfword, 62
subtract from halfword immediate, 63
subtract from word, 64
subtract from word immediate, 65
subtraction, two’s complement, 20
sum bytes into halfwords, 93
sumb, 93
support for denoms (DENORM), infinity (INF), and not a number (NaN), 196
sync, 242
sync instruction, 254, 255
caching SPU local storage access, 256
self-modifying code and, 256
SPU loads and stores and, 258
cync.c instruction, 254, 255
synchronization, 253
synchronization instructions, 255
synchronization primitives, 254, 255
synchronization, ordering and, 253
synchronizing, 242
  multiple accesses to local storage, 256
  through channel interface, 258
  through local storage, 257
synchronizing data, 243
systems with multiple accesses to local storage, 253

T
temporary RTL names, 18
trademarks, 2
transaction ordering, 253
transaction synchronization, 253
truncation, support for, 196
two’s complement addition, 20
two’s complement subtraction, 20

U, V, W
u, defined, 20
unary minus, 20
unary NOT operator, 20
underflow (UNF) exception condition, 196, 199
UNF. See underflow (UNF) exception condition
unsigned comparison signs, 20
unsigned multiplication symbol, 20
word insertion, 266
words
  bit and byte numbering, 26
wrch, 250
write channel, 250

X
xor, 108
xorbi, 109
xorhi, 110
xori, 111
xsbh, 94
xshw, 95
xswd, 96

Z
zeros, representation of, 195