Synergistic Processor Unit

Instruction Set Architecture

Version 1.2

January 27, 2007

Printed in the United States of America January 2007

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com**

The IBM semiconductor solutions home page can be found at **ibm.com**/chips

Version 1.2
January 27, 2007

Contents

List of Figures

List of Tables

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.

Title	Version	Date
Cell Broadband Engine Architecture	1.01	October 2006
PowerPC User Instruction Set Architecture, Book I	2.02	January 2005
PowerPC Virtual Environment Architecture, Book II	2.02	January 2005
PowerPC Operating Environment Architecture, Book III	2.02	January 2005

Document Organization

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

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

Version Numbering

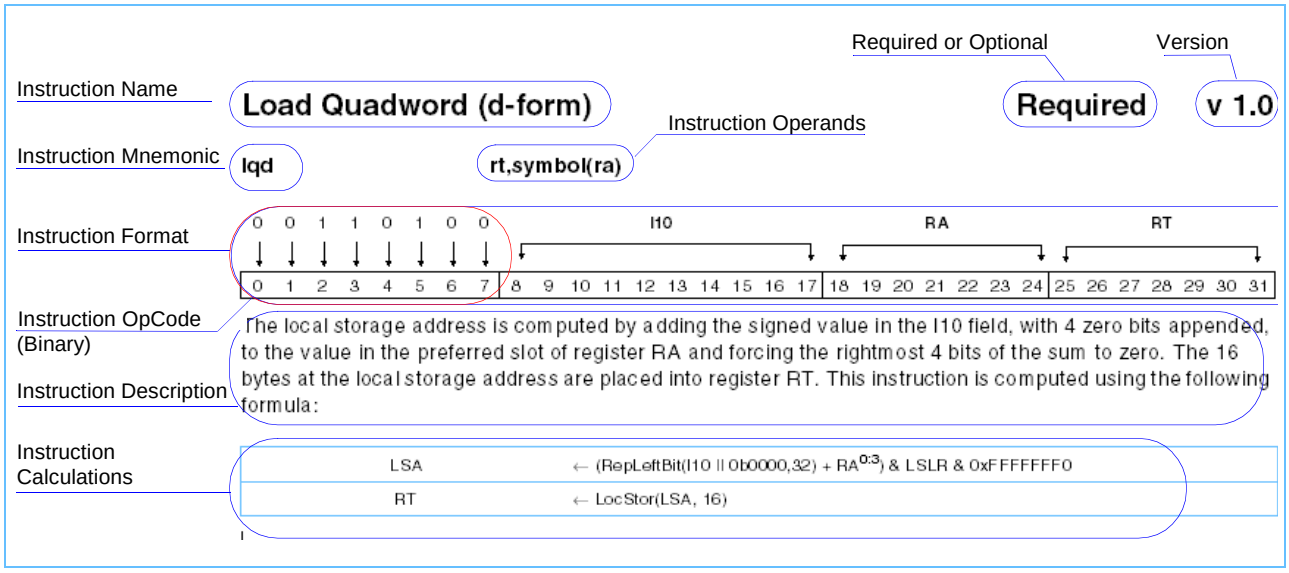
The document version number appears on the title page and in the footer of every page. The format of the version number is V.xy, where:

- 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



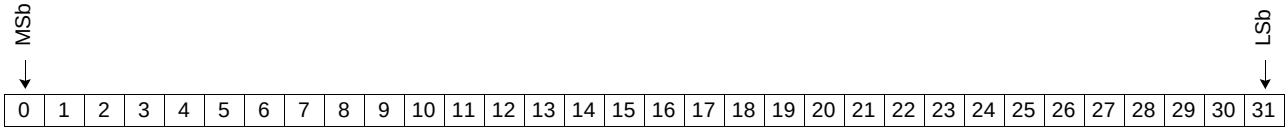
Conventions and Notations Used in This Manual

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.

Type of Reference	Format	Example
Reference to a specific register and a specific field using the register short name and the field name	Register_Short_Name[Field_Name]	MSR[R]
Reference to a field using the field name	[Field_Name]	[R]
Reference to a specific register and to multiple fields using the register short name and the field names	Register_Short_Name[Field_Name1, Field_Name2]	MSR[FE0, FE1]
Reference to a specific register and to multiple fields using the register short name and the bit positions.	Register_Short_Name[Bit_Number, Bit_Number]	MSR[52, 55]
Reference to a specific register and to a field using the register short name and the bit position or the bit range.	Register_Short_Name[Bit_Number]	MSR[52]
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]	MSR[39:44]
A field name followed by an equal sign (=) and a value indicates the value for that field.	Register_Short_Name[Field_Name]=*n*[1]	MSR[FE0]=0b1 MSR[FE]=0x1
	Register_Short_Name[Bit_Number]=*n*[1]	MSR[52]=0b0 MSR[52]=0x0
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]=*n*[1]	MSR[39:43]=0b10010 MSR[39:43]=0x11

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

Temporary Name	Width
b, byte, byte1, byte2, c	8 bits
r, s	16 bits
bbbb, EA, QA, t, t0, t1, t2, t3, u, v	32 bits
Q, R, Memdata	128 bits
Rconcat	256 bits
i, j, k, m	Meta (for description only)

Instruction Fields

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

Table ii. Instruction Fields

Field	Description
/, //, ///	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.
I7	7-bit immediate
I8	8-bit immediate
I10	10-bit immediate
I16	16-bit immediate
OP or OPCD	Opcode
RA[18-24]	Field used to specify a general-purpose register (GPR) to be used as a source or as a target.
RB[11-17]	Field used to specify a GPR to be used as a source or as a target.
RC[4-10]	Field used to specify a GPR to be used as a source or as a target.
RT[25-31]	Field used to specify a GPR to be used as a target.

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

Notation	Description	See Note
X_p	Means bit *p* of register or value field *X*	
$X_{p:q}$	Means bits *p* through q inclusive of register or value *X*	
X^p	Means byte *p* of register or value *X*	
$X^{p:q}$	Means bytes *p* through *q* inclusive of register or value *X*	
$X_{p::q}$	Means bits *p* and the bits that follow for a total of *q* bits	
$X^{p::q}$	Means bytes *p* and the bytes that follow for a total of *q* bytes	
${}_p0$ and ${}_p1$	Mean a string of *p 0 bits* and of *p 1 bits*.	1
¬	unary NOT operator	2
, ||	Signed multiplication, Unsigned multiplication	3
+	Two's complement addition	2
-	Two's complement subtraction, unary minus	2
= ≠	Equals Not Equals relations	
<, ≤, >, ≥	Signed comparison relations	
$<u, >u$	Unsigned comparison relations	
&	AND	2
|	OR	2
⊕	Exclusive OR (a & ¬b | ¬a & b)	2
←	Assignment	
LSA	Local Storage Address	
LSLR	Local Storage Limit Register	
LocStor(LSA,*width*)	Contents of the number of bytes indicated by the *width* variable in local storage at the LSA.	
if (cond) then ... else ...	Conditional execution. Else is optional. The range of the then and else clauses is indicated by indention. When the clauses are single statements, they are shown on the same line as the corresponding if and else.	
for ... end	For loop. *To* and *by* clauses specify incrementing an iteration variable, and a *while* clause provides termination conditions.	
do ... while (cond)	Do loop. *While* clause provides termination conditions.	
/, //, ///	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.	

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.

Revision Date	Contents of Modification	Errata?
January 27, 2007	Version 1.2 • Revised the introduction to the revision log (see *Revision Log* on page 21). • 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). • Corrected and clarified the programming note associated with the Multiply High instructions and added a code sample (see *Multiply High* on page 77). • Deleted “nonzero” from the description of an IEEE noncompliant result (see *Section 9.1 Single Precision (Extended-Range Mode)* on page 195). • 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). • Changed the description of handling denormal inputs (see *Section 9.2.1 Conversions Between Single-Precision and Double-Precision Format* on page 198). • Deleted “nonzero” from the description of FPSCR[31] (see *Section 9.3 Floating-Point Status and Control Register* on page 200). • 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). • 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). • Added the new instructions to *Appendix A* (see *Table A-1 Instructions Sorted by Mnemonic* on page 259). • Made various editorial changes to the glossary (see *Glossary* on page 267). • 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.	

Revision Date	Contents of Modification	Errata?
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 **bisled** 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 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)

Feature	Description
128-bit SIMD execution unit organization	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.
Software-managed memory	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).
Load/store architecture to support efficient static random access memory (SRAM) design	The SPU ISA microarchitecture is organized to enable efficient implementations that use single-ported (local storage) memory.
Large unified register file	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.
ISA support to eliminate branches	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.

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

Feature	Description
ISA support to avoid branch penalties on predictable branches	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.
Graphics-oriented single-precision (extended-range) floating-point support	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.
Channel architecture	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.
User-only architecture	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*.

Table 2-1. Bit and Byte Numbering Figures

For a figure that shows…	See…
Bit and Byte Numbering of Halfwords	*Figure 2-1* on page 26
Bit and Byte Numbering of Words	*Figure 2-2* on page 26
Bit and Byte Numbering of Doublewords	*Figure 2-3* on page 26
Bit and Byte Numbering of Quadwords	*Figure 2-4* on page 27
Register Layout of Data Types	*Figure 2-5* on page 28

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

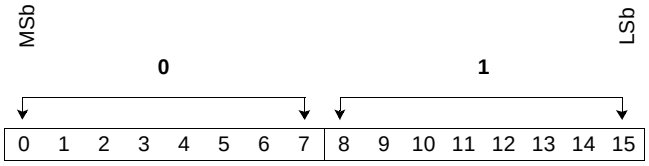


Figure 2-2. Bit and Byte Numbering of Words

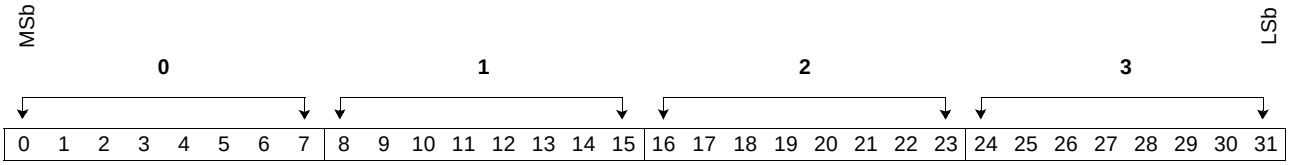


Figure 2-3. Bit and Byte Numbering of Doublewords

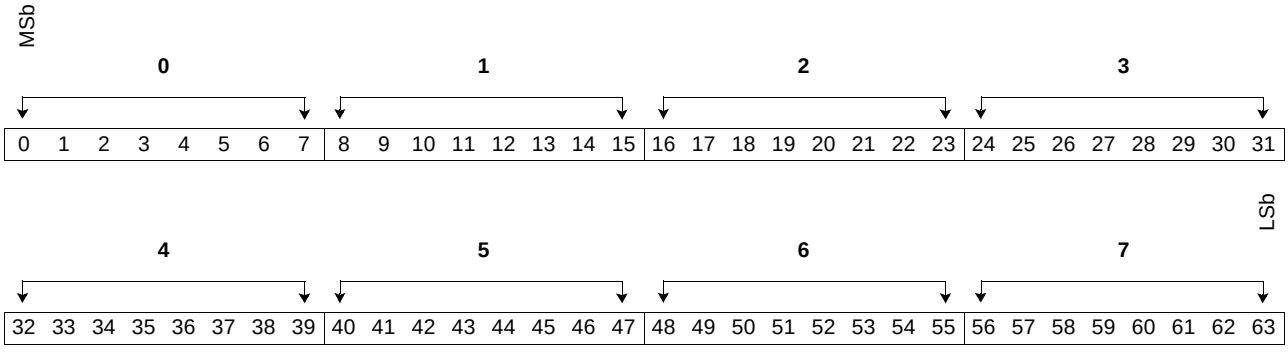
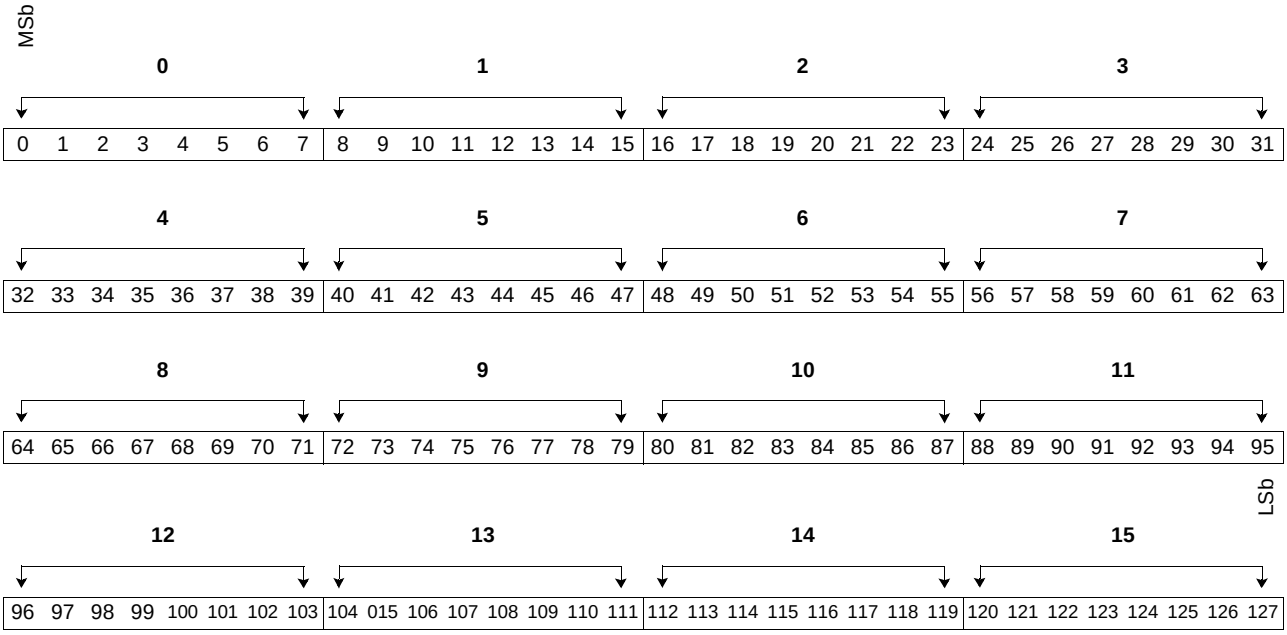


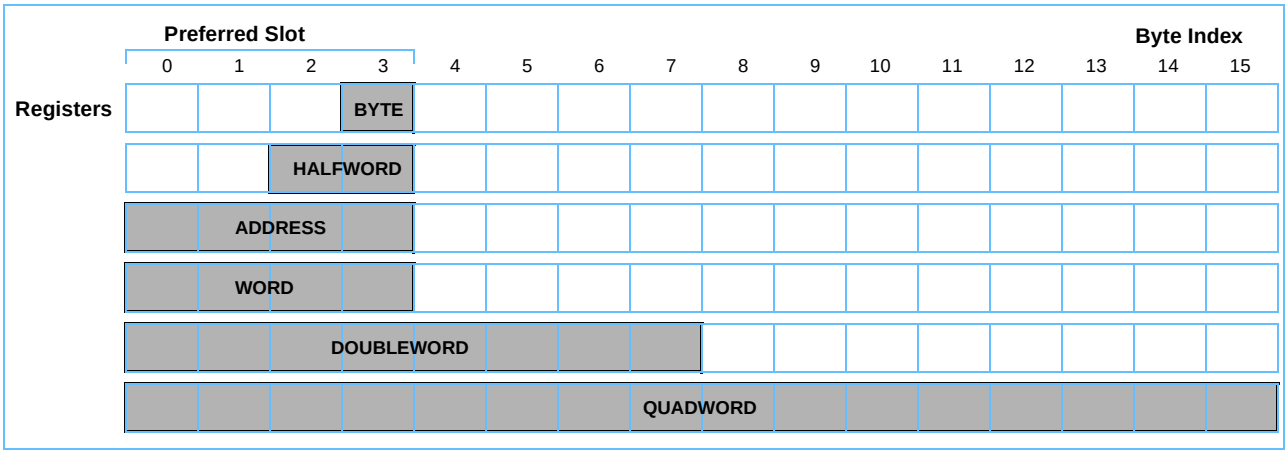
Figure 2-4. Bit and Byte Numbering of Quadwords



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



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*

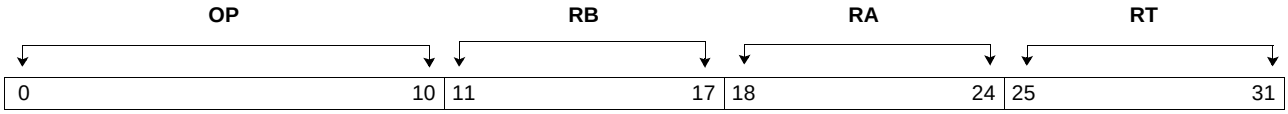


*Figure 2-7. **RRR** Instruction Format*

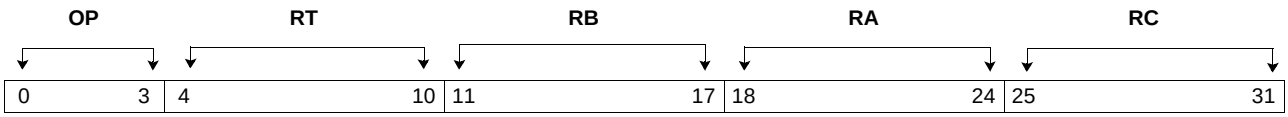


*Figure 2-8. **RI7** Instruction Format*

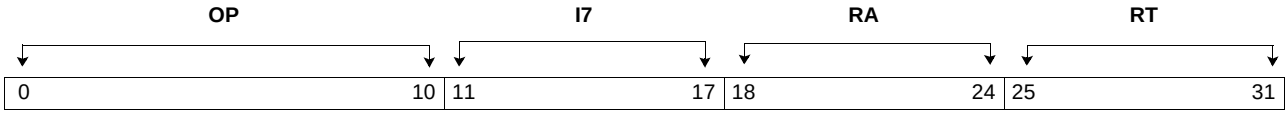


*Figure 2-9. **RI10** Instruction Format*

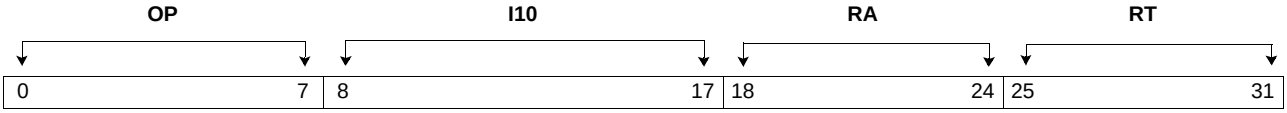


*Figure 2-10. **RI16** Instruction Format*

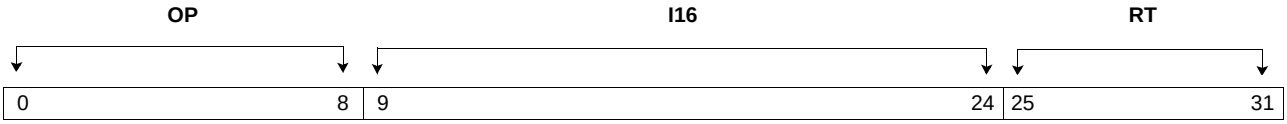
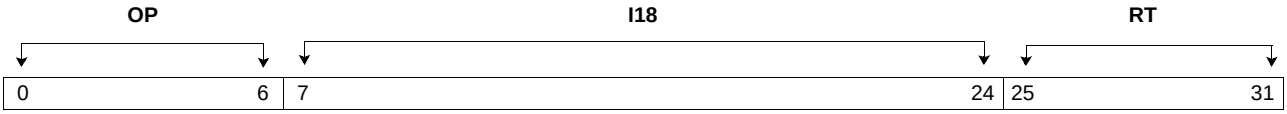


*Figure 2-11. **RI18** Instruction Format*



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.

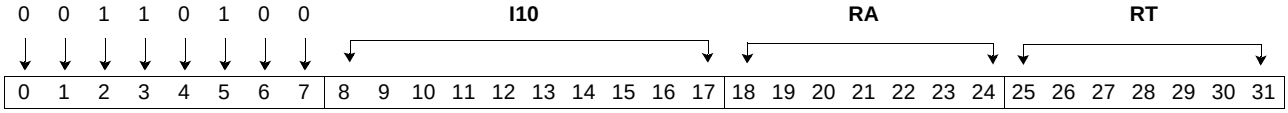
In a typical system configuration, the SPU local storage is externally accessible. The possibility therefore exists of SPU memory being modified asynchronously during the course of execution of an SPU program. All references (loads, stores) to local storage by an SPU program, and aligned external references 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 3-1. Example LSLR Values and Corresponding Local Storage Sizes

LSLR	Local Storage Size
0x0003 FFFF	256 KB
0x0001 FFFF	128 KB
0x0000 FFFF	64 KB
0x0000 7FFF	32 KB

Load Quadword (d-form) Required v 1.0

lqd rt,symbol(ra)

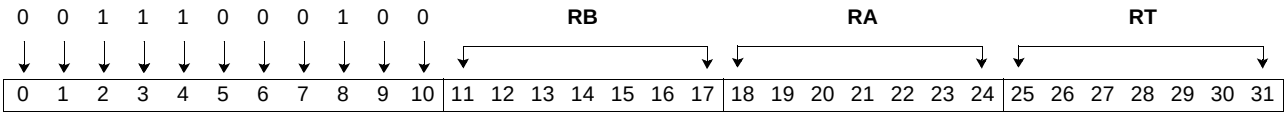


The local storage address is computed by adding the signed value in the I10 field, with 4 zero bits appended, to the value in the preferred slot of register RA and forcing the rightmost 4 bits of the sum to zero. The 16 bytes at the local storage address are placed into register RT. This instruction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

LSA	← (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$) & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA, 16)

Load Quadword (x-form) — Required — v 1.0

lqx **rt,ra,rb**

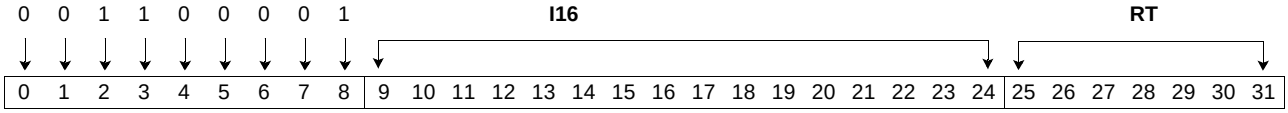


The local storage address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the rightmost 4 bits of the sum to zero. The 16 bytes at the local storage address are placed into register RT. This instruction is computed using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

LSA	$\leftarrow (RA^{0:3} + RB^{0:3})$ & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA,16)

Load Quadword (a-form) Required v 1.0

lqa rt,symbol

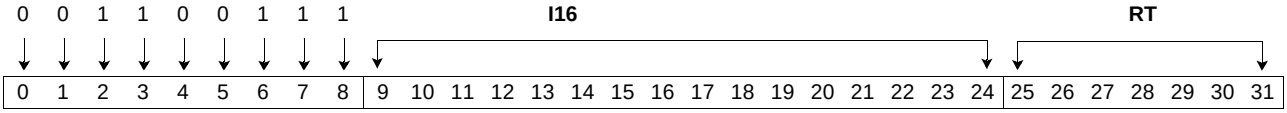


The value in the I16 field, with 2 zero bits appended and extended on the left with copies of the most-significant bit, is used as the local storage address. The 16 bytes at the local storage address are loaded into register RT.

LSA	← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA,16)

Load Quadword Instruction Relative (a-form) Required v 1.0

lqr rt,symbol

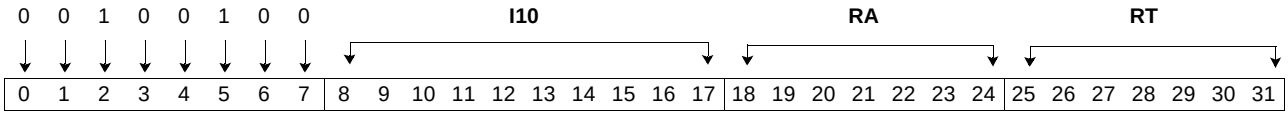


The value in the I16 field, with 2 zero bits appended, is added to the program counter (PC) to form the local storage address. The 16 bytes at the local storage address are loaded into register RT.

LSA	← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0
RT	← LocStor(LSA,16)

Store Quadword (d-form) Required v 1.0

stqd rt,symbol(ra)

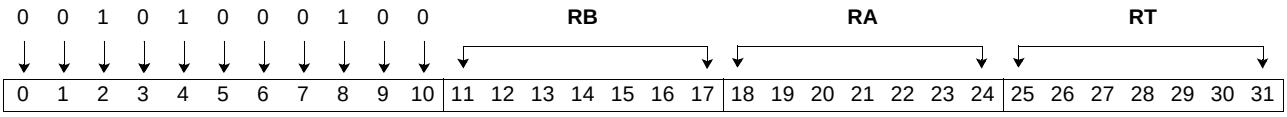


The local storage address is computed by adding the signed value in the I10 field, with 4 zero bits appended, to the value in the preferred slot of register RA and forcing the rightmost 4 bits of the sum to zero. The contents of register RT are stored at the local storage address.

LSA	← (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$) & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Store Quadword (x-form) — Required — v 1.0

stqx **rt,ra,rb**

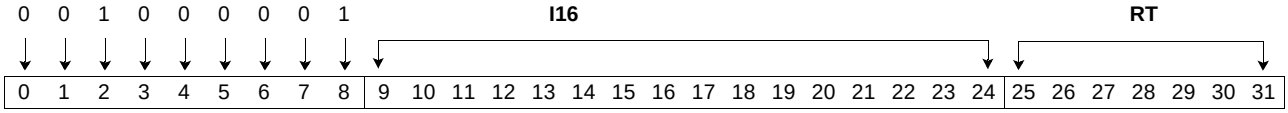


The local storage address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the rightmost 4 bits of the sum to zero. The contents of register RT are stored at the local storage address.

LSA	← $(RA^{0:3} + RB^{0:3})$ & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Store Quadword (a-form) Required v 1.0

stqa rt,symbol

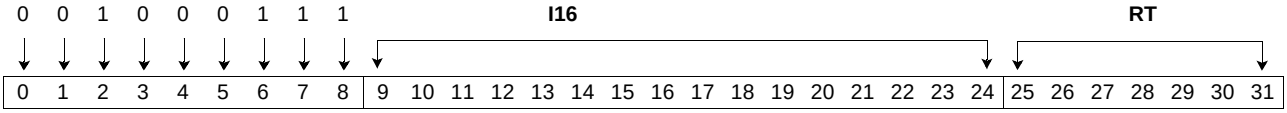


The value in the I16 field, with 2 zero bits appended and extended on the left with copies of the most-significant bit, is used as the local storage address. The contents of register RT are stored at the location given by the local storage address.

LSA	← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Store Quadword Instruction Relative (a-form) Required v 1.0

stqr rt,symbol

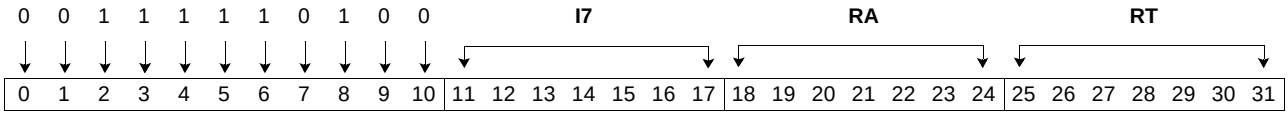


The value in the I16 field, with two zero bits appended and extended on the left with copies of the most-significant bit, is added to the program counter (PC) to form the local storage address. The contents of register RT are stored at the location given by the local storage address.

LSA	← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR & 0xFFFFFFF0
LocStor(LSA,16)	← RT

Generate Controls for Byte Insertion (d-form) Required v 1.0

cbd **rt,symbol(ra)**

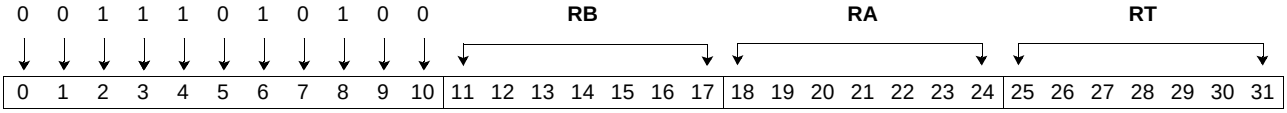


A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA. The address is used to determine the position of the addressed byte within a quadword. Based on the position, a mask is generated that can be used with the Shuffle Bytes (**shufb**) 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.

t	← $(RA^{0:3}$ + RepLeftBit(I7,32)) & 0x0000000F
RT	← 0x101112131415161718191A1B1C1D1E1F
RT^t	← 0x03

Generate Controls for Byte Insertion (x-form) Required v 1.0

cbx **rt,ra,rb**

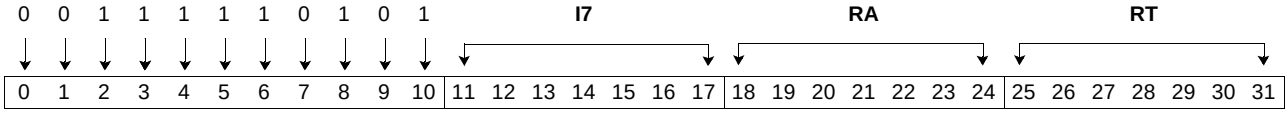


A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB. The address is used to determine the position of the addressed byte within a quadword. Based on the position, a mask is generated that can be used with the **shufb** 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.

t	$\leftarrow (RA^{0:3} + RB^{0:3}) \& 0x0000000F$
RT	$\leftarrow$ 0x101112131415161718191A1B1C1D1E1F
RT^t	$\leftarrow$ 0x03

Generate Controls for Halfword Insertion (d-form) Required v 1.0

chd **rt,symbol(ra)**

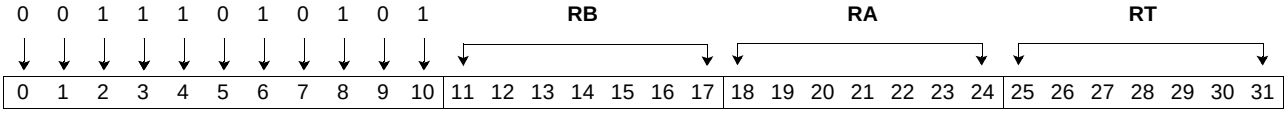


A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA and forcing the least-significant bit to zero. The address is used to determine the position of an aligned halfword within a quadword. Based on the position, a mask is generated that 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.

t	$\leftarrow (RA^{0:3} + \text{RepLeftBit}(I7,32)) \,\&\, 0x0000000E$
RT	$\leftarrow$ 0x101112131415161718191A1B1C1D1E1F
$RT^{t::2}$	$\leftarrow$ 0x0203

Generate Controls for Halfword Insertion (x-form) Required v 1.0

chx **rt,ra,rb**

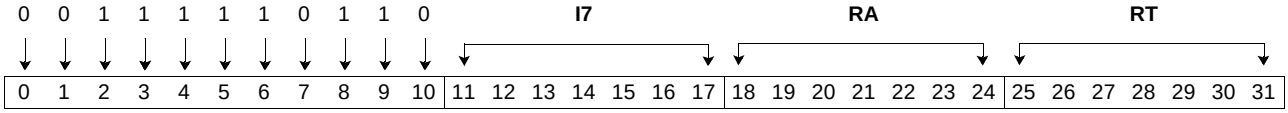


A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the least-significant bit to zero. The address is used to determine the position of an aligned halfword within a quadword. Based on the position, a mask is 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.

t	$\leftarrow (RA^{0:3} + RB^{0:3})$ & 0x0000000E
RT	← 0x101112131415161718191A1B1C1D1E1F
$RT^{t:2}$	← 0x0203

Generate Controls for Word Insertion (d-form) Required v 1.0

cwd **rt,symbol(ra)**

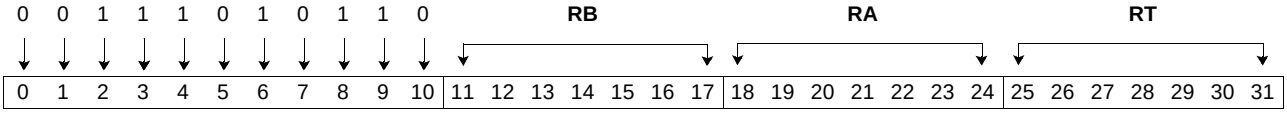


A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA and forcing the least-significant 2 bits to zero. The address is used to determine the position of an aligned word within a quadword. Based on the position, a mask is generated that can be used with the **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.

t	$\leftarrow (RA^{0:3} + \text{RepLeftBit}(I7,32)) \,\&\, 0x0000000C$
RT	$\leftarrow$ 0x101112131415161718191A1B1C1D1E1F
$RT^{t::4}$	$\leftarrow$ 0x00010203

Generate Controls for Word Insertion (x-form) Required v 1.0

cwx **rt,ra,rb**

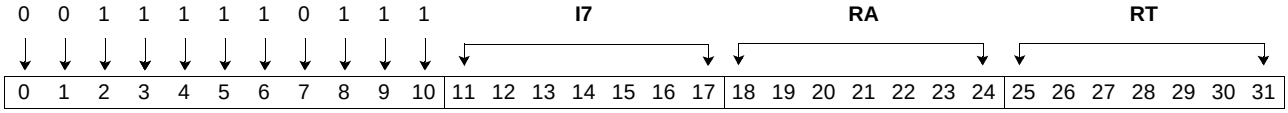


A 4-bit address is computed by adding the value in the preferred slot of register RA to the value in the preferred slot of register RB and forcing the least-significant 2 bits to zero. The address is used to determine the position of an aligned word within a quadword. Based on the position, a mask is 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.

t	$\leftarrow (RA^{0:3} + RB^{0:3}) \& 0x0000000C$
RT	$\leftarrow$ 0x101112131415161718191A1B1C1D1E1F
$RT^{t::4}$	$\leftarrow$ 0x00010203

Generate Controls for Doubleword Insertion (d-form) Required v 1.0

cdd rt,symbol(ra)

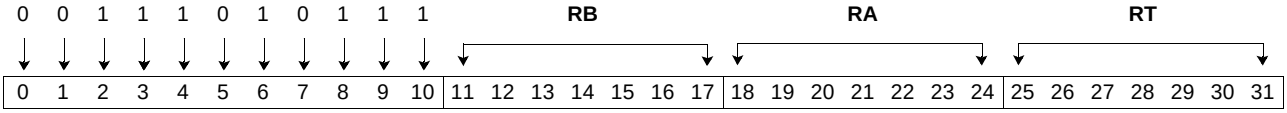


A 4-bit address is computed by adding the value in the signed I7 field to the value in the preferred slot of register RA and forcing the least-significant 3 bits to zero. The address is used to determine the position of an aligned doubleword within a quadword. Based on the position, a mask is generated 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	$\leftarrow (RA^{0:3} + RepLeftBit(I7,32)) \& 0x00000008$
RT	$\leftarrow 0x101112131415161718191A1B1C1D1E1F$
$RT^{t::8}$	$\leftarrow 0x0001020304050607$

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.

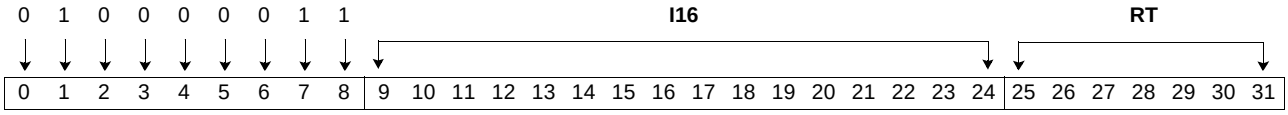
t	$\leftarrow (RA^{0:3} + RB^{0:3}) \& 0x00000008$
RT	$\leftarrow$ 0x101112131415161718191A1B1C1D1E1F
$RT^{t:8}$	$\leftarrow$ 0x0001020304050607

4. Constant-Formation Instructions

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

Immediate Load Halfword — Required — v 1.0

ilh **rt,symbol**



For each of eight halfword slots:

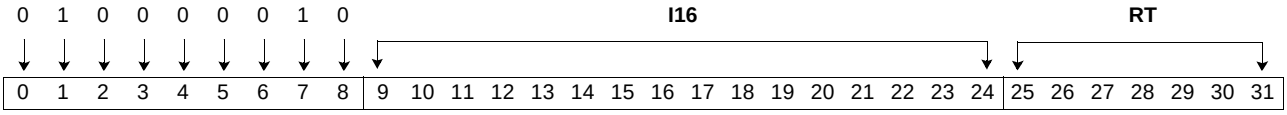
- The value in the I16 field is placed in register RT.

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

s	← I16
$RT^{0:1}$	← s
$RT^{2:3}$	← s
$RT^{4:5}$	← s
$RT^{6:7}$	← s
$RT^{8:9}$	← s
$RT^{10:11}$	← s
$RT^{12:13}$	← s
$RT^{14:15}$	← s

Immediate Load Halfword Upper — Required — v 1.0

ilhu **rt,symbol**



For each of four word slots:

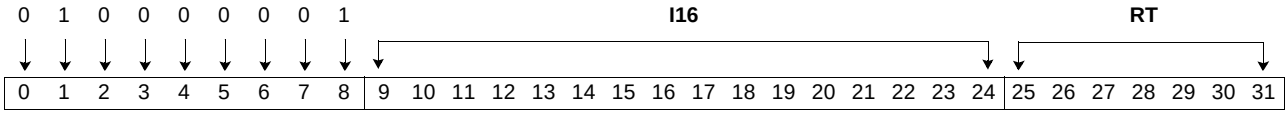
- The value in the I16 field is placed in the leftmost 16 bits of the word.
- The remaining bits of the word are set to zero.

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.

t	← I16 || 0x0000
$RT^{0:3}$	← t
$RT^{4:7}$	← t
$RT^{8:11}$	← t
$RT^{12:15}$	← t

Immediate Load Word — Required — v 1.0

il **rt,symbol**



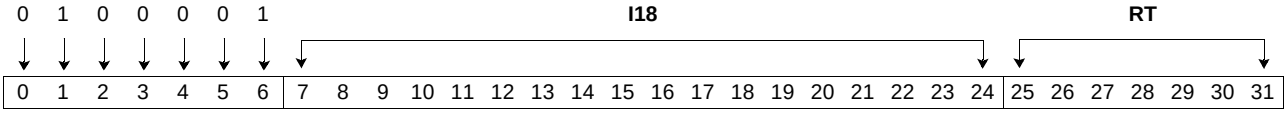
For each of four word slots:

- 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 — Required — v 1.0

ila **rt,symbol**



For each of four word slots:

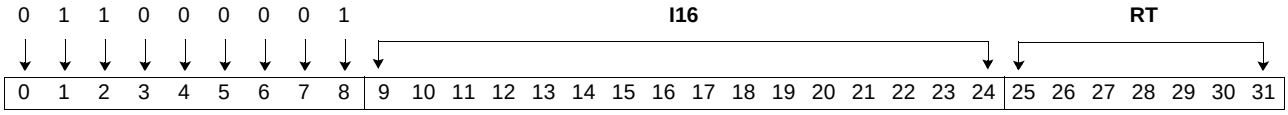
- The value in the I18 field is placed unchanged in the rightmost 18 bits of register RT.
- The remaining bits of register RT are set to zero.

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	$\leftarrow {}_{14}0 \parallel I18$
$RT^{0:3}$	$\leftarrow t$
$RT^{4:7}$	$\leftarrow t$
$RT^{8:11}$	$\leftarrow t$
$RT^{12:15}$	$\leftarrow t$

Immediate Or Halfword Lower — Required — v 1.0

iohl **rt,symbol**



For each of four word slots:

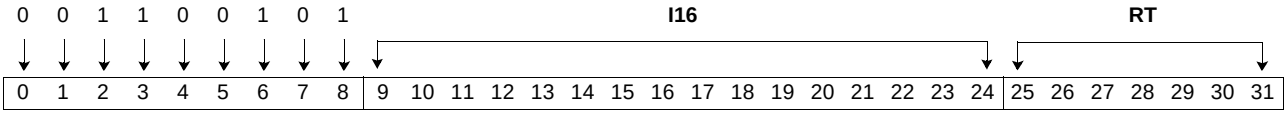
- The value in the I16 field is prefaced with zeros and ORed with the value in register RT.
- The result is placed into register RT.

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 — Required — v 1.0

fsmbi **rt,symbol**



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.

```
s ← I16
for j = 0 to 15
        If s_j = 0 then r^j ← 0x00
        else            r^j ← 0xFF
end
RT ← r
```

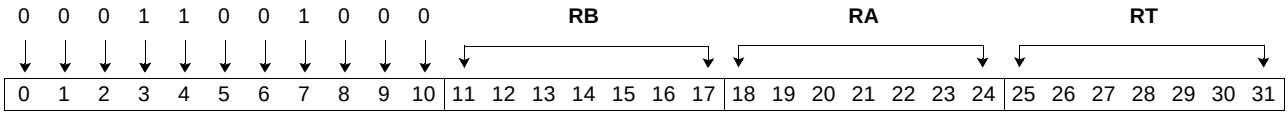
5. Integer and Logical Instructions

This section lists and describes the SPU integer and logical instructions.

Add Halfword

Required **v 1.0**

ah **rt,ra,rb**



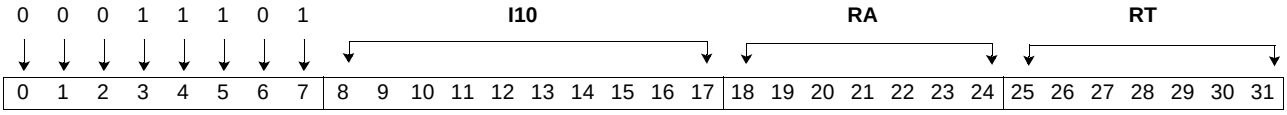
For each of eight halfword slots:

- 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.

$RT^{0:1}$	$\leftarrow RA^{0:1} + RB^{0:1}$
$RT^{2:3}$	$\leftarrow RA^{2:3} + RB^{2:3}$
$RT^{4:5}$	$\leftarrow RA^{4:5} + RB^{4:5}$
$RT^{6:7}$	$\leftarrow RA^{6:7} + RB^{6:7}$
$RT^{8:9}$	$\leftarrow RA^{8:9} + RB^{8:9}$
$RT^{10:11}$	$\leftarrow RA^{10:11} + RB^{10:11}$
$RT^{12:13}$	$\leftarrow RA^{12:13} + RB^{12:13}$
$RT^{14:15}$	$\leftarrow RA^{14:15} + RB^{14:15}$

Add Halfword Immediate — Required — v 1.0

ahi **rt,ra,value**



For each of eight halfword slots:

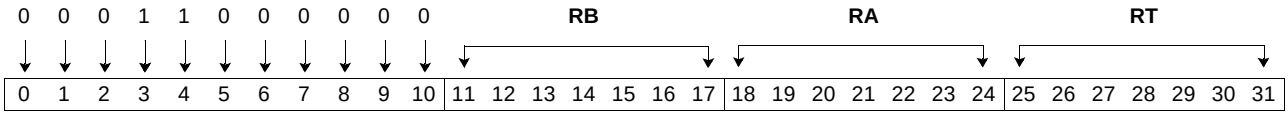
- The signed value in the I10 field is added to the value in register RA.
- The 16-bit result is placed in RT.
- Overflows and carries are not detected.

s	← RepLeftBit(I10,16)
$RT^{0:1}$	← $RA^{0:1}$ + s
$RT^{2:3}$	← $RA^{2:3}$ + s
$RT^{4:5}$	← $RA^{4:5}$ + s
$RT^{6:7}$	← $RA^{6:7}$ + s
$RT^{8:9}$	← $RA^{8:9}$ + s
$RT^{10:11}$	← $RA^{10:11}$ + s
$RT^{12:13}$	← $RA^{12:13}$+ s
$RT^{14:15}$	← $RA^{14:15}$ + s

Add Word

Required **v 1.0**

a **rt,ra,rb**



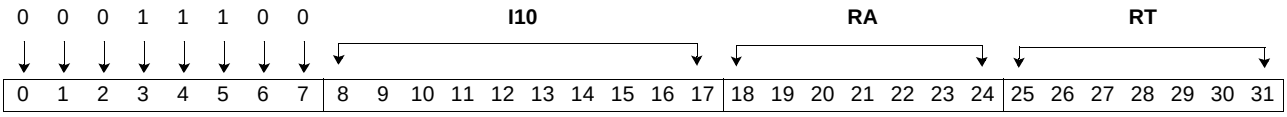
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{0:3} + RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7} + RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11} + RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15} + RB^{12:15}$

Add Word Immediate — Required — v 1.0

ai **rt,ra,value**



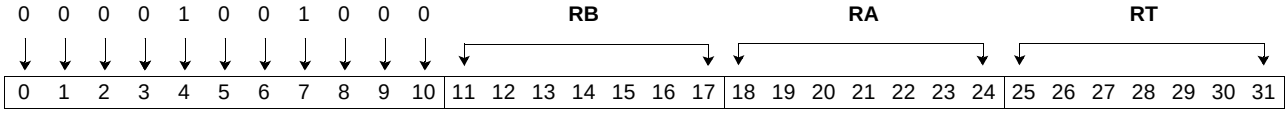
For each of four word slots:

- 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)
$RT^{0:3}$	← $RA^{0:3}$ + t
$RT^{4:7}$	← $RA^{4:7}$ + t
$RT^{8:11}$	← $RA^{8:11}$ + t
$RT^{12:15}$	← $RA^{12:15}$ + t

Subtract from Halfword — Required — v 1.0

sfh **rt,ra,rb**



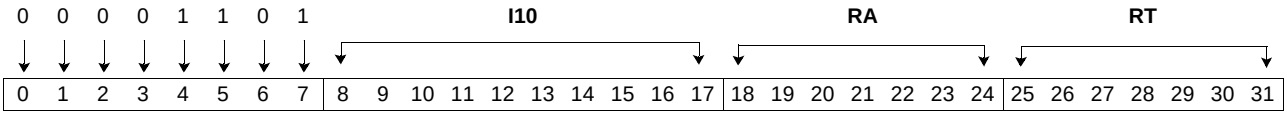
For each of eight halfword slots:

- 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.

$RT^{0:1}$	$\leftarrow RB^{0:1} + (\neg RA^{0:1}) + 1$
$RT^{2:3}$	$\leftarrow RB^{2:3} + (\neg RA^{2:3}) + 1$
$RT^{4:5}$	$\leftarrow RB^{4:5} + (\neg RA^{4:5}) + 1$
$RT^{6:7}$	$\leftarrow RB^{6:7} + (\neg RA^{6:7}) + 1$
$RT^{8:9}$	$\leftarrow RB^{8:9} + (\neg RA^{8:9}) + 1$
$RT^{10:11}$	$\leftarrow RB^{10:11} + (\neg RA^{10:11}) + 1$
$RT^{12:13}$	$\leftarrow RB^{12:13} + (\neg RA^{12:13}) + 1$
$RT^{14:15}$	$\leftarrow RB^{14:15} + (\neg RA^{14:15}) + 1$

Subtract from Halfword Immediate — Required — v 1.0

sfhi **rt,ra,value**



For each of eight halfword slots:

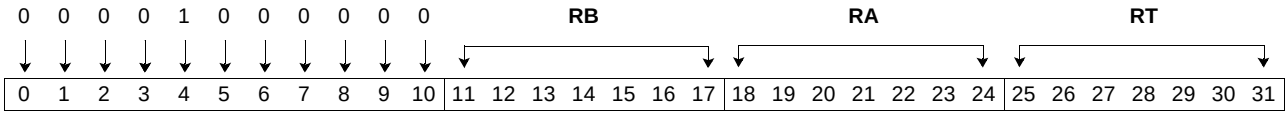
- The value in register RA is subtracted from the signed value in the I10 field.
- The 16-bit result is placed in register RT.
- Overflows are not detected.

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)
$RT^{0:1}$	← t + ($\neg RA^{0:1}$) + 1
$RT^{2:3}$	← t + ($\neg RA^{2:3}$) + 1
$RT^{4:5}$	← t + ($\neg RA^{4:5}$) + 1
$RT^{6:7}$	← t + ($\neg RA^{6:7}$) + 1
$RT^{8:9}$	← t + ($\neg RA^{8:9}$) + 1
$RT^{10:11}$	← t + ($\neg RA^{10:11}$) + 1
$RT^{12:13}$	← t + ($\neg RA^{12:13}$) + 1
$RT^{14:15}$	← t + ($\neg RA^{14:15}$) + 1

Subtract from Word — Required — v 1.0

sf **rt,ra,rb**



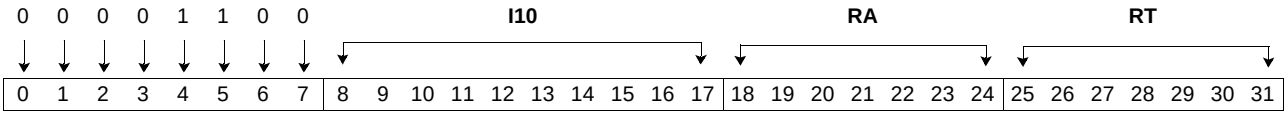
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RB^{0:3} + (\neg RA^{0:3}) + 1$
$RT^{4:7}$	$\leftarrow RB^{4:7} + (\neg RA^{4:7}) + 1$
$RT^{8:11}$	$\leftarrow RB^{8:11} + (\neg RA^{8:11}) + 1$
$RT^{12:15}$	$\leftarrow RB^{12:15} + (\neg RA^{12:15}) + 1$

Subtract from Word Immediate — Required — v 1.0

sfi **rt,ra,value**



For each of four word slots:

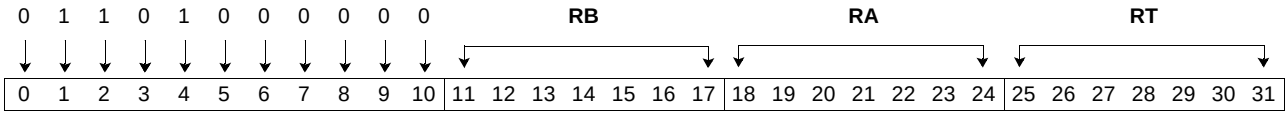
- The value in register RA is subtracted from the value in the I10 field.
- The result is placed in register RT.
- Overflows and carries are not detected.

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

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← t + ($\neg RA^{0:3}$) + 1
$RT^{4:7}$	← t + ($\neg RA^{4:7}$) + 1
$RT^{8:11}$	← t + ($\neg RA^{8:11}$) + 1
$RT^{12:15}$	← t + ($\neg RA^{12:15}$) + 1

Add Extended — Required — v 1.0

addx **rt,ra,rb**



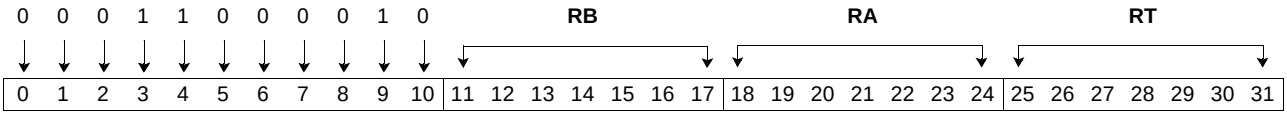
For each of four word slots:

- The operand from register RA is added to the operand from register RB and the least-significant bit of 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.

$RT^{0:3}$	$\leftarrow RA^{0:3} + RB^{0:3} + RT_{31}$
$RT^{4:7}$	$\leftarrow RA^{4:7} + RB^{4:7} + RT_{63}$
$RT^{8:11}$	$\leftarrow RA^{8:11} + RB^{8:11} + RT_{95}$
$RT^{12:15}$	$\leftarrow RA^{12:15} + RB^{12:15} + RT_{127}$

Carry Generate — Required — v 1.0

cg **rt,ra,rb**



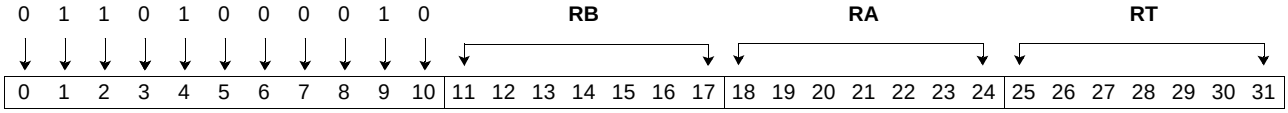
For each of four word slots:

- 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
        t0:32 = ((0 || RAj::4) + (0 || RBj::4))
        RTj::4 ← 310 || t0
end
```

Carry Generate Extended — Required — v 1.0

cgx **rt,ra,rb**



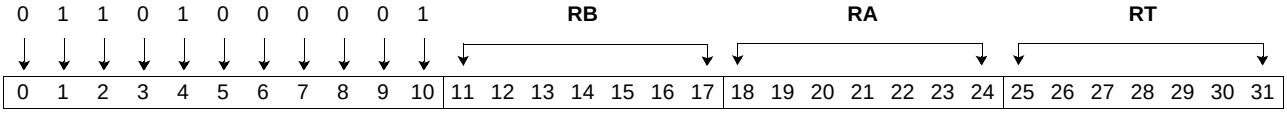
For each of four word slots:

- 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.

```
for j = 0 to 15 by 4
        t0:32 = (0 || RA^(j::4)) + (0 || RB^(j::4)) + (32 0 || RTj * 8 + 31)
        RT^(j::4) ← 31 0 || t0
end
```

Subtract from Extended — Required — v 1.0

sfx **rt,ra,rb**



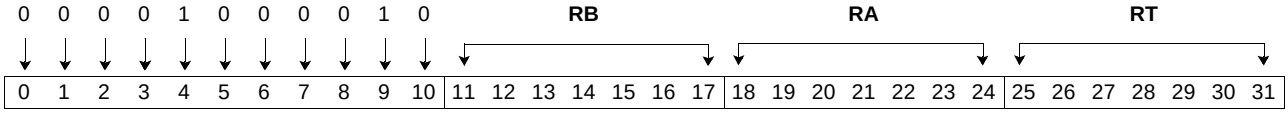
For each of four word slots:

- 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’.
- The 32-bit result is placed in register RT. Bits 0 to 30 of the RT input are reserved and should be zero.

$RT^{0:3}$	$\leftarrow RB^{0:3} + (\neg RA^{0:3}) + RT_{31}$
$RT^{4:7}$	$\leftarrow RB^{4:7} + (\neg RA^{4:7}) + RT_{63}$
$RT^{8:11}$	$\leftarrow RB^{8:11} + (\neg RA^{8:11}) + RT_{95}$
$RT^{12:15}$	$\leftarrow RB^{12:15} + (\neg RA^{12:15}) + RT_{127}$

Borrow Generate — Required — v 1.0

bg **rt,ra,rb**



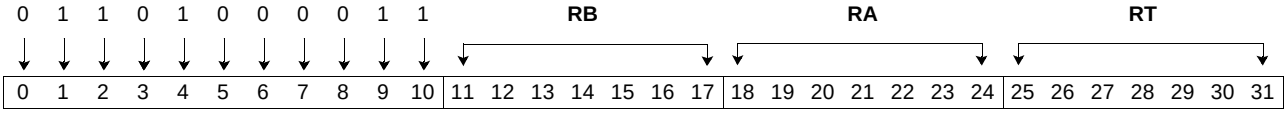
For each of four word slots:

- 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.

```
for j = 0 to 15 by 4
        if (RB^j::4 ≥^u RA^j::4) then   RT^j::4 ← 1
        else                            RT^j::4 ← 0
end
```

Borrow Generate Extended — Required — v 1.0

bgx **rt,ra,rb**



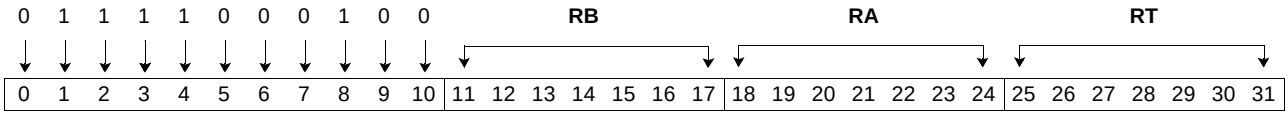
For each of four word slots:

- 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.

```
for j = 0 to 15 by 4
        if (RT_{j*8+31}) then
                if (RB^{j::4} ≥^u RA^{j::4}) then RT^{j::4} ← 1
                else                            RT^{j::4} ← 0
        else
                if (RB^{j::4} >^u RA^{j::4}) then RT^{j::4} ← 1
                else                            RT^{j::4} ← 0
end
```

Multiply Required v 1.0

mpy rt,ra,rb



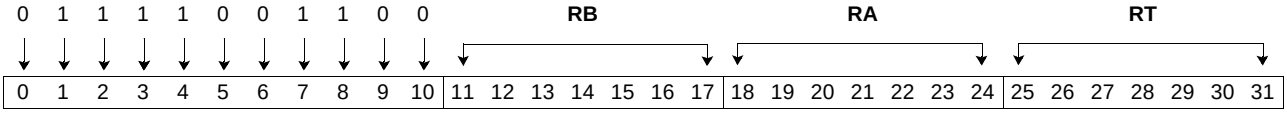
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{2:3} * RB^{2:3}$
$RT^{4:7}$	$\leftarrow RA^{6:7} * RB^{6:7}$
$RT^{8:11}$	$\leftarrow RA^{10:11} * RB^{10:11}$
$RT^{12:15}$	$\leftarrow RA^{14:15} * RB^{14:15}$

Multiply Unsigned — Required — v 1.0

mpyu **rt,ra,rb**



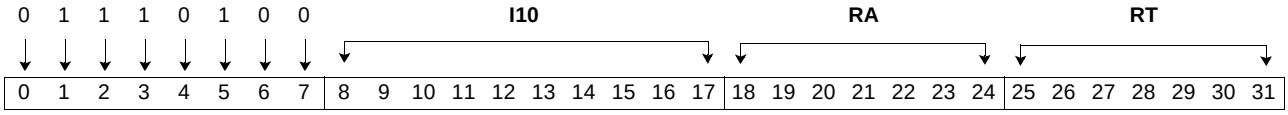
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{2:3}$ |*| $RB^{2:3}$
$RT^{4:7}$	$\leftarrow RA^{6:7}$ |*| $RB^{6:7}$
$RT^{8:11}$	$\leftarrow RA^{10:11}$ |*| $RB^{10:11}$
$RT^{12:15}$	$\leftarrow RA^{14:15}$ |*| $RB^{14:15}$

Multiply Immediate — Required — v 1.0

mpyi **rt,ra,value**



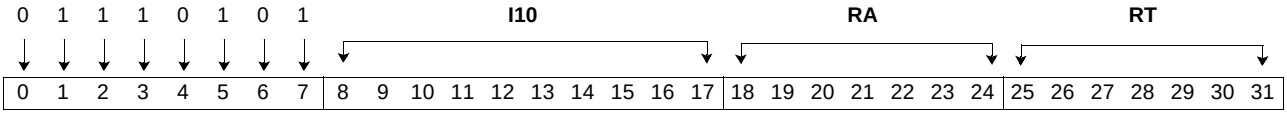
For each of four word slots:

- 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.

t	← RepLeftBit(I10,16)
$RT^{0:3}$	← $RA^{2:3}$ * t
$RT^{4:7}$	← $RA^{6:7}$ * t
$RT^{8:11}$	← $RA^{10:11}$ * t
$RT^{12:15}$	← $RA^{14:15}$ * t

Multiply Unsigned Immediate — Required — v 1.0

mpyui **rt,ra,value**



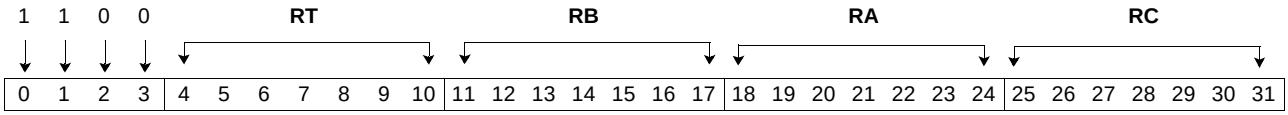
For each of four word slots:

- 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.

t	← RepLeftBit(I10,16)
$RT^{0:3}$	← $RA^{2:3}$ |*| t
$RT^{4:7}$	← $RA^{6:7}$ |*| t
$RT^{8:11}$	← $RA^{10:11}$ |*| t
$RT^{12:15}$	← $RA^{14:15}$ |*| t

Multiply and Add — Required — v 1.0

mpya **rt,ra,rb,rc**



For each of four word slots:

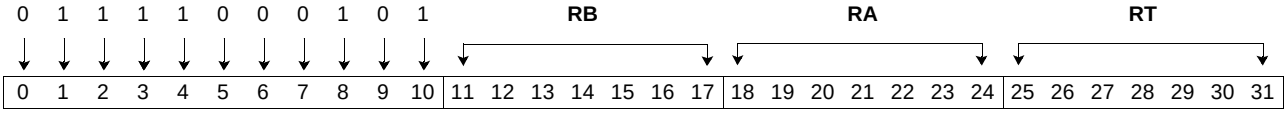
- The value in register RA is treated as a 16-bit signed integer and multiplied by the 16-bit signed value in register RB. The resulting product is added to the value in register RC.
- The result is placed in register RT.
- Overflows and carries are not detected.

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

t0	$\leftarrow RA^{2:3} * RB^{2:3}$
t1	$\leftarrow RA^{6:7} * RB^{6:7}$
t2	$\leftarrow RA^{10:11} * RB^{10:11}$
t3	$\leftarrow RA^{14:15} * RB^{14:15}$
$RT^{0:3}$	$\leftarrow t0 + RC^{0:3}$
$RT^{4:7}$	$\leftarrow t1 + RC^{4:7}$
$RT^{8:11}$	$\leftarrow t2 + RC^{8:11}$
$RT^{12:15}$	$\leftarrow t3 + RC^{12:15}$

Multiply High — Required — v 1.0

mpyh **rt,ra,rb**



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.

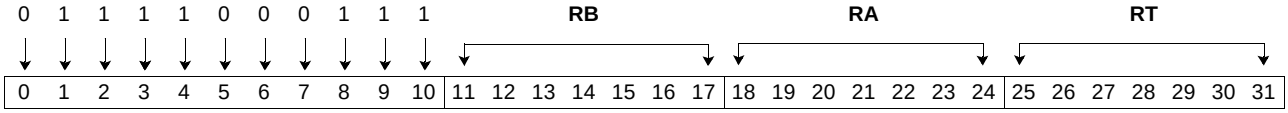
t0	$\leftarrow RA^{0:1} * RB^{2:3}$
t1	$\leftarrow RA^{4:5} * RB^{6:7}$
t2	$\leftarrow RA^{8:9} * RB^{10:11}$
t3	$\leftarrow RA^{12:13} * RB^{14:15}$
$RT^{0:3}$	$\leftarrow t0^{2:3}$ || 0x0000
$RT^{4:7}$	$\leftarrow t1^{2:3}$ || 0x0000
$RT^{8:11}$	$\leftarrow t2^{2:3}$ || 0x0000
$RT^{12:15}$	$\leftarrow t3^{2:3}$ || 0x0000

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**



For each of four word slots:

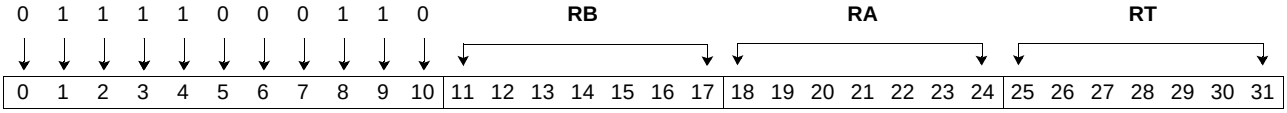
- The value in the rightmost 16 bits of register RA is multiplied by the value in the rightmost 16 bits of register RB.
- The leftmost 16 bits of the 32-bit product are placed in the rightmost 16 bits of register RT, with the sign bit replicated into the left 16 bits of the register.

t0	$\leftarrow RA^{2:3} * RB^{2:3}$
t1	$\leftarrow RA^{6:7} * RB^{6:7}$
t2	$\leftarrow RA^{10:11} * RB^{10:11}$
t3	$\leftarrow RA^{14:15} * RB^{14:15}$
$RT^{0:3}$	$\leftarrow$ RepLeftBit($t0^{0:1}$,32)
$RT^{4:7}$	$\leftarrow$ RepLeftBit($t1^{0:1}$,32)
$RT^{8:11}$	$\leftarrow$ RepLeftBit($t2^{0:1}$,32)
$RT^{12:15}$	$\leftarrow$ RepLeftBit($t3^{0:1}$,32)

Multiply High High

Required **v 1.0**

mpyhh **rt,ra,rb**



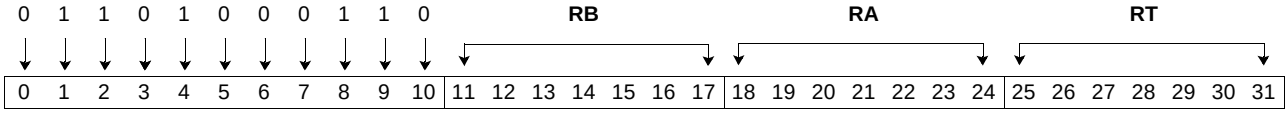
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{0:1} * RB^{0:1}$
$RT^{4:7}$	$\leftarrow RA^{4:5} * RB^{4:5}$
$RT^{8:11}$	$\leftarrow RA^{8:9} * RB^{8:9}$
$RT^{12:15}$	$\leftarrow RA^{12:13} * RB^{12:13}$

Multiply High High and Add — Required — v 1.0

mpyhha **rt,ra,rb**



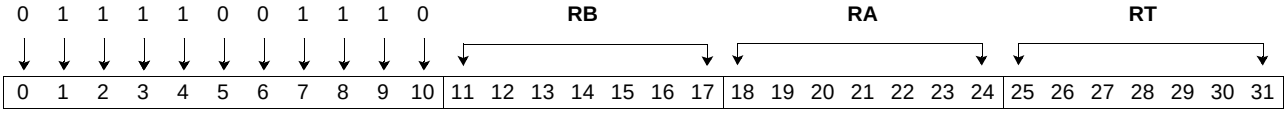
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{0:1} * RB^{0:1} + RT^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:5} * RB^{4:5} + RT^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:9} * RB^{8:9} + RT^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:13} * RB^{12:13} + RT^{12:15}$

Multiply High High Unsigned — Required — v 1.0

mpyhhu **rt,ra,rb**



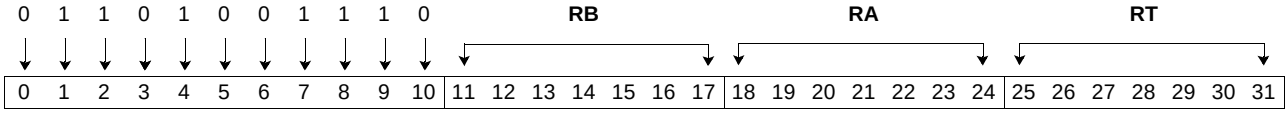
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{0:1} \mid * \mid RB^{0:1}$
$RT^{4:7}$	$\leftarrow RA^{4:5} \mid * \mid RB^{4:5}$
$RT^{8:11}$	$\leftarrow RA^{8:9} \mid * \mid RB^{8:9}$
$RT^{12:15}$	$\leftarrow RA^{12:13} \mid * \mid RB^{12:13}$

Multiply High High Unsigned and Add — Required — v 1.0

mpyhhau **rt,ra,rb**



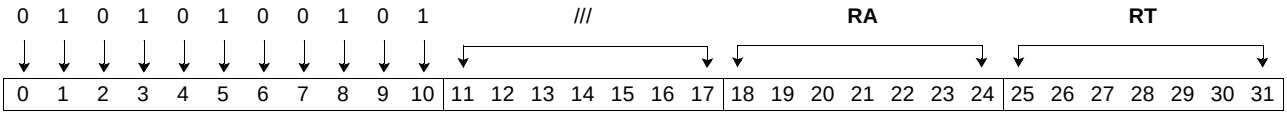
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{0:1} \mid * \mid RB^{0:1} + RT^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:5} \mid * \mid RB^{4:5} + RT^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:9} \mid * \mid RB^{8:9} + RT^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:13} \mid * \mid RB^{12:13} + RT^{12:15}$

Count Leading Zeros — Required — v 1.0

clz **rt,ra**



For each of four word slots:

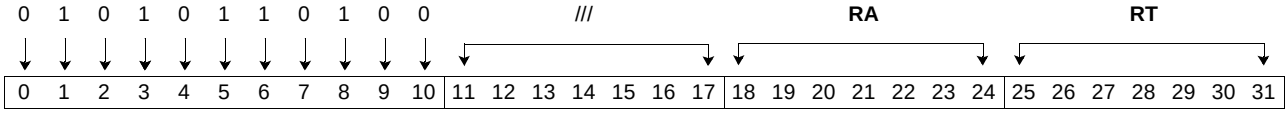
- The number of zero bits to the left of the first ‘1’ bit in the operand in register RA is computed.
- The result is placed in register RT. If register RA is zero, the result is 32.

Programming Note: The result placed in register RT satisfies $0 \le RT \le 32$. 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 — Required — v 1.0

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.

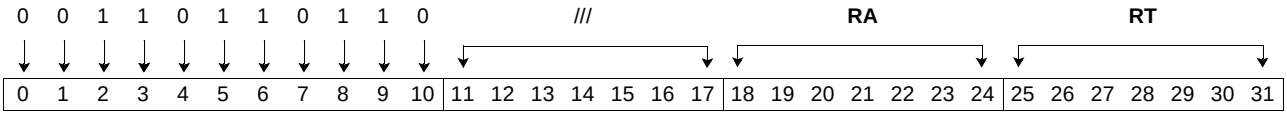
Programming Note: The result placed in register RT satisfies $0 \le RT \le 8$. The value in register RT is zero, for example, if the value in RA is zero. The value in RT is 8 if the value in RA is -1.

```
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.)

Form Select Mask for Bytes — Required — v 1.0

fsmb **rt,ra**

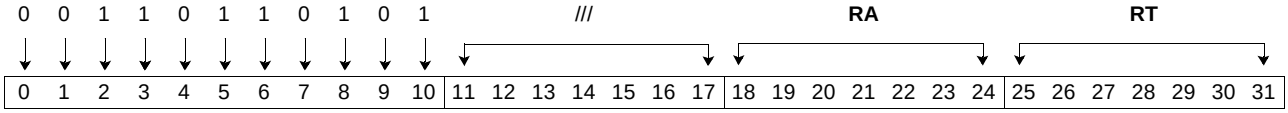


The rightmost 16 bits of the preferred slot of register RA are used to create a mask in register RT by replicating each bit eight times. Bits in the operand are related to bytes in the result in a left-to-right correspondence.

```
s ← RA^2:3
for j = 0 to 15
        If s_j = 0 then     r^j ← 0x00
        else                r^j ← 0xFF
end
RT ← r
```

Form Select Mask for Halfwords — Required — v 1.0

fsmh **rt,ra**

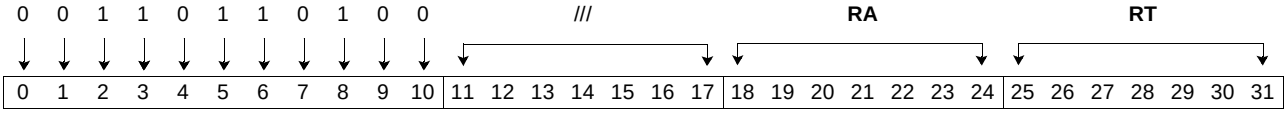


The rightmost 8 bits of the preferred slot of register RA are used to create a mask in register RT by replicating each bit 16 times. Bits in the operand are related to halfwords in the result, in a left-to-right correspondence.

```
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 — Required — v 1.0

fsm **rt,ra**

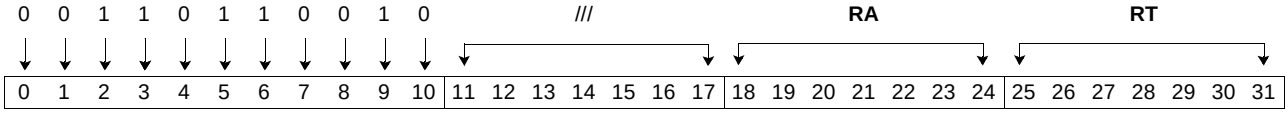


The rightmost 4 bits of the preferred slot of register RA are used to create a mask in register RT by replicating each bit 32 times. Bits in the operand are related to words in the result in a left-to-right correspondence.

```
s ← RA28:31
k = 0
for j = 0 to 3
        If sj = 0 then      rk::4 ← 0x00000000
        else                rk::4 ← 0xFFFFFFFF
        k = k + 4
end
RT ← r
```

Gather Bits from Bytes — Required — v 1.0

gbb **rt,ra**

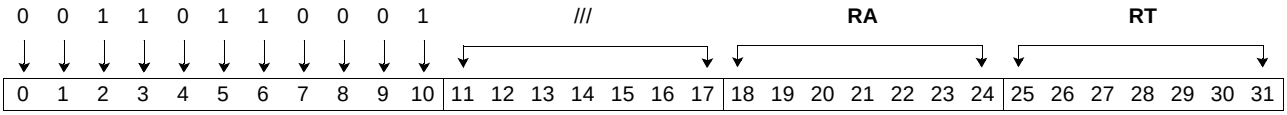


A 16-bit quantity is formed in the right half of the preferred slot of register RT by concatenating the rightmost bit in each byte of register RA. The leftmost 16 bits of register RT are set to zero, as are the remaining slots of register RT.

```
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 — Required — v 1.0

gbh **rt,ra**

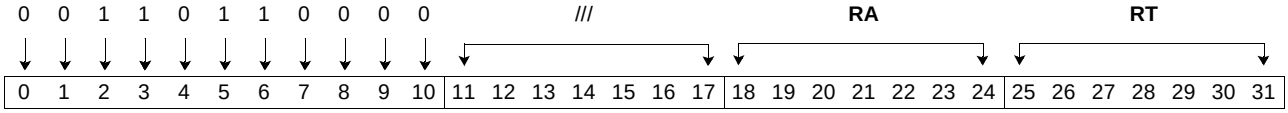


An 8-bit quantity is formed in the rightmost byte of the preferred slot of register RT by concatenating the rightmost bit in each halfword of register RA. The leftmost 24 bits of the preferred slot of register RT are set to zero, as are the remaining slots of register RT.

```
k ← 0
s ← 0x00
for j = 15 to 128 by 16
        s_k ←RA_j
        k ← k + 1
end
RT^0:3 ← 0x000000 || s
RT^4:7 ← 0
RT^8:11 ← 0
RT^12:15 ← 0
```

Gather Bits from Words — Required — v 1.0

gb **rt,ra**

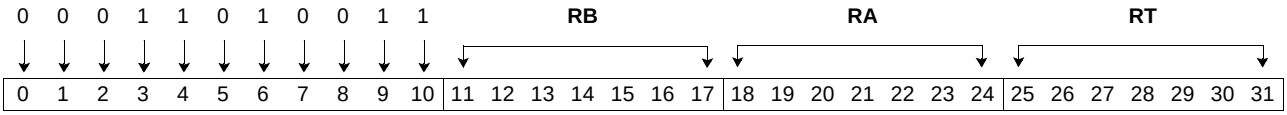


A 4-bit quantity is formed in the rightmost 4 bits of register RT by concatenating the rightmost bit in each word of register RA. The leftmost 28 bits of register RT are set to zero, as are the remaining slots of register RT.

```
k = 0
s = 0x0
for j = 31 to 128 by 32
        s_k ← RA_j
        k ← k +1
end
RT^0:3 ← 0x0000000 || s
RT^4:7 ← 0
RT^8:11 ← 0
RT^12:15 ← 0
```

Average Bytes — Required — v 1.0

avgb **rt,ra,rb**



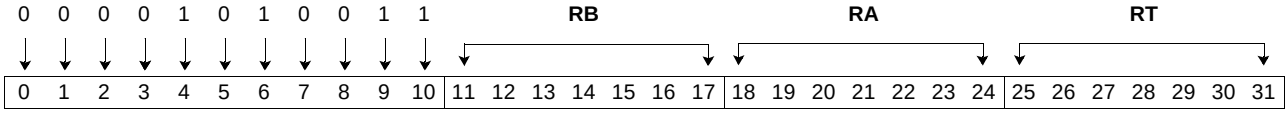
For each of 16 byte slots:

- 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.

```
for j = 0 to 15
        RT^j ← ((0x00 || RA^j) + (0x00 || RB^j) + 1)_7:14
end
```

Absolute Differences of Bytes — Required — v 1.0

absdb **rt,ra,rb**



For each of 16 byte slots:

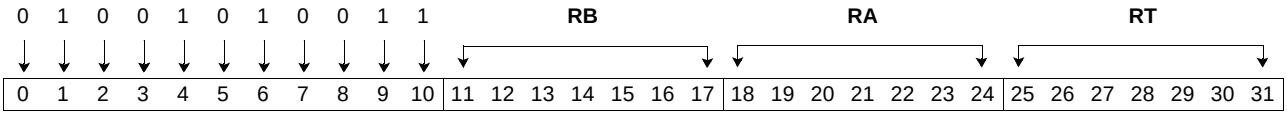
- The operand in register RA is subtracted from the operand in register RB.
- The absolute value of the result is placed in register RT.

Programming Note: The operands are unsigned.

```
for j = 0 to 15
        if (RB^j >^u RA^j) then     RT^j ← RB^j - RA^j
        else                        RT^j ← RA^j - RB^j
end
```

Sum Bytes into Halfwords — Required — v 1.0

sumb **rt,ra,rb**



For each of four word slots:

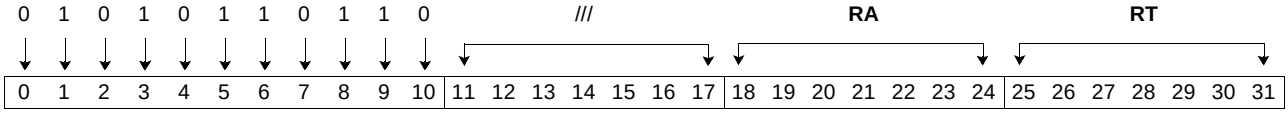
- The 4 bytes in register RB are added, and the 16-bit result is placed in bytes 0 and 1 of register RT.
- The 4 bytes in register RA are added, and the 16-bit result is placed in bytes 2 and 3 of register RT.

Programming Note: The operands are unsigned.

$RT^{0:1}$	$\leftarrow RB^0 + RB^1 + RB^2 + RB^3$
$RT^{2:3}$	$\leftarrow RA^0 + RA^1 + RA^2 + RA^3$
$RT^{4:5}$	$\leftarrow RB^4 + RB^5 + RB^6 + RB^7$
$RT^{6:7}$	$\leftarrow RA^4 + RA^5 + RA^6 + RA^7$
$RT^{8:9}$	$\leftarrow RB^8 + RB^9 + RB^{10} + RB^{11}$
$RT^{10:11}$	$\leftarrow RA^8 + RA^9 + RA^{10} + RA^{11}$
$RT^{12:13}$	$\leftarrow RB^{12} + RB^{13} + RB^{14} + RB^{15}$
$RT^{14:15}$	$\leftarrow RA^{12} + RA^{13} + RA^{14} + RA^{15}$

Extend Sign Byte to Halfword — Required — v 1.0

xsbh **rt,ra**



For each of eight halfword slots:

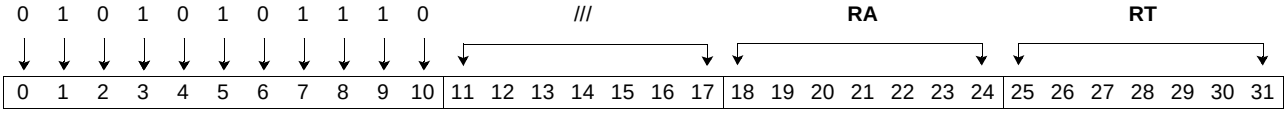
- The sign of the byte in the right byte of the operand in register RA is propagated to the left byte.
- The resulting 16-bit integer is stored in register RT.

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

$RT^{0:1}$	← RepLeftBit(RA^{1},16)
$RT^{2:3}$	← RepLeftBit(RA^{3},16)
$RT^{4:5}$	← RepLeftBit(RA^{5},16)
$RT^{6:7}$	← RepLeftBit(RA^{7},16)
$RT^{8:9}$	← RepLeftBit(RA^{9},16)
$RT^{10:11}$	← RepLeftBit(RA^{11},16)
$RT^{12:13}$	← RepLeftBit(RA^{13},16)
$RT^{14:15}$	← RepLeftBit(RA^{15},16)

Extend Sign Halfword to Word — Required — v 1.0

xshw **rt,ra**



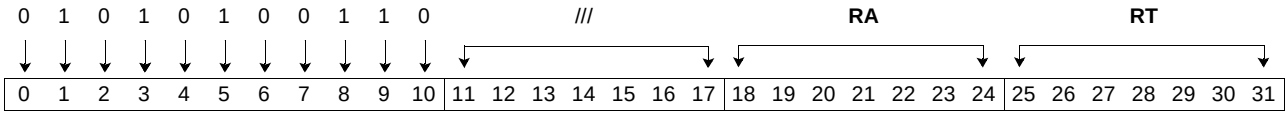
For each of four word slots:

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

$RT^{0:3}$	$\leftarrow$ RepLeftBit($RA^{2:3}$,32)
$RT^{4:7}$	$\leftarrow$ RepLeftBit($RA^{6:7}$,32)
$RT^{8:11}$	$\leftarrow$ RepLeftBit($RA^{10:11}$,32)
$RT^{12:15}$	$\leftarrow$ RepLeftBit($RA^{14:15}$,32)

Extend Sign Word to Doubleword — Required — v 1.0

xswd **rt,ra**



For each of two doubleword slots:

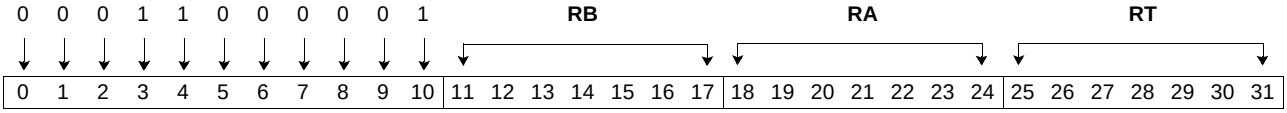
- The sign of the word in the right slot is propagated to the left word.
- The resulting 64-bit integer is stored in register RT.

$RT^{0:7}$	← RepLeftBit($RA^{4:7}$,64)
$RT^{8:15}$	← RepLeftBit($RA^{12:15}$,64)

And

Required v 1.0

and rt,ra,rb



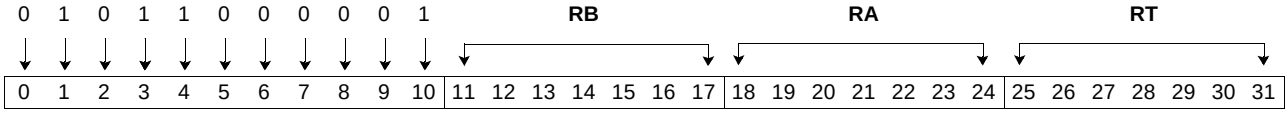
The values in register RA and register RB are logically ANDed. The result is placed in register RT.

$RT^{0:3}$	$\leftarrow RA^{0:3}$ & $RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7}$ & $RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11}$ & $RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15}$ & $RB^{12:15}$

And with Complement

Required **v 1.0**

andc **rt,ra,rb**

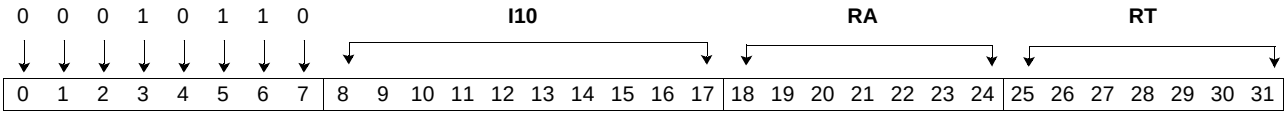


The value in register RA is logically ANDed with the complement of the value in register RB. The result is placed in register RT.

$RT^{0:3}$	$\leftarrow RA^{0:3}$ & ($\neg RB^{0:3}$)
$RT^{4:7}$	$\leftarrow RA^{4:7}$ & ($\neg RB^{4:7}$)
$RT^{8:11}$	$\leftarrow RA^{8:11}$ & ($\neg RB^{8:11}$)
$RT^{12:15}$	$\leftarrow RA^{12:15}$ & ($\neg RB^{12:15}$)

And Byte Immediate — Required — v 1.0

andbi **rt,ra,value**

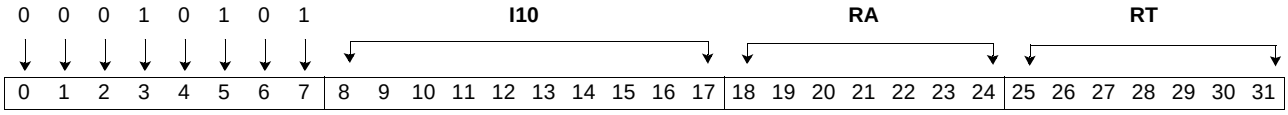


For each of 16 byte slots, the rightmost 8 bits of the I10 field are ANDed with the value in register RA. The result is placed in register RT.

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 — Required — v 1.0

andhi **rt,ra,value**



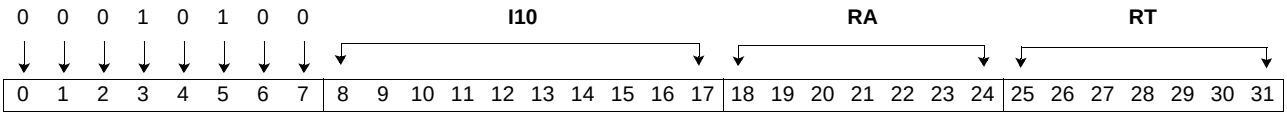
For each of eight halfword slots:

- 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.

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

And Word Immediate — Required — v 1.0

andi **rt,ra,value**



For each of four word slots:

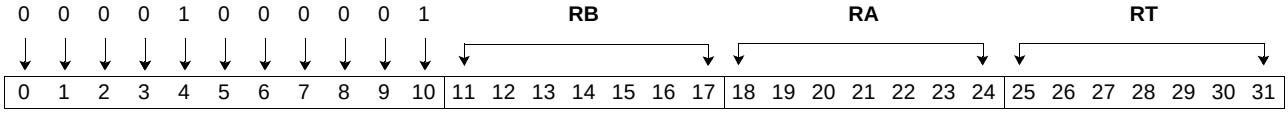
- The value of the I10 field is extended to 32 bits by replicating its leftmost bit. The result is ANDed with the contents of register RA.
- The result is placed in register RT.

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← $RA^{0:3}$ & t
$RT^{4:7}$	← $RA^{4:7}$ & t
$RT^{8:11}$	← $RA^{8:11}$ & t
$RT^{12:15}$	← $RA^{12:15}$ & t

Or

Required v 1.0

or rt,ra,rb

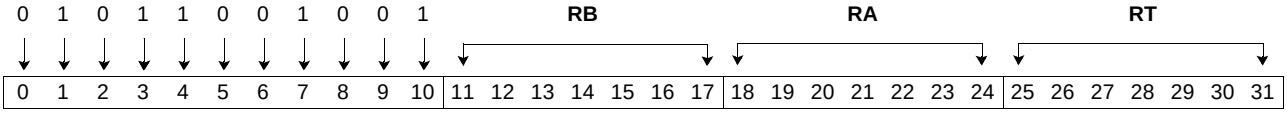


The values in register RA and register RB are logically ORed. The result is placed in register RT.

$RT^{0:3}$	$\leftarrow RA^{0:3} \mid RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7} \mid RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11} \mid RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15} \mid RB^{12:15}$

Or with Complement — Required — v 1.0

orc **rt,ra,rb**

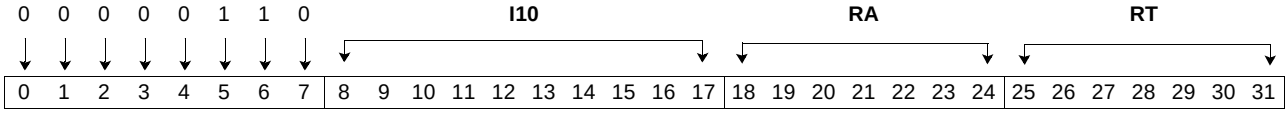


The value in register RA is ORed with the complement of the value in register RB. The result is placed in register RT.

$RT^{0:3}$	$\leftarrow RA^{0:3} \mid (\neg RB^{0:3})$
$RT^{4:7}$	$\leftarrow RA^{4:7} \mid (\neg RB^{4:7})$
$RT^{8:11}$	$\leftarrow RA^{8:11} \mid (\neg RB^{8:11})$
$RT^{12:15}$	$\leftarrow RA^{12:15} \mid (\neg RB^{12:15})$

Or Byte Immediate — Required — v 1.0

orbi **rt,ra,value**



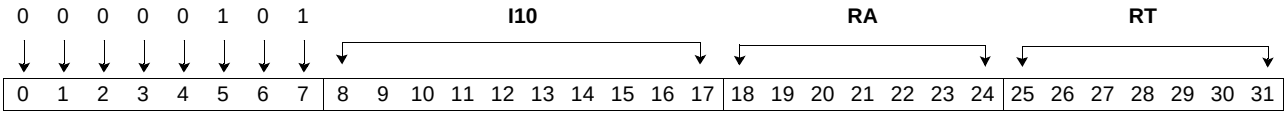
For each of 16 byte slots:

- The rightmost 8 bits of the I10 field are ORed with the value in register RA.
- The result is placed in register RT.

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

Or Halfword Immediate — Required — v 1.0

orhi **rt,ra,value**



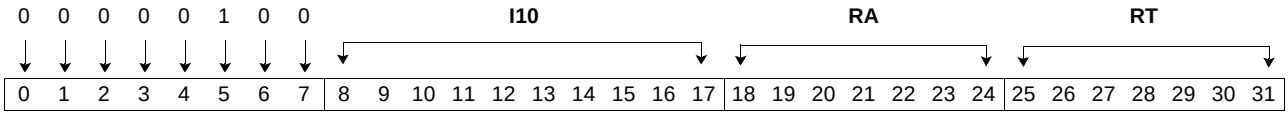
For each of eight halfword slots:

- 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

Or Word Immediate — Required — v 1.0

ori **rt,ra,value**



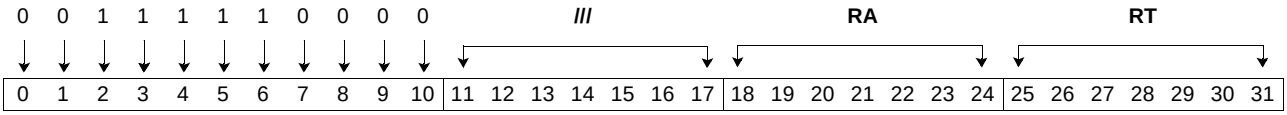
For each of four word slots:

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

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← $RA^{0:3}$ | t
$RT^{4:7}$	← $RA^{4:7}$ | t
$RT^{8:11}$	← $RA^{8:11}$ | t
$RT^{12:15}$	← $RA^{12:15}$ | t

Or Across — Required — v 1.0

orx **rt,ra**



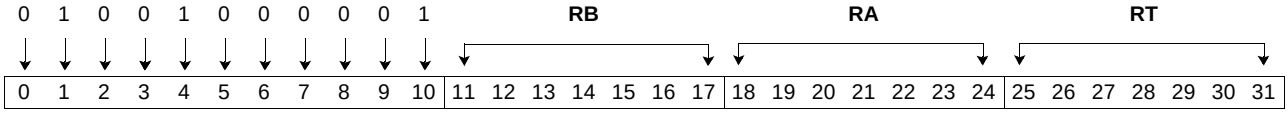
The four words of RA are logically ORed. The result is placed in the preferred slot of register RT. The other three slots of the register are written with zeros.

$RT^{0:3}$	$\leftarrow RA^{0:3} \mid RA^{4:7} \mid RA^{8:11} \mid RA^{12:15}$
$RT^{4:15}$	$\leftarrow 0$

Exclusive Or

Required **v 1.0**

xor **rt,ra,rb**

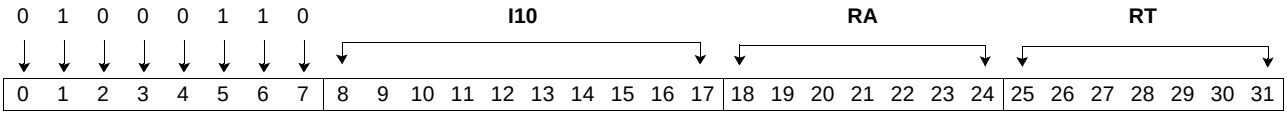


The values in register RA and register RB are logically XORed. The result is placed in register RT.

$RT^{0:3}$	$\leftarrow RA^{0:3} \oplus RB^{0:3}$
$RT^{4:7}$	$\leftarrow RA^{4:7} \oplus RB^{4:7}$
$RT^{8:11}$	$\leftarrow RA^{8:11} \oplus RB^{8:11}$
$RT^{12:15}$	$\leftarrow RA^{12:15} \oplus RB^{12:15}$

Exclusive Or Byte Immediate — Required — v 1.0

xorbi **rt,ra,value**



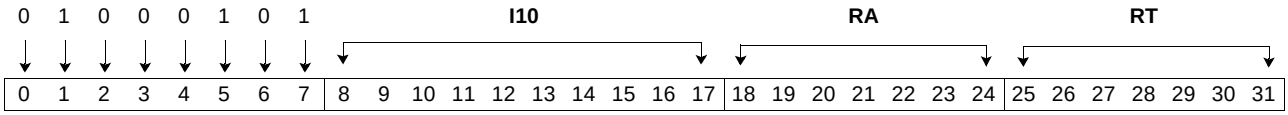
For each of 16 byte slots:

- The rightmost 8 bits of the I10 field are XORed 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

Exclusive Or Halfword Immediate — Required — v 1.0

xorhi **rt,ra,value**



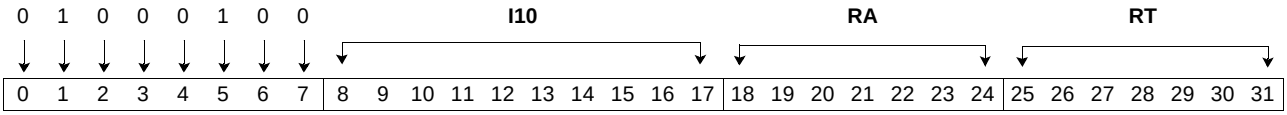
For each of eight halfword slots:

- 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)
$RT^{0:1}$	← $RA^{0:1} \oplus t$
$RT^{2:3}$	← $RA^{2:3} \oplus t$
$RT^{4:5}$	← $RA^{4:5} \oplus t$
$RT^{6:7}$	← $RA^{6:7} \oplus t$
$RT^{8:9}$	← $RA^{8:9} \oplus t$
$RT^{10:11}$	← $RA^{10:11} \oplus t$
$RT^{12:13}$	← $RA^{12:13} \oplus t$
$RT^{14:15}$	← $RA^{14:15} \oplus t$

Exclusive Or Word Immediate — Required — v 1.0

xori **rt,ra,value**



For each of four word slots:

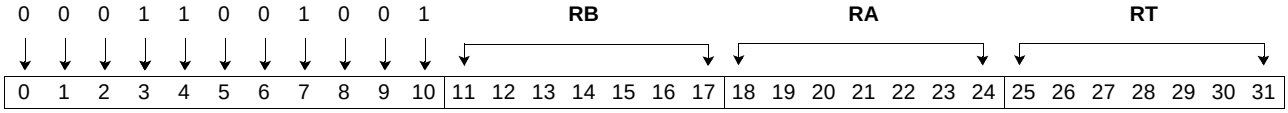
- The I10 field is sign-extended to 32 bits and XORed with the contents of register RA.
- The 32-bit result is placed in register RT.

t	← RepLeftBit(I10,32)
$RT^{0:3}$	← $RA^{0:3} \oplus t$
$RT^{4:7}$	← $RA^{4:7} \oplus t$
$RT^{8:11}$	← $RA^{8:11} \oplus t$
$RT^{12:15}$	← $RA^{12:15} \oplus t$

Nand

Required **v 1.0**

nand **rt,ra,rb**



For each of four word slots:

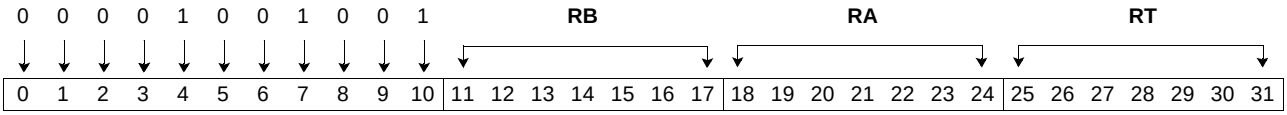
- The complement of the AND of the bit in register RA and the bit in register RB is placed in register RT.

$RT^{0:3}$	$\leftarrow \neg(RA^{0:3} \& RB^{0:3})$
$RT^{4:7}$	$\leftarrow \neg(RA^{4:7} \& RB^{4:7})$
$RT^{8:11}$	$\leftarrow \neg(RA^{8:11} \& RB^{8:11})$
$RT^{12:15}$	$\leftarrow \neg(RA^{12:15} \& RB^{12:15})$

Nor

Required v 1.0

nor **rt,ra,rb**



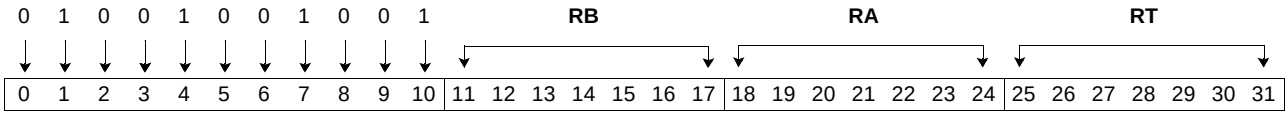
For each of four word slots:

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

$RT^{0:3}$	$\leftarrow \neg(RA^{0:3} \mid RB^{0:3})$
$RT^{4:7}$	$\leftarrow \neg(RA^{4:7} \mid RB^{4:7})$
$RT^{8:11}$	$\leftarrow \neg(RA^{8:11} \mid RB^{8:11})$
$RT^{12:15}$	$\leftarrow \neg(RA^{12:15} \mid RB^{12:15})$

Equivalent — Required v 1.0

eqv **rt,ra,rb**



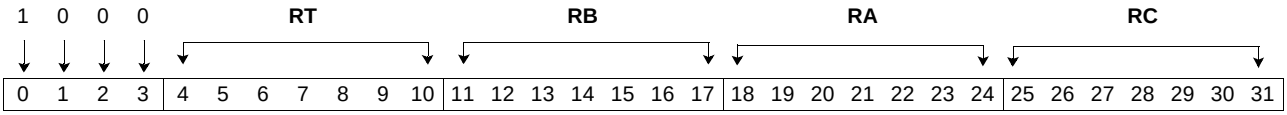
For each of four word slots:

- 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.

$RT^{0:3}$	$\leftarrow RA^{0:3} \oplus (\neg RB^{0:3})$
$RT^{4:7}$	$\leftarrow RA^{4:7} \oplus (\neg RB^{4:7})$
$RT^{8:11}$	$\leftarrow RA^{8:11} \oplus (\neg RB^{8:11})$
$RT^{12:15}$	$\leftarrow RA^{12:15} \oplus (\neg RB^{12:15})$

Select Bits — Required — v 1.0

selb **rt,ra,rb,rc**



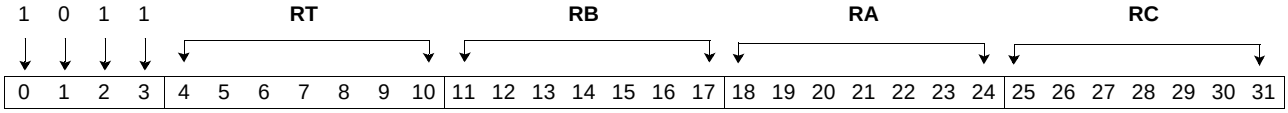
A result is formed by using bits from RC to choose corresponding bits either from RA or RB.

- 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.

$RT^{0:15} \leftarrow RC^{0:15}$ & $RB^{0:15}$ | ($\neg RC^{0:15}$) & $RA^{0:15}$

Shuffle Bytes — Required — v 1.0

shufb **rt,ra,rb,rc**



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

Value in Register RC (Expressed in Binary)	Result Byte
10xxxxxx	0x00
110xxxxx	0xFF
111xxxxx	0x80
Otherwise	The byte of the concatenated register addressed by the rightmost 5 bits of register RC

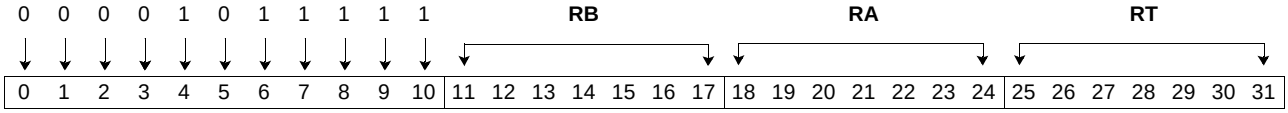
```
Rconcat ← RA || RB
for j = 0 to 15
        b ← RC^j
        If b_0:1 = 0b10 then c ← 0x00
        else If b_0:2 = 0b110 then c ← 0xFF
        else If b_0:2 = 0b111 then c ← 0x80
        else
                b ← b & 0x1F;
                c ← Rconcat^b;
        RT^j ← c
end
```

6. Shift and Rotate Instructions

This section describes the SPU shift and rotate instructions.

Shift Left Halfword — Required — v 1.0

shlh **rt,ra,rb**



For each of eight halfword slots:

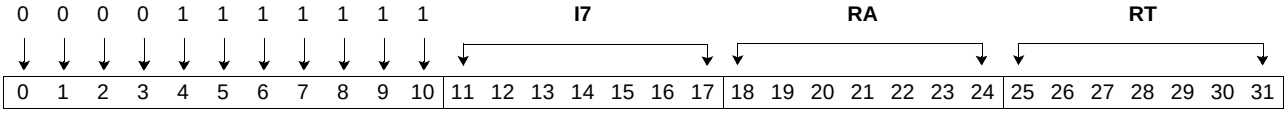
- The contents of register RA are shifted to the left according to the count in bits 11 to 15 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.
- Bits shifted 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 ← RB^(j::2) & 0x001F
        t ← RA^(j::2)
        for b = 0 to 15
                if b + s < 16 then      r_b ← t_(b + s)
                else                    r_b ← 0
        end
        RT^(j::2) ← r
end
```

Shift Left Halfword Immediate — Required — v 1.0

shlhi **rt,ra,value**



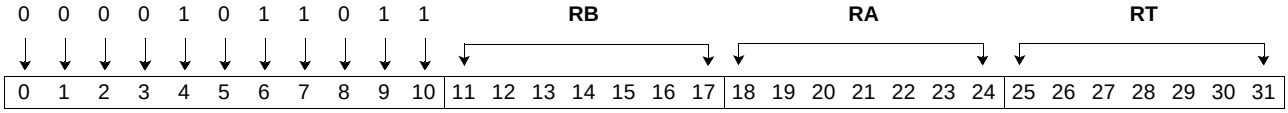
For each of eight halfword slots:

- 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^(j::2)
        for b = 0 to 15
                if b + s < 16 then      r_b ← t_(b + s)
                else                    r_b ← 0
        end
        RT^(j::2) ← r
end
```

Shift Left Word — Required — v 1.0

shl **rt,ra,rb**



For each of four word slots:

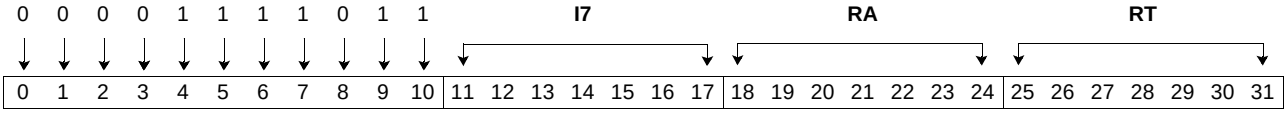
- The contents of register RA are shifted to the left according to the count in bits 26 to 31 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 31, the result is zero.
- Bits shifted 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 ← RB^(j::4) & 0x0000003F
        t ← RA^(j::4)
        for b = 0 to 31
                if b + s < 32 then      r_b ← t_(b + s)
                else                    r_b ← 0
        end
        RT^(j::4) ← r
end
```

Shift Left Word Immediate — Required — v 1.0

shli **rt,ra,value**



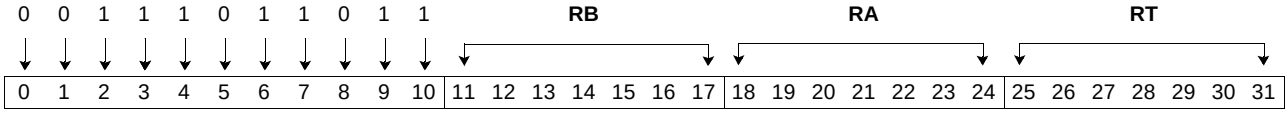
For each of four word slots:

- 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 ← RA^{j:4}
        for b = 0 to 31
                if b + s < 32 then      r_b ← t_{b + s}
                else                    r_b ← 0
        end
        RT^{j:4} ← r
end
```

Shift Left Quadword by Bits — Required — v 1.0

shlqbi **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.

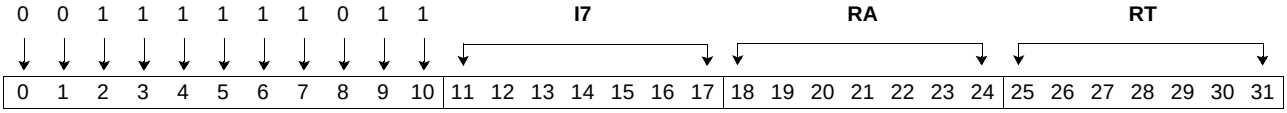
If the count is zero, the contents of register RA are copied unchanged into register RT.

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

```
s ← RB29:31
for b = 0 to 127
        if b + s < 128 then rb ← RAb + s
        else                rb ← 0
end
RT ← r
```

Shift Left Quadword by Bits Immediate — Required — v 1.0

shlqbii **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.

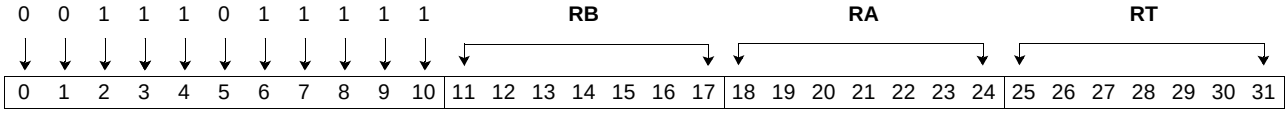
If the count is zero, the contents of register RA are copied unchanged into register RT.

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

```
s ← I7 & 0x07
for b = 0 to 127
        if b + s < 128 then r_b ← RA_(b + s)
        else                r_b ← 0
end
RT ← r
```

Shift Left Quadword by Bytes — Required — v 1.0

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.

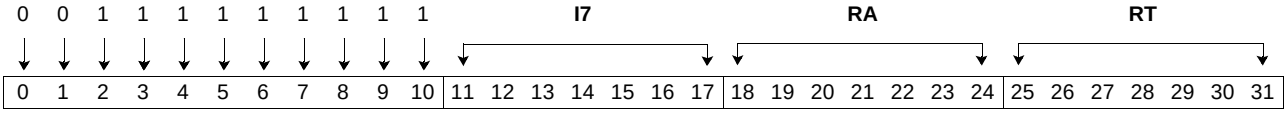
If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.

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

```
s ← RB27:31
for b = 0 to 15
        if b + s < 16 then rb ← RAb + s
        else               rb ← 0
end
RT ← r
```

Shift Left Quadword by Bytes Immediate — Required — v 1.0

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.

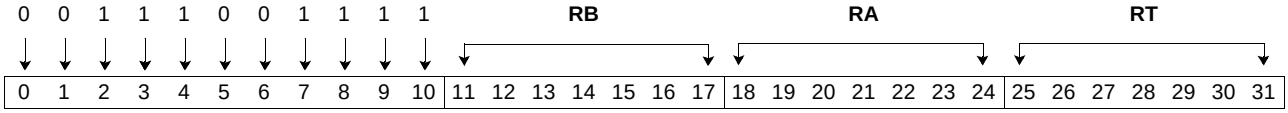
If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.

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

```
s ← I7 & 0x1F
for b = 0 to 15
        if b + s < 16 then r^b ← RA^(b + s)
        else               r^b ← 0
end
RT ← r
```

Shift Left Quadword by Bytes from Bit Shift Count Required v 1.0

shlqbybi rt,ra,rb



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.

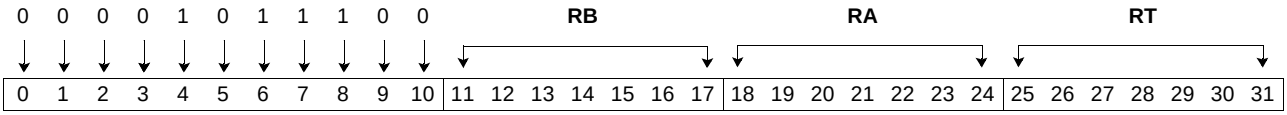
If the count is zero, the contents of register RA are copied unchanged into register RT. If the count is greater than 15, the result is zero.

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

```
s ← RB24:28
for b = 0 to 15
        if b + s < 16 then rb ← RAb + s
        else               rb ← 0x00
end
RT ← r
```

Rotate Halfword — Required — v 1.0

roth **rt,ra,rb**



For each of eight halfword slots:

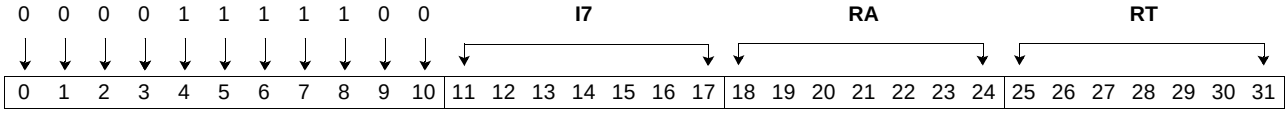
- The contents of register RA are rotated to the left according to the count in bits 12 to 15 of register RB.
- The result is placed in register RT.
- If the count is zero, the contents of register RA are copied unchanged into register RT.
- Bits rotated out of the left end of the halfword are rotated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

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

```
for j = 0 to 15 by 2
        s ← RB^(j::2) & 0x000F
        t ← RA^(j::2)
        for b = 0 to 15
                if b + s < 16 then      r_b ← t_(b + s)
                else                    r_b ← t_(b + s - 16)
        end
        RT^(j::2) ← r
end
```

Rotate Halfword Immediate — Required — v 1.0

rothi **rt,ra,value**



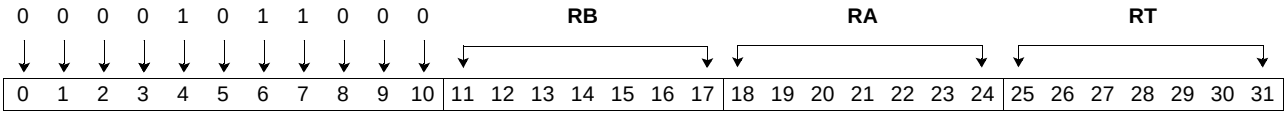
For each of eight halfword slots:

- 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 ← RepLeftBit(I7,16) & 0x000F
for j = 0 to 15 by 2
        t ← RA^(j::2)
        for b = 0 to 15
                if b + s < 16 then      r_b ← t_(b + s)
                else                    r_b ← t_(b + s - 16)
        end
        RT^(j::2) ← r
end
```

Rotate Word — Required — v 1.0

rot **rt,ra,rb**



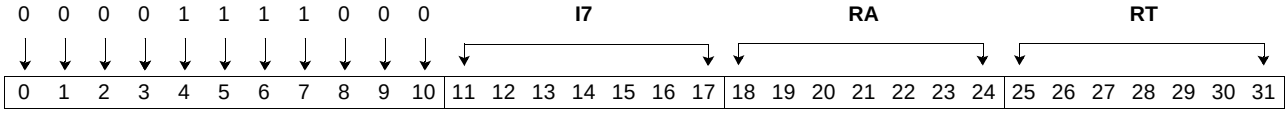
For each of four word slots:

- 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.

```
for j = 0 to 15 by 4
        s ← RB^(j::4) & 0x0000001F
        t ← RA^(j::4)
        for b = 0 to 31
                if b + s < 32 then      r_b ← t_(b + s)
                else                    r_b ← t_(b + s - 32)
        end
        RT^(j::4) ← r
end
```

Rotate Word Immediate — Required — v 1.0

roti **rt,ra,value**



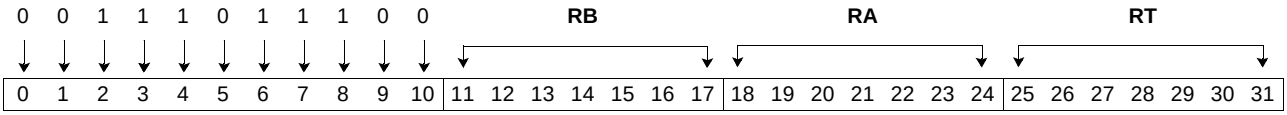
For each of four word slots:

- 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 ← RepLeftBit(I7,32) & 0x0000001F
for j = 0 to 15 by 4
        t ← RA^(j::4)
        for b = 0 to 31
                if b + s < 32 then      r_b ← t_(b + s)
                else                    r_b ← t_(b + s - 32)
        end
        RT^(j::4) ← r
end
```

Rotate Quadword by Bytes — Required — v 1.0

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.

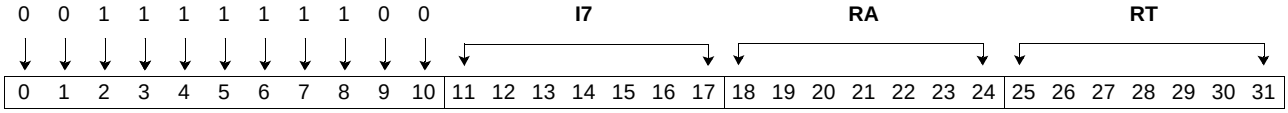
If the count is zero, the contents of register RA are copied unchanged into register RT.

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 Required v 1.0

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.

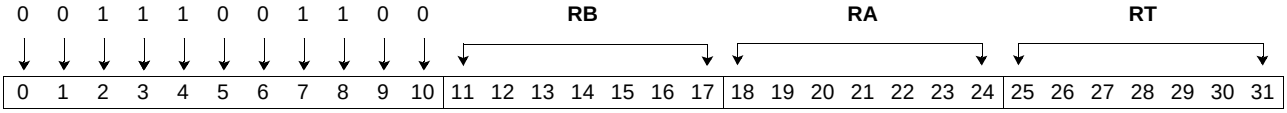
If the count is zero, the contents of register RA are copied unchanged into register RT.

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

```
s ← I7 14:17
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 Bytes from Bit Shift Count Required v 1.0

rotqbybi rt,ra,rb



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.

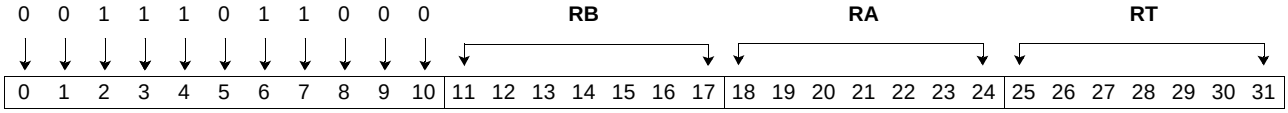
If the count is zero, the contents of register RA are copied unchanged into register RT.

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

```
s ← RB24:28
for b = 0 to 15
        if b + s < 16 then rb ← RAb + s
        else               rb ← RAb + s - 16
end
RT ← r
```

Rotate Quadword by Bits — Required — v 1.0

rotqbi **rt,ra,rb**



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.

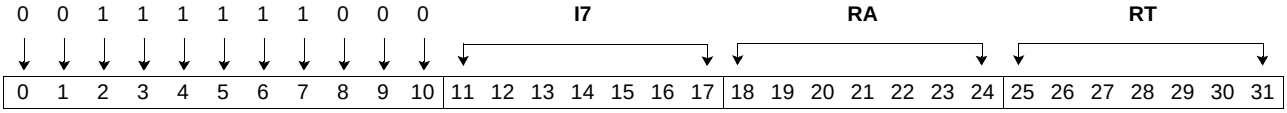
If the count is zero, the contents of register RA are copied unchanged into register RT.

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

```
s ← RB29:31
for b = 0 to 127
        if b + s < 128 then rb ← RAb + s
        else                rb ← RAb + s - 128
end
RT ← r
```

Rotate Quadword by Bits Immediate — Required — v 1.0

rotqbii **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.

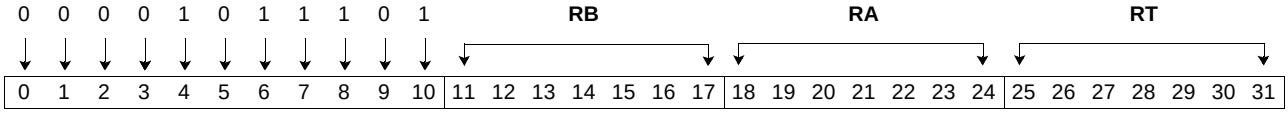
If the count is zero, the contents of register RA are copied unchanged into register RT.

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

```
s ← I4:6
for b = 0 to 127
        if b + s < 128 then rb ← RAb + s
        else                rb ← RAb + s - 128
end
RT ← r
```

Rotate and Mask Halfword — Required — v 1.0

rothm **rt,ra,rb**



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 - RB^(j::2)) & 0x001F
        t ← RA^(j::2)
        for b = 0 to 15
                if b ≥ s then    r_b ← t_(b - s)
                else             r_b ← 0
        end
        RT^(j::2) ← r
end
```

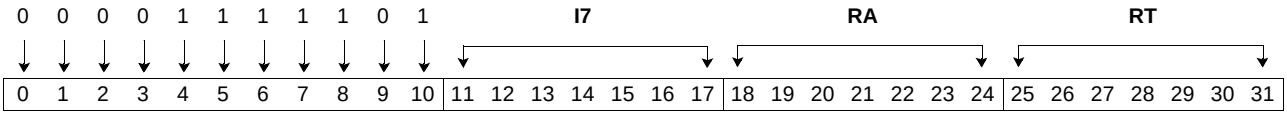
Programming Note: The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions 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 Halfword Immediate — Required — v 1.0

rothmi **rt,ra,value**



For each of eight halfword slots:

- 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 bits, with zero fill at the left.
- Otherwise, RT is set to zero.

```
s ← (0 - RepLeftBit(I7,32)) & 0x0000001F
for j = 0 to 15 by 2
        t ← RA^(j::2)
        for b = 0 to 15
                if b≥s then     r_b ← t_(b-s)
                else            r_b ← 0
        end
        RT^(j::2) ← r
end
```

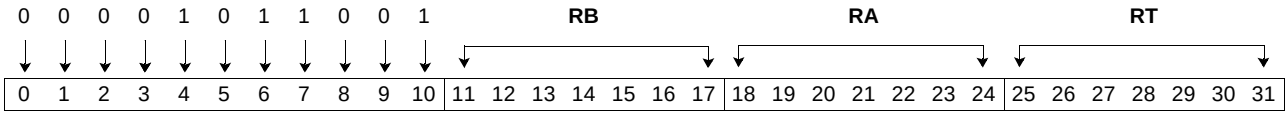
Programming Note: The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions 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 Word — Required — v 1.0

rotm **rt,ra,rb**



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, with zero fill at the left.
- Otherwise, RT is set to zero.

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

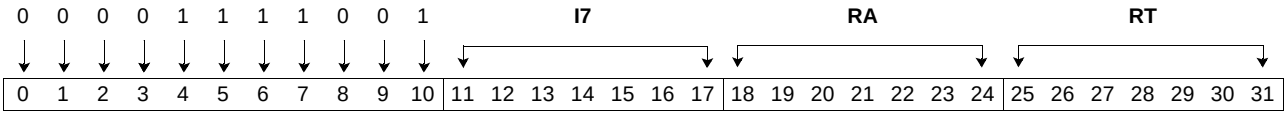
Programming Note: The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions 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 Word Immediate — Required — v 1.0

rotmi **rt,ra,value**



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 ← RA^(j::4)
        for b = 0 to 31
                if b ≥ s then   r_b ← t_(b - s)
                else            r_b ← 0
        end
        RT^(j::4) ← r
end
```

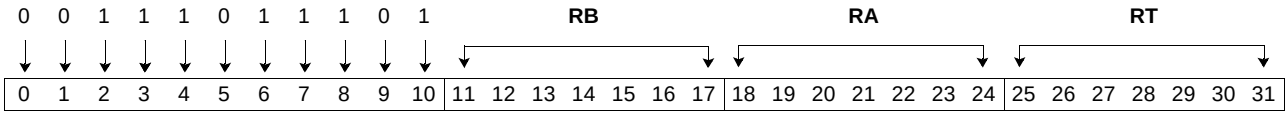
Programming Note: The Rotate and Mask instructions provide support for a logical right shift, and the Rotate and Mask Algebraic instructions provide support for an algebraic right shift. They differ from a conventional right logical or algebraic shift in that the shift amount accepted by the instructions 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 — Required — v 1.0

rotqmby **rt,ra,rb**

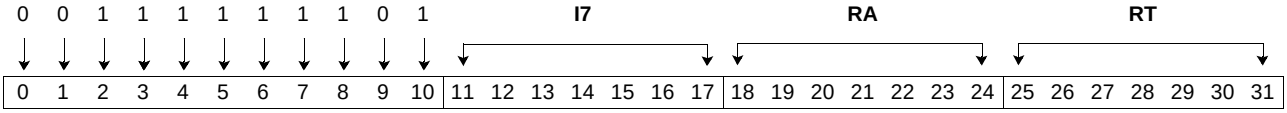


The shift_count is (0 - the preferred word of RB) modulo 32. If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bytes, filling at the left with 0x00 bytes. Otherwise, RT is set to zero.

```
s ← (0 - RB27:31) & 0x1F
for b = 0 to 15
        if b ≥ s then       rb ← tb - s
        else                rb ← 0x00
end
RT ← r
```

Rotate and Mask Quadword by Bytes Immediate Required v 1.0

rotqmbyi rt,ra,value

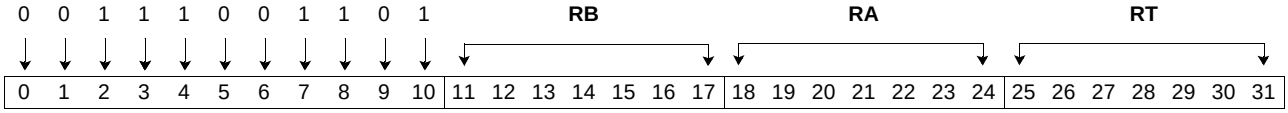


The shift_count is (0 - I7) modulo 32. If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bytes, filling at the left with 0x00 bytes. Otherwise, all bytes of RT are set to 0x00.

```
s ← (0 - I7) & 0x1F
for b = 0 to 15
        if b ≥ s then     r^b ← t^(b - s)
        else              r^b ← 0x00
end
RT ← r
```

Rotate and Mask Quadword Bytes from Bit Shift Count Required v 1.0

rotqmbybi rt,ra,rb

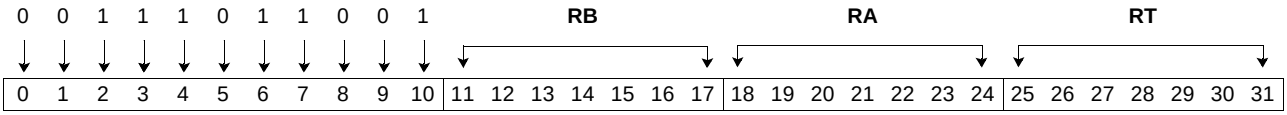


The shift_count is (0 minus bits 24 to 28 of RB) modulo 32. If the shift_count is less than 16, then RT is set to the contents of RA, which is shifted right shift_count bytes, and filled at the left with 0x00 bytes. Otherwise, all bytes of RT are set to 0x00.

```
s ← (0 - RB24:28) & 0x1F
for b = 0 to 15
        if b ≥ s then       rb ← RAb - s
        else                rb ← 0x00
end
```

Rotate and Mask Quadword by Bits — Required — v 1.0

rotqmbi **rt,ra,rb**

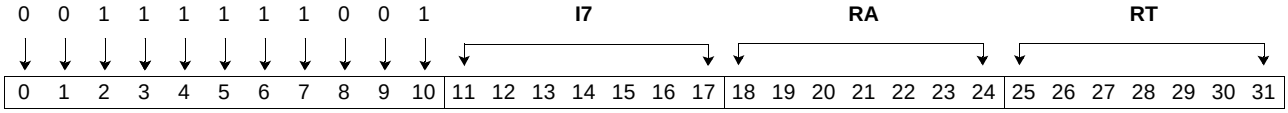


The shift_count is (0 - the preferred word of RB) modulo 8. RT is set to the contents of RA, shifted right by shift_count bits, filling at the left with zero bits.

```
s ← (0 - RB29:31) & 0x07
for b = 0 to 127
        if b ≥ s then     rb ← tb - s
        else              rb ← 0
end
RT ← r
```

Rotate and Mask Quadword by Bits Immediate Required v 1.0

rotqmbii rt,ra,value

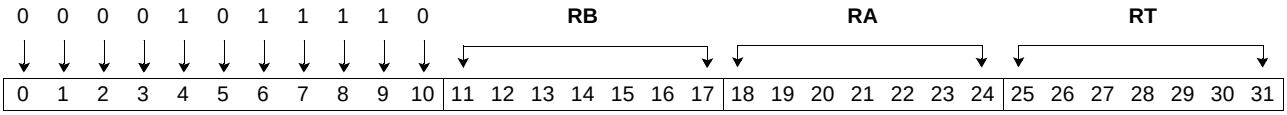


The shift_count is (0 - I7) modulo 8. RT is set to the contents of RA, shifted right by shift_count bits, filling at the left with zero bits.

```
s ← (0 - I7) & 0x07
for b = 0 to 127
        if b ≥ s then       r_b ← t_(b - s)
        else                r_b ← 0
end
RT ← r
```

Rotate and Mask Algebraic Halfword — Required — v 1.0

rotmah **rt,ra,rb**



For each of eight halfword slots:

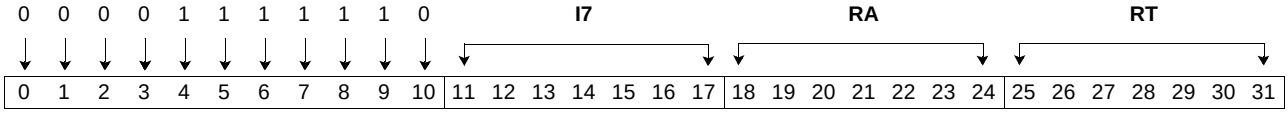
- The shift_count is (0 - RB) modulo 32.
- If the shift_count is less than 16, then RT is set to the contents of RA shifted right shift_count bits, replicating bit 0 (of the halfword) at the left.
- Otherwise, all bits of this halfword of RT are set to bit 0 of this halfword of RA.

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

```
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    r_b ← t_(b - s)
                else             r_b ← t_0
        end
        RT^(j::2) ← r
end
```

Rotate and Mask Algebraic Halfword Immediate Required v 1.0

rotmahi rt,ra,value



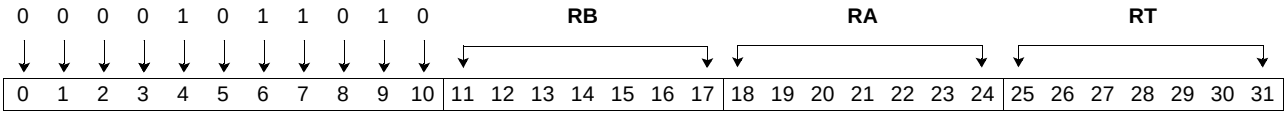
For each of eight halfword slots:

- The shift_count is (0 - I7) 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.

```
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    r_b ← t_(b - s)
                else             r_b ← t_0
        end
        RT^(j::2) ← r
end
```

Rotate and Mask Algebraic Word — Required — v 1.0

rotma **rt,ra,rb**



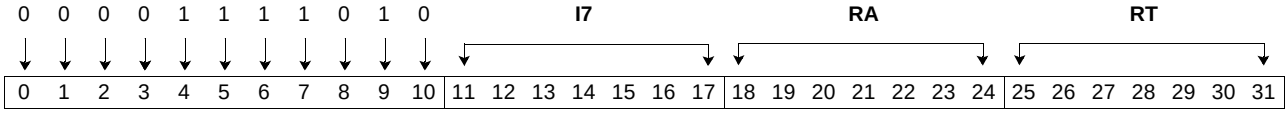
For each of four word slots:

- 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.

```
for j = 0 to 15 by 4
        s ← (0 - RB^(j:4)) & 0x0000003F
        t ← RA^(j:4)
        for b = 0 to 31
                if b ≥ s then    r_b ← t_(b - s)
                else             r_b ← t_0
        end
        RT^(j:4) ← r
end
```

Rotate and Mask Algebraic Word Immediate — Required — v 1.0

rotmai **rt,ra,value**



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 - RepLeftBit(I7,32)) & 0x0000003F
for j = 0 to 15 by 4
        t ← RA^(j::4)
        for b = 0 to 31
                if b ≥ 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.

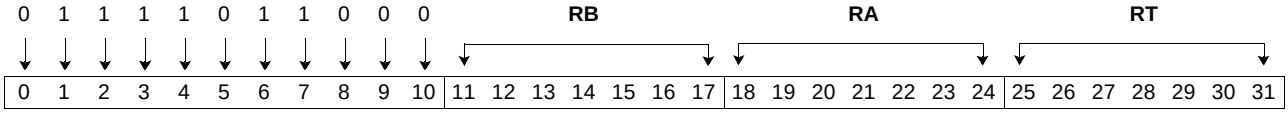
Logical comparison instructions treat the operands as unsigned integers. Other compare instructions treat the operands as two's complement signed integers.

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 — Required — v 1.0

heq **ra,rb**



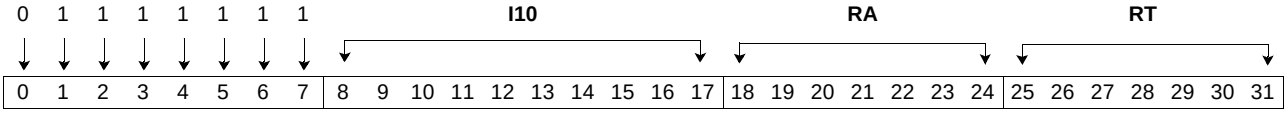
The value in the preferred slot of register RA is compared with the value in the preferred slot of register RB. If the values are equal, execution of the program stops at or after the halt.

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.

If $RA^{0:3} = RB^{0:3}$ then
    Stop after executing zero or more instructions after the halt.

Halt If Equal Immediate — Required — v 1.0

heqi **ra,symbol**



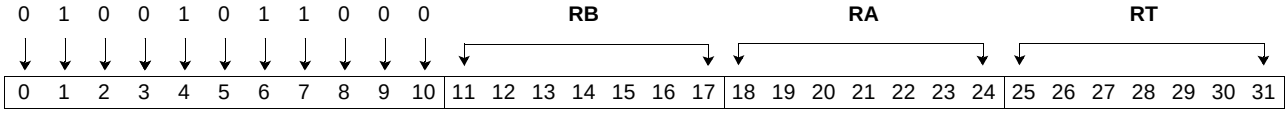
The value in the I10 field is extended to 32 bits by replicating the leftmost bit. The result is compared to the value in the preferred slot of register RA. If the value from register RA is equal to the immediate value, execution of the SPU program stops at or after the halt instruction.

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 $RA^{0:3}$ = RepLeftBit(I10,32) then
    Stop after executing zero or more instructions after the halt.

Halt If Greater Than — Required — v 1.0

hgt **ra,rb**



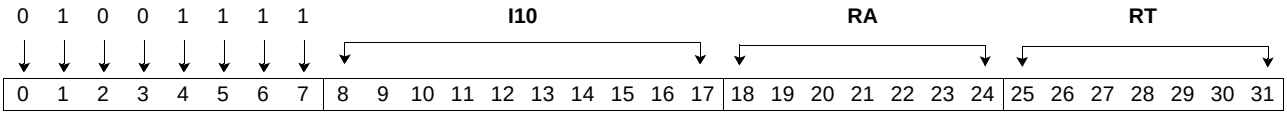
The value in the preferred slot of register RA is algebraically compared with the value in the preferred slot of register RB. If the value from register RA is greater than the RB value, execution of the SPU program stops at or after the halt instruction.

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

hgti **ra,symbol**



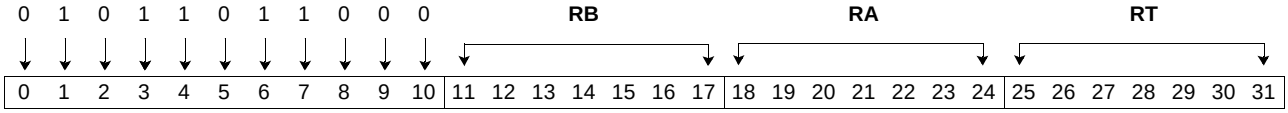
The value in the I10 field is extended to 32 bits by replicating the leftmost bit. The result is algebraically compared to the value in the preferred slot of register RA. If the value from register RA is greater than the immediate value, execution of the SPU program stops at or after the halt instruction.

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 $RA^{0:3}$ > RepLeftBit(I10,32) then
    Stop after executing zero or more instructions after the halt.

Halt If Logically Greater Than — Required — v 1.0

hlgt **ra,rb**



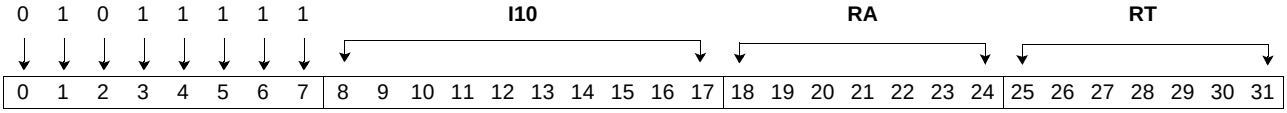
The value in the preferred slot of register RA is logically compared with the value in the preferred slot of register RB. If the value from register RA is greater than the value from register RB, execution of the SPU program stops at or after the halt instruction.

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 $RA^{0:3} >^{u} RB^{0:3}$ then
    Stop after executing zero or more instructions after the halt.

Halt If Logically Greater Than Immediate — Required — v 1.0

hlgti **ra,symbol**



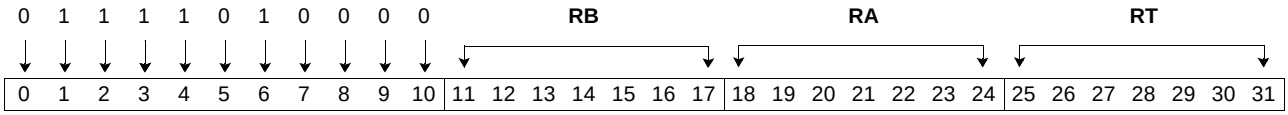
The value in the I10 field is extended to 32 bits by replicating the leftmost bit. The result is logically compared to the value in the preferred slot of register RA. If the value from register RA is logically greater than the immediate value, execution of the SPU program stops at or after the halt instruction.

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 $RA^{0:3} >^{u}$ RepLeftBit(I10,32) then
    Stop after executing zero or more instructions after the halt.

Compare Equal Byte — Required — v 1.0

ceqb **rt,ra,rb**



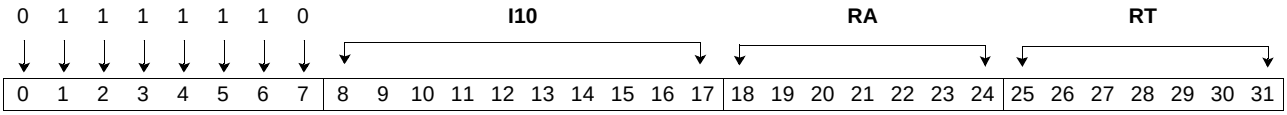
For each of 16 byte slots:

- 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 — Required — v 1.0

ceqbi **rt,ra,value**



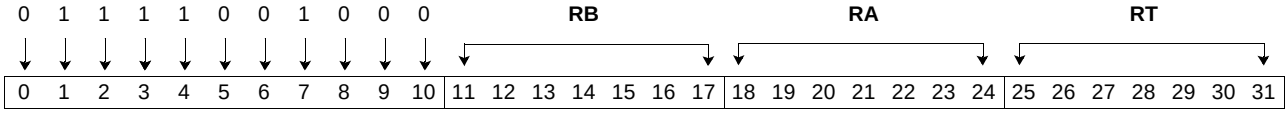
For each of 16 byte slots:

- 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.

```
for i = 0 to 15
        If RA^i = I10_{2:9} then      RT^i ← 0xFF
        else                          RT^i ← 0x00
end
```

Compare Equal Halfword — Required — v 1.0

ceqh **rt,ra,rb**



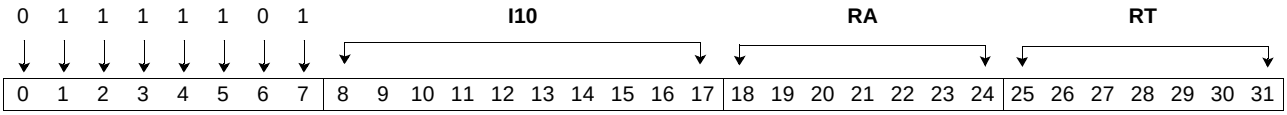
For each of 8 halfword slots:

- 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.

```
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 — Required — v 1.0

ceqhi **rt,ra,value**



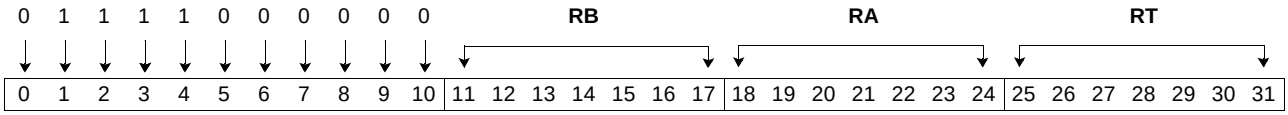
For each of eight halfword slots:

- 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.

```
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
```

Compare Equal Word — Required — v 1.0

ceq **rt,ra,rb**



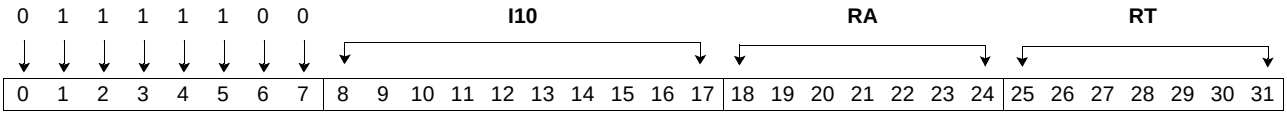
For each of four word slots:

- 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.

```
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 — Required — v 1.0

ceqi **rt,ra,value**



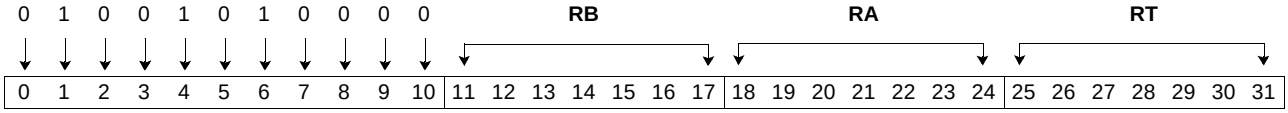
For each of four word slots:

- 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**



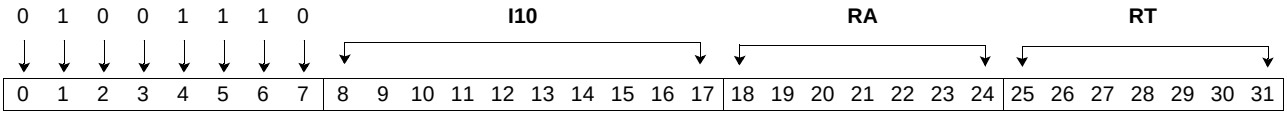
For each of 16 byte slots:

- 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 — Required — v 1.0

cgtbi **rt,ra,value**



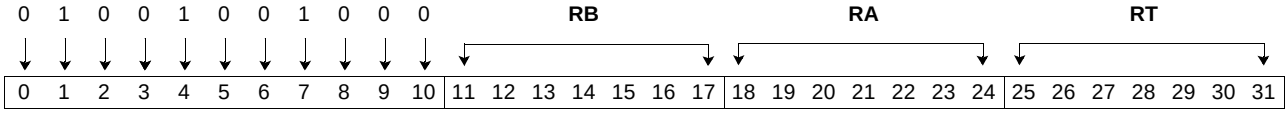
For each of 16 byte slots:

- 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.

```
for i = 0 to 15
        If RA^i > I10_{2:9} then      RT^i ← 0xFF
        else                          RT^i ← 0x00
end
```

Compare Greater Than Halfword — Required — v 1.0

cgth **rt,ra,rb**



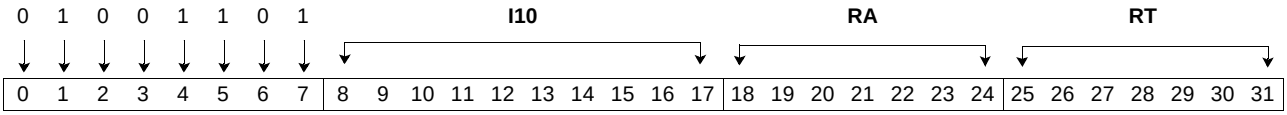
For each of 8 halfword slots:

- 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.

```
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 Greater Than Halfword Immediate — Required — v 1.0

cgthi **rt,ra,value**



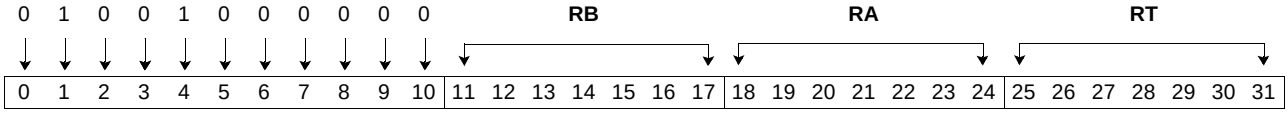
For each of eight halfword slots:

- 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.

```
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
```

Compare Greater Than Word — Required — v 1.0

cgt **rt,ra,rb**



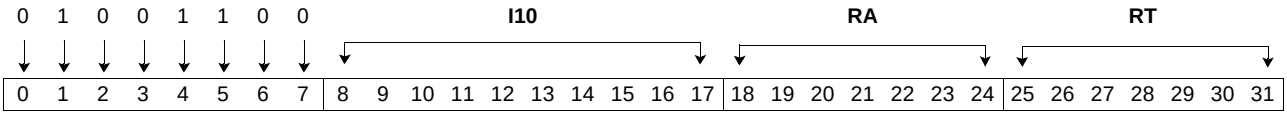
For each of four word slots:

- 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 — Required — v 1.0

cgti **rt,ra,value**



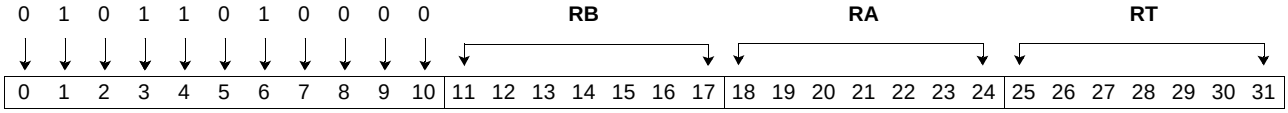
For each of four word slots:

- 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.

```
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 — Required — v 1.0

clgtb **rt,ra,rb**



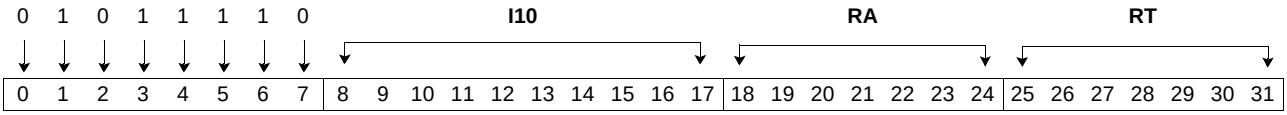
For each of 16 byte slots:

- 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.

```
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 Required v 1.0

clgtbi rt,ra,value



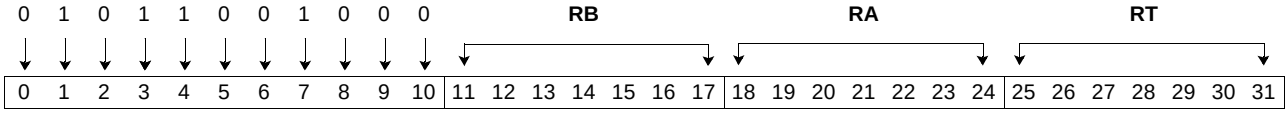
For each of 16 byte slots:

- 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.

```
for i = 0 to 15
        If RA^i >^u I10_2:9 then      RT^i ← 0xFF
        else                          RT^i ← 0x00
end
```

Compare Logical Greater Than Halfword — Required — v 1.0

clgth **rt,ra,rb**



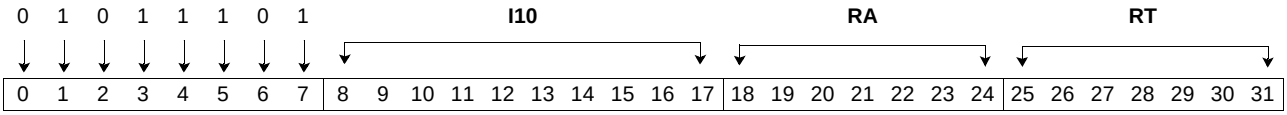
For each of eight halfword slots:

- 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 Required v 1.0

clgthi rt,ra,value



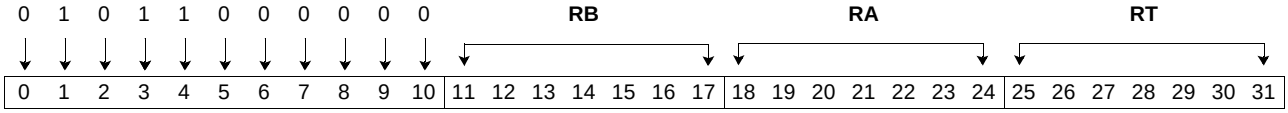
For each of eight halfword slots:

- 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.

```
for i = 0 to 15 by 2
        If RA^i::2 >^u RepLeftBit(I10,16) then      RT^i::2 ← 0xFFFF
        else                                        RT^i::2 ← 0x0000
end
```

Compare Logical Greater Than Word — Required — v 1.0

clgt **rt,ra,rb**



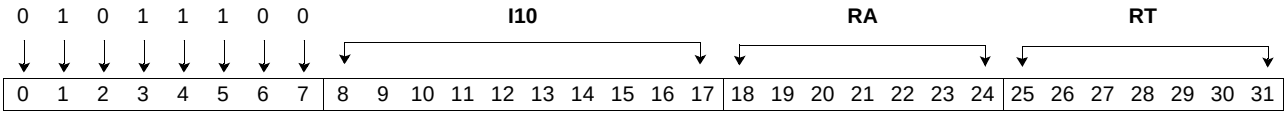
For each of four word slots:

- 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.

```
for i = 0 to 15 by 4
        If RA^i::4 >^u RB^i::4 then    RT^i::4 ← 0xFFFFFFFF
        else                           RT^i::4 ← 0x00000000
end
```

Compare Logical Greater Than Word Immediate Required v 1.0

clgti rt,ra,value



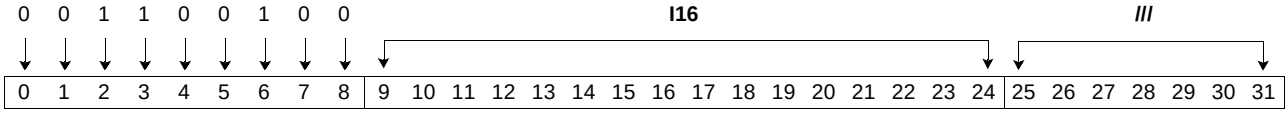
For each of four word slots:

- 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.

```
for i = 0 to 15 by 4
        If RA^i:4 >^u RepLeftBit(I10,32) then    RT^i:4 ← 0xFFFFFFFF
        else                                     RT^i:4 ← 0x00000000
end
```

Branch Relative — Required — v 1.0

br **symbol**



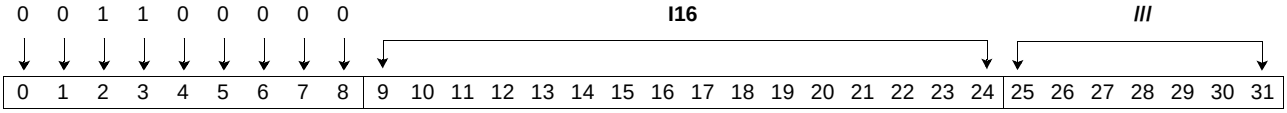
Execution proceeds with the target instruction. The address of the target instruction is computed by adding the value of the I16 field, extended on the right with two zero bits with the result treated as a signed quantity, to the address of the Branch Relative instruction.

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

PC	← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR

Branch Absolute — Required — v 1.0

bra **symbol**

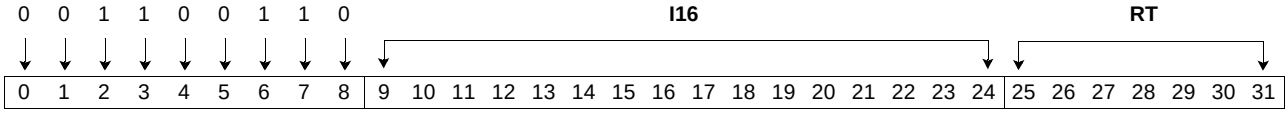


Execution proceeds with the target instruction. The address of the target instruction is the value of the I16 field, extended on the right with two zero bits and extended on the left with copies of the most-significant bit.

PC	← RepLeftBit(I16 || 0b00,32) & LSLR

Branch Relative and Set Link — Required — v 1.0

brsl **rt,symbol**



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.

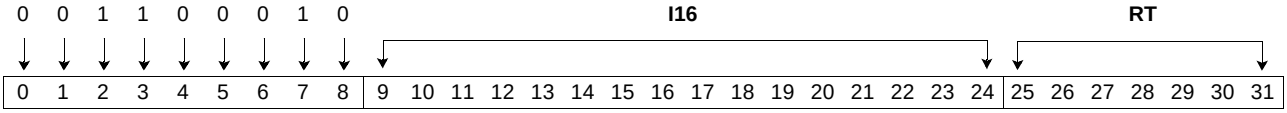
The preferred slot of register RT is set to the address of the byte following the Branch Relative and Set Link instruction. The remaining slots of register RT are set to zero.

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

$RT^{0:3}$	← (PC + 4) & LSLR
$RT^{4:15}$	← 0
PC	← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR

Branch Absolute and Set Link — Required — v 1.0

brasl **rt,symbol**



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

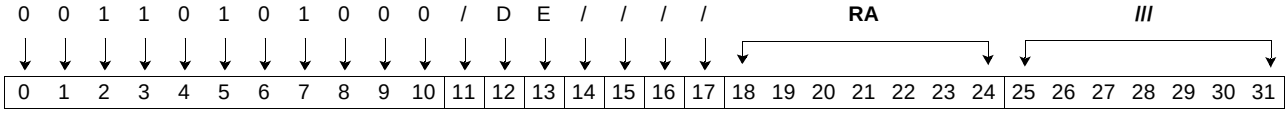
The address of the target instruction is the value of the I16 field, extended on the right with two zero bits and extended on the left with copies of the most-significant bit.

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

Branch Indirect — Required — v 1.0

bi **ra**

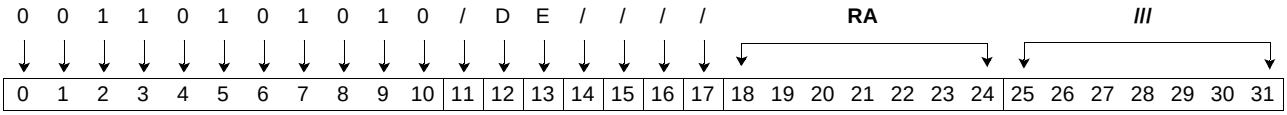


Execution proceeds with the instruction addressed by the preferred slot of register RA. The rightmost 2 bits of the value in register RA are ignored and assumed to be zero. Interrupts can be enabled or disabled with the E or D feature bits (see *Section 12 SPU Interrupt Facility* on page 251).

PC ← $RA^{0:3}$ & LSLR & 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

Interrupt Return — Required — v 1.0

iret **ra**



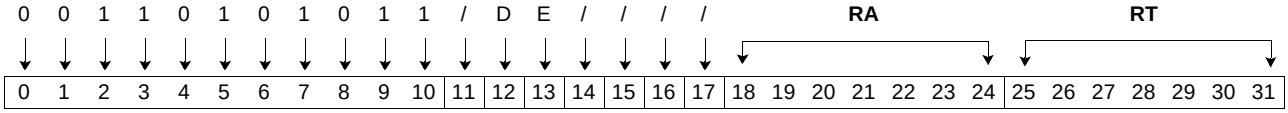
Execution proceeds with the instruction addressed by SRR0. RA is considered to be a valid source whose value is ignored. Interrupts can be enabled or disabled with the E or D feature bits (see *Section 12 SPU Interrupt Facility* on page 251).

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 — Required — v 1.0

bisled **rt,ra**



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.

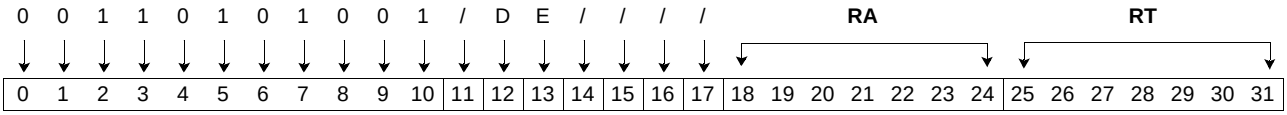
If the branch is taken, interrupts can be enabled or disabled with the E or D feature bits (see *Section 12 SPU Interrupt Facility* on page 251).

```
u ← LSLR & (PC + 4)
t ← RA^0:3 & LSLR & 0xFFFFFFFC
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 — Required — v 1.0

bisl **rt,ra**



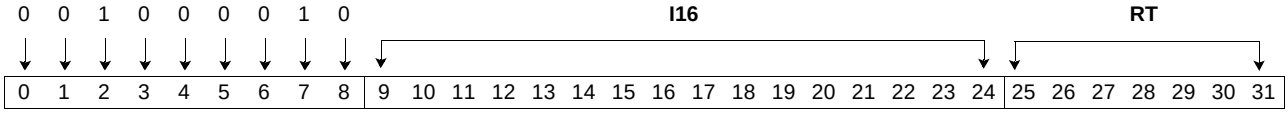
The effective address of the next instruction is taken from the preferred word slot of register RA, with the rightmost 2 bits assumed to be zero. The address of the instruction following the **bisl** instruction is placed into the preferred word slot of register RT. The remainder of register RT is set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

$t \leftarrow RA^{0:3}$ & LSLR & 0xFFFFFFFC
u ← LSLR & (PC + 4)
$RT^{0:3} \leftarrow u$
$RT^{4:15} \leftarrow$ 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**



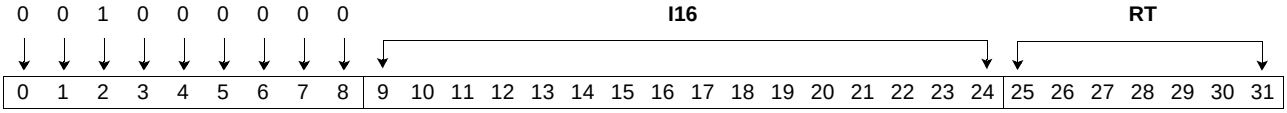
Examine the preferred slot; if it is not zero, proceed with the branch target. Otherwise, proceed with the next instruction.

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 ≠ 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC+4) & LSLR
```

Branch If Zero Word — Required — v 1.0

brz **rt,symbol**



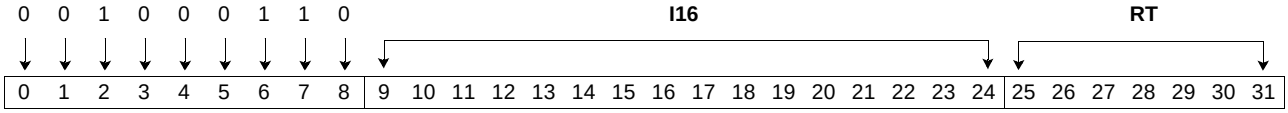
Examine the preferred slot. If it is zero, proceed with the branch target. Otherwise, proceed with the next instruction.

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.

```
If RT^0:3 = 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC + 4) & LSLR
```

Branch If Not Zero Halfword — Required — v 1.0

brhnz **rt,symbol**



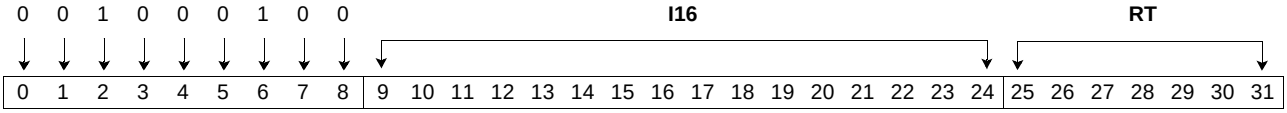
Examine the preferred slot. If the rightmost halfword is not zero, proceed with the branch target. Otherwise, proceed with the next instruction.

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 — Required — v 1.0

brhz **rt,symbol**



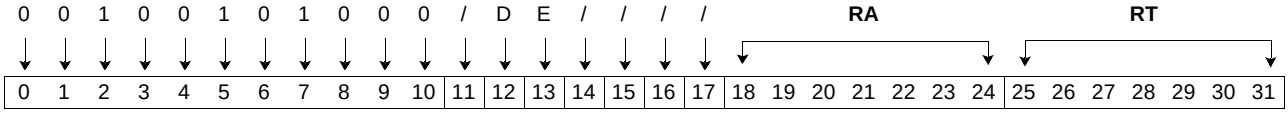
Examine the preferred slot. If the rightmost halfword is zero, proceed with the branch target. Otherwise, proceed with the next instruction.

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 RT2:3 = 0 then
        PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
else
        PC ← (PC + 4) & LSLR
```

Branch Indirect If Zero — Required — v 1.0

biz rt,ra



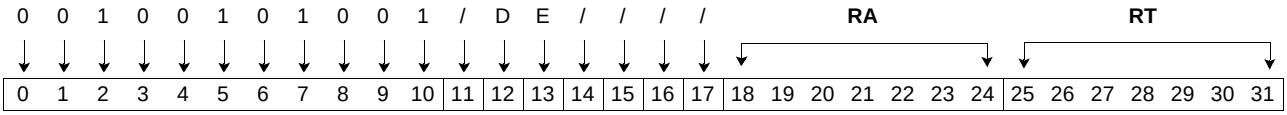
If the preferred slot of register RT is not zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled or disabled with the E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

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

If RT^0:3 = 0 then
        PC ← t & LSLR & 0xFFFF FFFC
        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 — Required — v 1.0

binz **rt,ra**



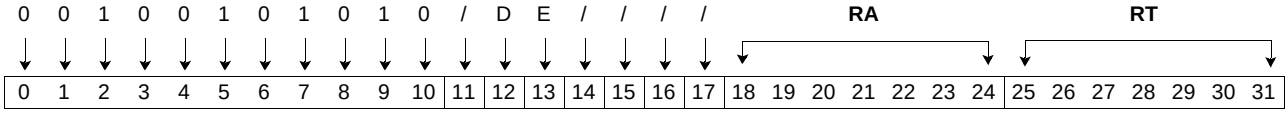
If the preferred slot of register RT is zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled or disabled with the E or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

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

If RT^0:3 != 0 then
        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 Zero Halfword — Required — v 1.0

bihz rt,ra



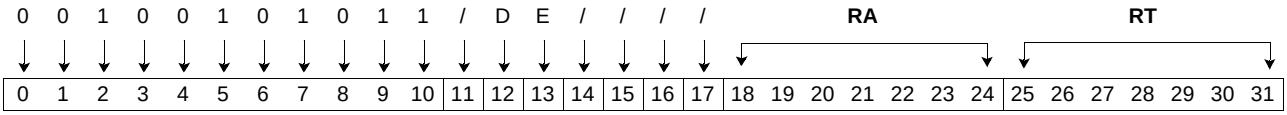
If the rightmost halfword of the preferred slot of register RT is not zero, execution proceeds with the next sequential instruction. Otherwise, execution proceeds at the address in the preferred slot of register RA, treating the rightmost 2 bits as zero. If the branch is taken, interrupts can be enabled 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

bihnz **rt,ra**



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).

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

If RT^2:3 != 0 then
        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
```

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.

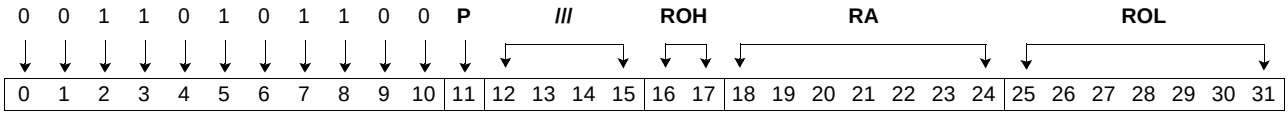
Each of the hint-for-branch instructions specifies the address of a branch instruction and the address of the expected branch target address. If the expectation is that the branch is not taken, the target address is the address of the instruction following the branch.

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

hbr **brinst,brtarg**



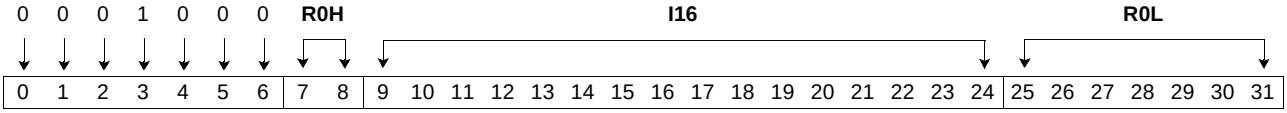
The address of the branch target is given by the contents of the preferred slot of register RA. The RO field gives the signed word offset from the **hbr** instruction to the branch instruction.

If the P feature bit is set, **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.

branch target address ← $RA^{0:3}$ & LSLR & 0xFFFFFFFC
branch instruction address ← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & LSLR

Hint for Branch (a-form) Required v 1.0

hbra **brinst,brtarg**



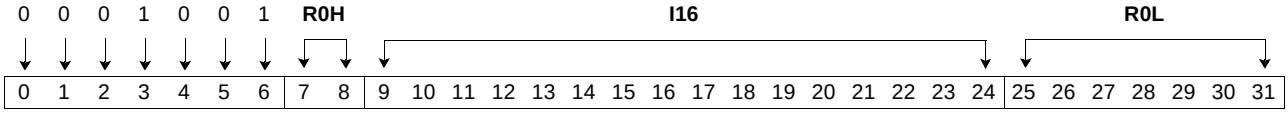
The address of the branch target is specified by an address in the I16 field. The value has 2 bits of zero appended on the right before it is used.

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

```
branch target address ← RepLeftBit(I16 || 0b00,32) & LSLR
branch instruction address ← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & LSLR
```

Hint for Branch Relative — Required — v 1.0

hbrr **brinst,brtarg**



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.

branch target address ← (RepLeftBit(I16 || 0b00,32) + PC) & LSLR
branch instruction address ← (RepLeftBit(ROH || ROL || 0b00,32) + PC) & 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

Number Format	Minimum Positive Magnitude (Smin)			Maximum Positive Magnitude (Smax)			Notes
Register Value	0x00800000			0x7FFFFFFF			
Bit Fields	Sign	8-Bit Biased Exponent	Fraction (implied [1] and 23 bits)	Sign	8-Bit Biased Exponent	Fraction (implied [1] and 23 bits)	1
	0	00000001	[1.]000...000	0	11111111	[1.]111...111	
Value in Powers of 2	+	$2^{(1 - 127)}$	1	+	$2^{(255 - 127)}$	$2 - 2^{-23}$	2
Combined Exponent and Fraction	$2^{-126} * (+1)$			$2^{128} * (+[2 - 2^{-23}])$			
Value of Register in Decimal	$1.2 * 10^{-38}$			$6.8 * 10^{38}$			

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 9-2. Instructions and Exception Settings

Instruction	Set OVF	Set UNF	Set DBZ	Set DIFF
fa, fs, fm, fma, fms, fnms, fi	Yes	Yes	No	Yes
frest, frsqest	No	No	Yes	No
csflt, cuflt	Yes	Yes	No	Yes
cflts, cfltu, fceq, fcneq, fcgt, fcmgt	No	No	No	No

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

Number Format	Minimum Positive Denormalized Magnitude (Dmin)			Maximum Positive Normalized Magnitude (Dmax)			Notes
Register Value	0x0000000000000001			0x7FEFFFFFFFFFFFFF			
Bit Fields	Sign	11-Bit Biased Exponent	Fraction (implied [0] and 52 bits for denormal-ized number)	Sign	11-Bit Biased Exponent	Fraction (implied [1] and 52 bits for normalized number)	1
	0	00000000000	[0.]000...001	0	11111111110	[1.]111...111	2
Value in Powers of 2	+	$2^{(0 + 1 - 1023)}$	2^{-52}	+	$2^{(2046 - 1023)}$	$2 - 2^{-52}$	3,4
Combined Exponent and Fraction	$2^{-1022} * (+2^{-52})$			$2^{1023} * (+[2 - 2^{-52}])$			
Value of Register in Decimal	$4.9 * 10^{-324}$			$1.8 * 10^{308}$			

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 $2 - 2^{-52}$ 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 modes 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

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

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:

UNF = tiny AND 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:

UNF = tiny_before_rounding AND 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,

```
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

Instruction	Set OVF	Set UNF	Set INX	Set INV	Set NAN	Set DENORM
dfa, dfs, dfm, dfma, dfms, dfnms, dfnma	Yes	Yes	Yes	Yes	Yes	Yes
fesd	No	No	No	Yes	Yes	Yes
frds	Yes	Yes	Yes	Yes	Yes	Yes

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.

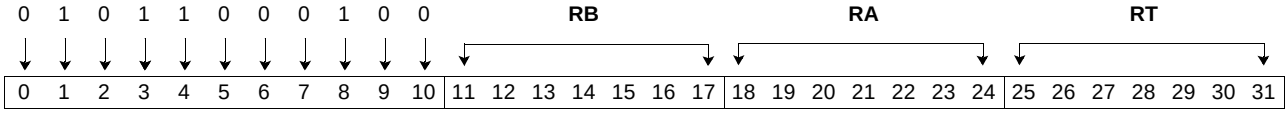
The format of the FPSCR is as follows.

Bits	Description
0:19	Unused
20:21	Rounding control for slice 0 of the 2-way SIMD double-precision operations (RN0) 00 Round to nearest even 01 Round towards zero (truncate) 10 Round towards +infinity 11 Round towards -infinity
22:23	Rounding control for slice 1 of the 2-way SIMD double-precision operations (RN1) 00 Round to nearest even 01 Round towards zero (truncate) 10 Round towards +infinity 11 Round towards -infinity
24:28	Unused
29:31	Single-precision exception flags for slice 0 29 Overflow (OVF) 30 Underflow (UNF) 31 Result produced with extended-range arithmetic could be different from the IEEE compliant result (DIFF)
32:49	Unused
50:55	IEEE exception flags for slice 0 of the 2-way SIMD double-precision operations 50 Overflow (OVF) 51 Underflow (UNF) 52 Inexact result (INX) 53 Invalid operation (INV) 54 Possibly noncompliant result because of QNaN propagation (NaN) 55 Possibly noncompliant result because of denormal operand (DENORM)
56:60	Unused

Bits	Description
61:63	Single-precision exception flags for slice 1 (OVF, UNF, DIFF)
64:81	Unused
82:87	IEEE exception flags for slice 1 of the 2-way SIMD double-precision operations (OVF, UNF, INX, INV, NAN, DENORM)
88:92	Unused
93:95	Single-precision exception flags for slice 2 (OVF, UNF, DIFF)
96:115	Unused
116:119	Single-precision divide-by-zero flags for each of the four slices 116 DBZ for slice 0 117 DBZ for slice 1 118 DBZ for slice 2 119 DBZ for slice 3
120:124	Unused
125:127	Single-precision exception flags for slice 3 (OVF, UNF, DIFF)

Floating Add — Required — v 1.0

fa **rt,ra,rb**

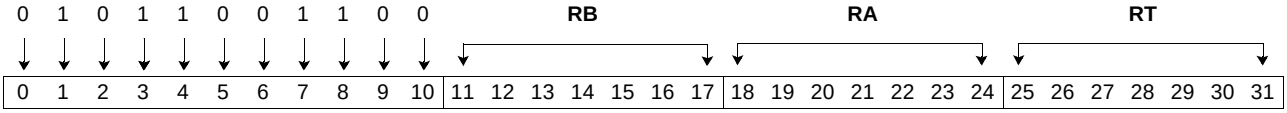


For each of the four word slots:

- 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 — Required — v 1.0

dfa **rt,ra,rb**

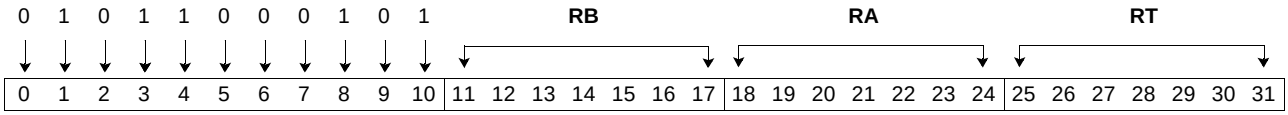


For each of two doubleword slots:

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

Floating Subtract — Required — v 1.0

fs **rt,ra,rb**

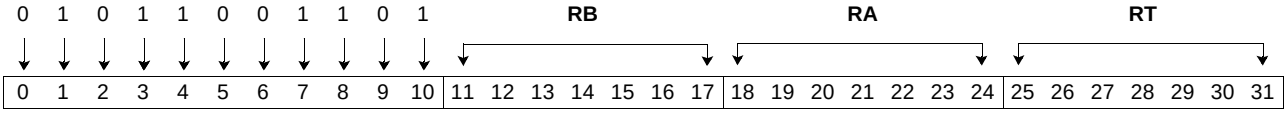


For each of the four word slots:

- 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 — Required — v 1.0

dfs **rt,ra,rb**

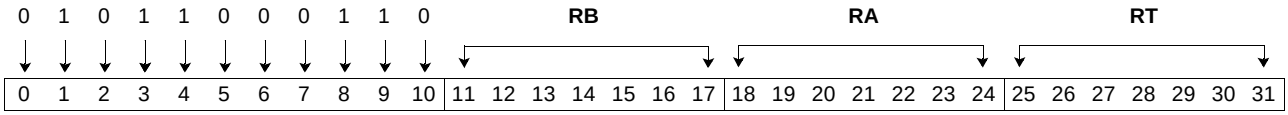


For each of two doubleword slots:

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

Floating Multiply — Required — v 1.0

fm **rt,ra,rb**

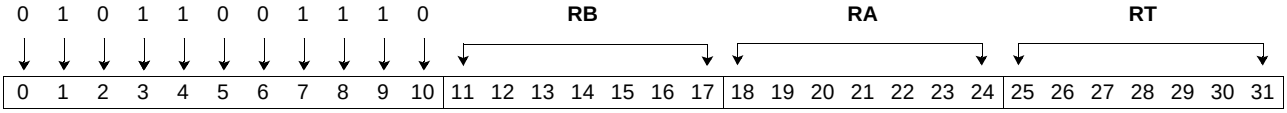


For each of the four word slots:

- 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 — Required — v 1.0

dfm **rt,ra,rb**

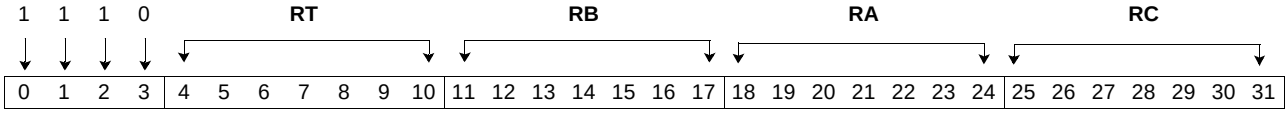


For each of two doubleword slots:

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

Floating Multiply and Add — Required — v 1.0

fma **rt,ra,rb,rc**

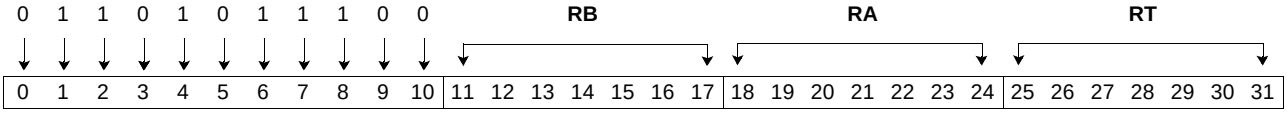


For each of the four word slots:

- The operand from register RA is multiplied by the operand from register RB and 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 — Required — v 1.0

dfma **rt,ra,rb**

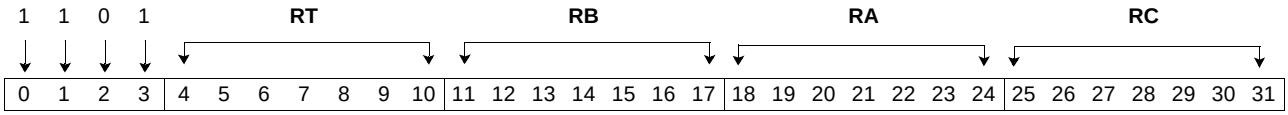


For each of two doubleword slots:

- 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 is placed in register RT.

Floating Negative Multiply and Subtract — Required — v 1.0

fnms **rt,ra,rb,rc**

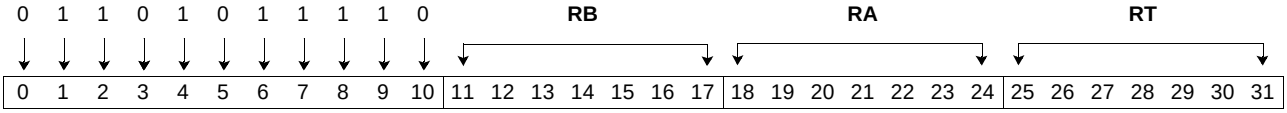


For each of the four word slots:

- 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 Required v 1.0

dfnms rt,ra,rb

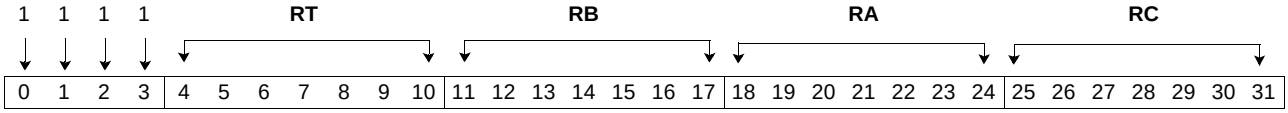


For each of two doubleword slots:

- 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**

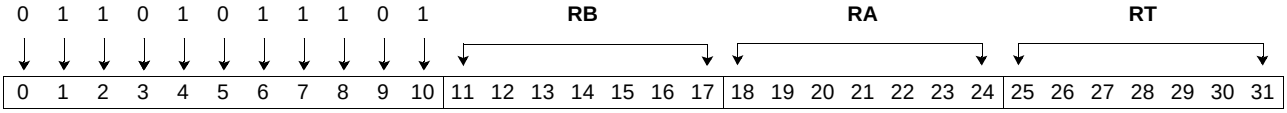


For each of the four word slots:

- 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 — Required — v 1.0

dfms **rt,ra,rb**

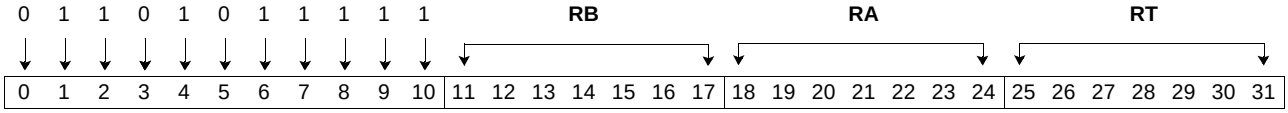


For each of two doubleword slots:

- 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 — Required — v 1.0

dfnma **rt,ra,rb**

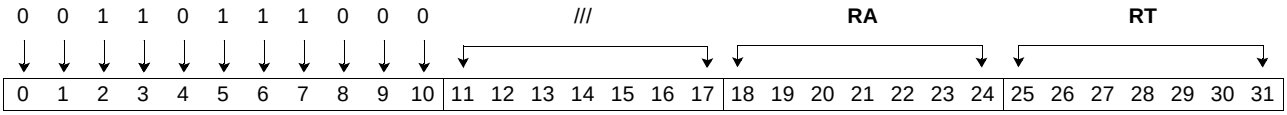


For each of two doubleword slots:

- 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.

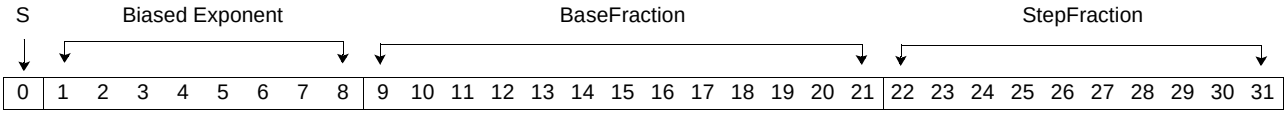
Floating Reciprocal Estimate — Required — v 1.0

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.



- 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:

Base	S 1.BaseFraction * $2^{BiasedExponent - 127}$
Step	0.000 StepFraction * $2^{BiasedExponent - 127}$

- 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'7FFF FFFF' ($1.99\overline{9} * 2^{128}$)

Big

If $|x| \geq 2^{126}$, then $1/x$ underflows to zero, $y2 = 0$.

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:

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

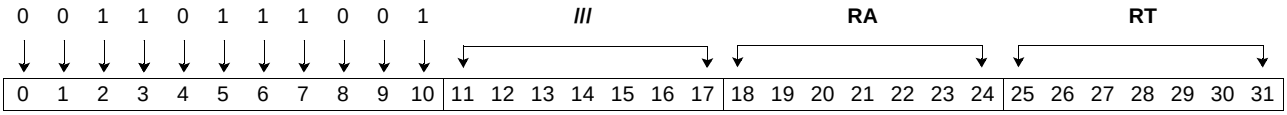
Normal

$1/x = Y$ where $x * Y < 1.0$ and $x * INC(Y) \geq 1.0$.
$INC(y)$ gives the sfp number with the same sign as y and next larger magnitude.
The absolute error bound is:

$| Y - y2 | \leq 1$ ulp (either y2 = Y, or INC(y2) = Y)

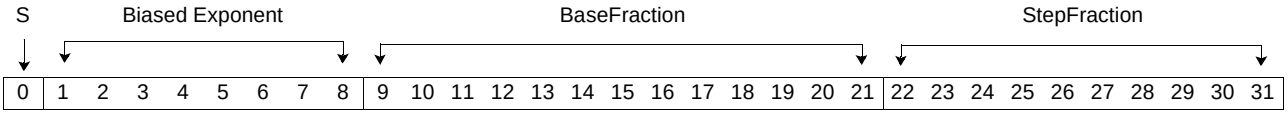
Floating Reciprocal Absolute Square Root Estimate Required v 1.0

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 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:

```
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 ($1.999\overline{9} * 2^{128}$)

Zeros where: x fraction > 0x000ff53c, y2 $\geq$ 0x7fc00000

The following sequence could be used to correct the answer:

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

Normal

$1/sqrt(x) = Y$ where $x * Y^2 < 1.0$ and $x * INC(Y)^2 \geq 1.0$

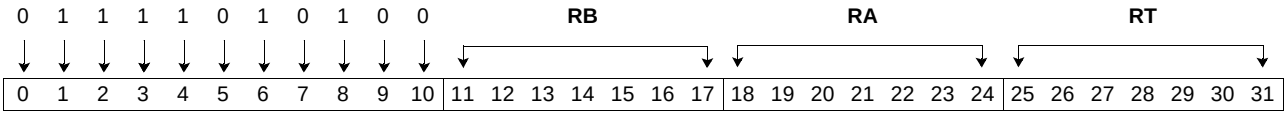
$INC(y)$ gives the sfp number with the same sign as y and next larger magnitude.

The absolute error bound is:

$|\ Y - y2\ | \leq$ 1 ulp (0 and ± 1 are all possible)

Floating Interpolate — Required — v 1.0

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, $Y \leftarrow 0.RA_{13:31}$.

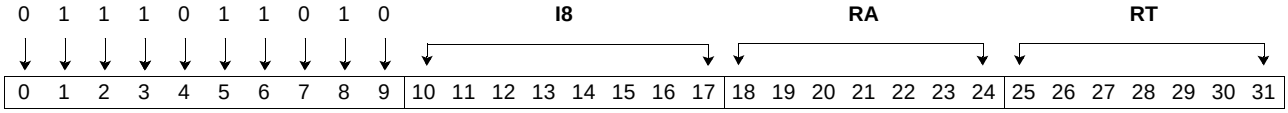
  The result is computed by the following equation:

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

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 — Required — v 1.0

csflt **rt,ra,scale**

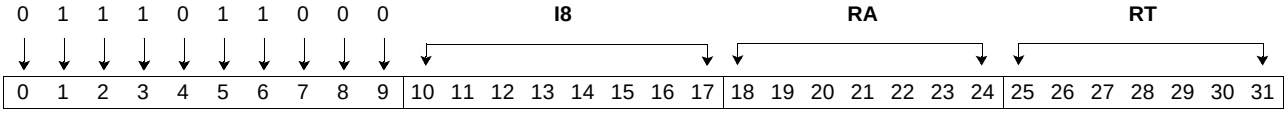


For each of four word slots:

- 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 — Required — v 1.0

cflts **rt,ra,scale**

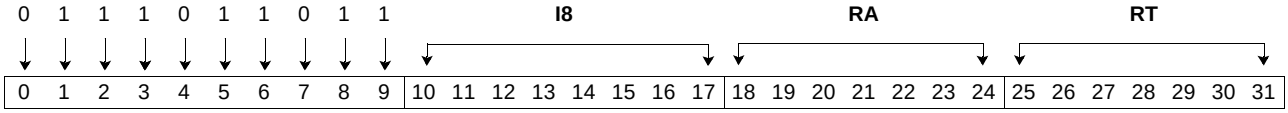


For each of four word slots:

- 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 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.

Convert Unsigned Integer to Floating — Required — v 1.0

cuflt **rt,ra,scale**

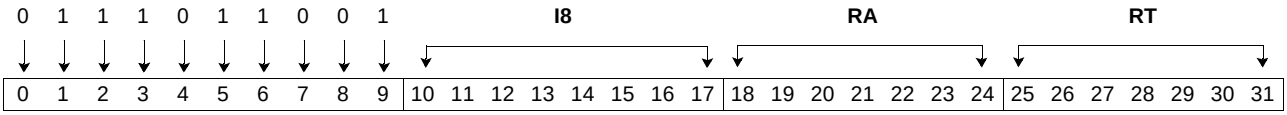


For each of four word slots:

- 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^{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 — Required — v 1.0

cfltu **rt,ra,scale**

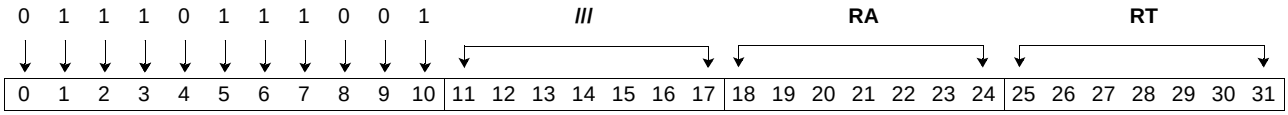


For each of four word slots:

- 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 — Required — v 1.0

frds **rt,ra**

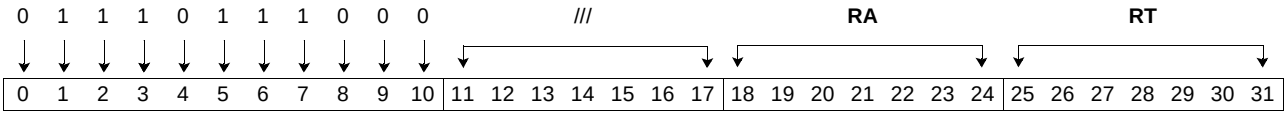


For each of two doubleword slots:

- 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 — Required — v 1.0

fesd **rt,ra**

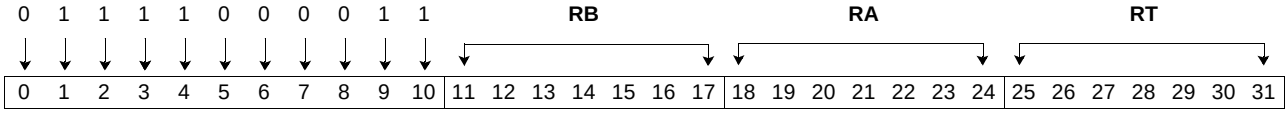


For each of two doubleword slots:

- 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**

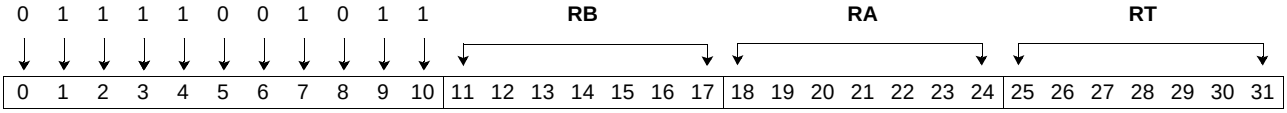


For each of the two doubleword slots:

- 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**

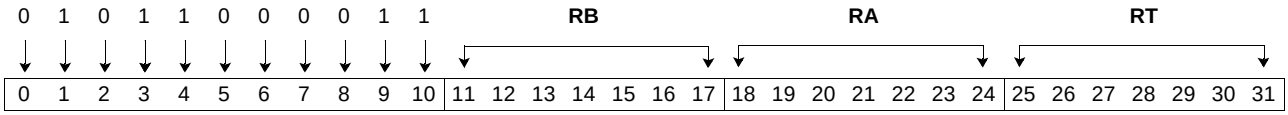


For each of the two doubleword slots:

- 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 — Optional — v 1.2

dfcgt **rt,ra,rb**

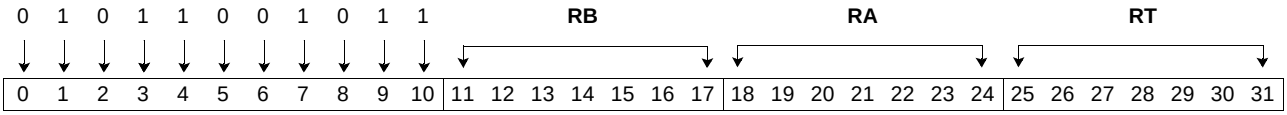


For each of the two doubleword slots:

- 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 Optional v 1.2

dfcmgt rt,ra,rb

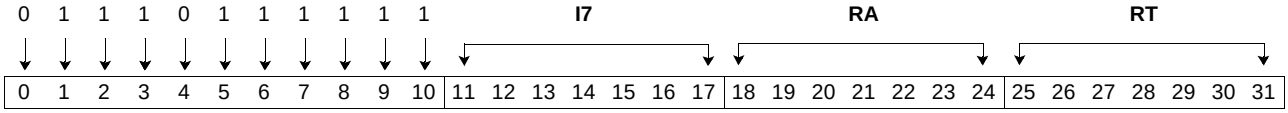


For each of the two doubleword slots:

- 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 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:

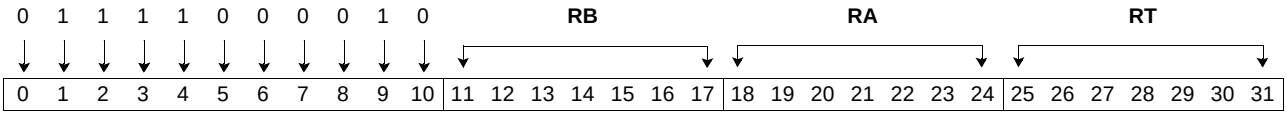
- The double-precision floating-point value in register RA is tested for special values. The bits of I7 enable the following seven checks

I7	RA Value Category
1000000	NaN
0100000	+Infinity
0010000	-Infinity
0001000	+0
0000100	-0
0000010	Positive Denorm
0000001	Negative Denorm

- 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 — Required — v 1.0

fceq **rt,ra,rb**

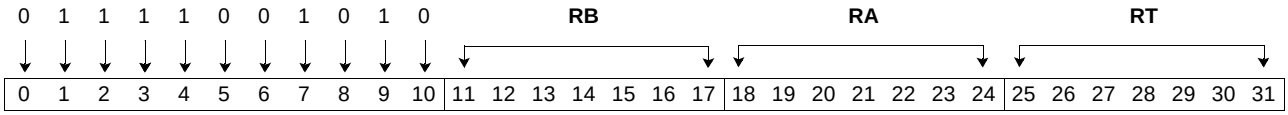


For each of four word slots:

- 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 — Required — v 1.0

fcmeq **rt,ra,rb**

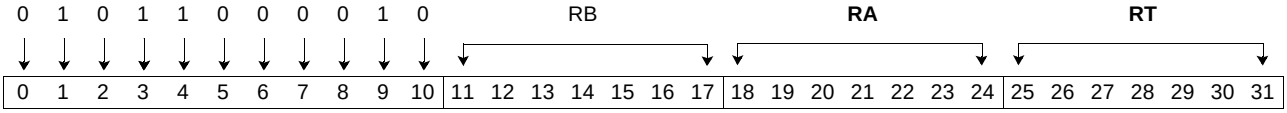


For each of four word slots:

- 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 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 — Required — v 1.0

fcgt **rt,ra,rb**

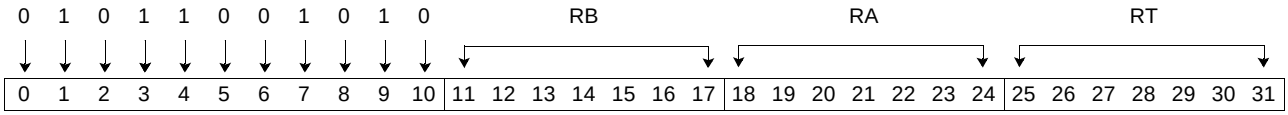


For each of four word slots:

- 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 Required v 1.0

fcmgt rt,ra,rb

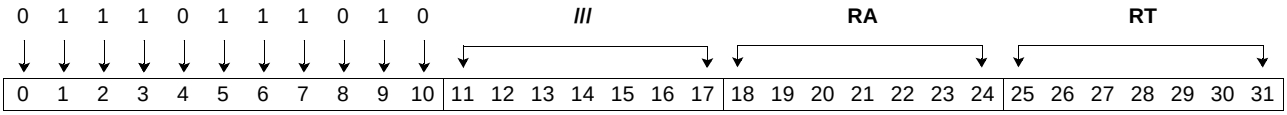


For each of four word slots:

- 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 — Required — v 1.0

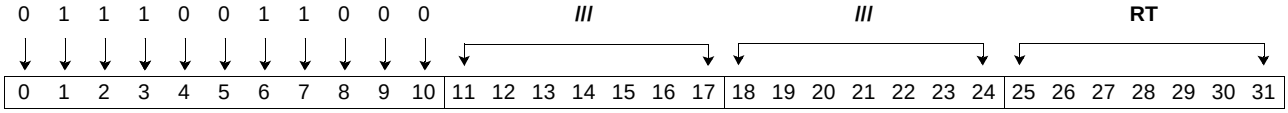
fscrwr **ra**



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 Required v 1.0

fscrrd rt



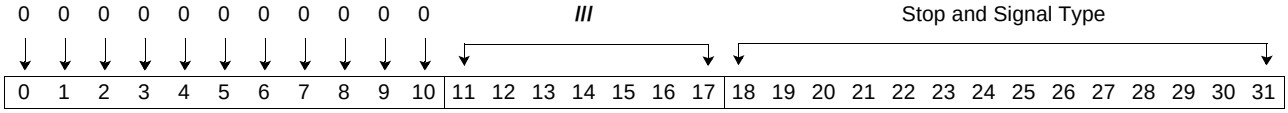
This instruction reads the value of the FPSCR. In the result, the unused bits of the FPSCR are forced to zero. The result is placed in the register RT.

10. Control Instructions

This section lists and describes the SPU control instructions.

Stop and Signal — Required — v 1.0

stop

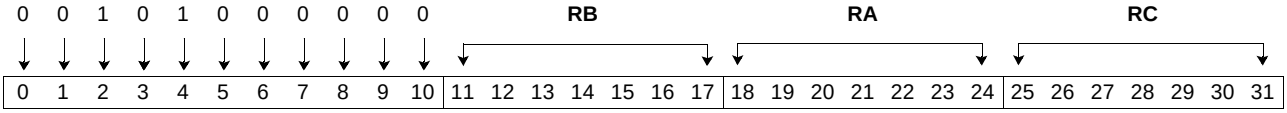


Execution of the program in the SPU stops, and the external environment is signaled. No further instructions are executed.

PC ← PC + 4 & LSLR
precise stop

Stop and Signal with Dependencies — Required — v 1.0

stopd



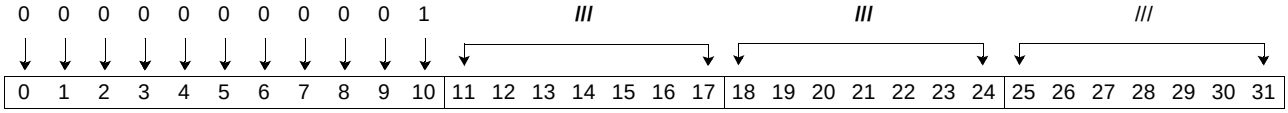
Execution of the program in the SPU stops.

PC ← PC + 4 & LSLR
precise stop

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) Required v 1.0

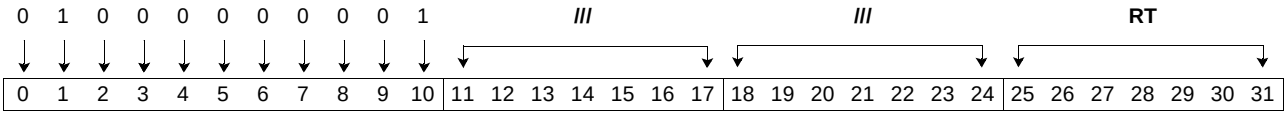
lnop



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

No Operation (Execute) — Required — v 1.0

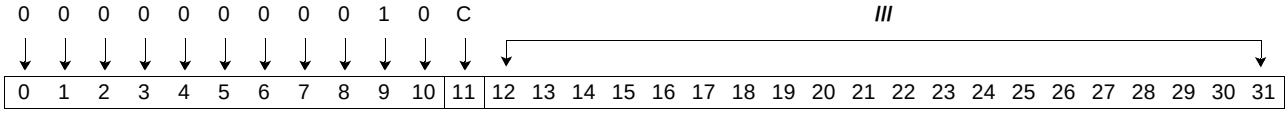
nop



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.

Synchronize Required v 1.0

sync

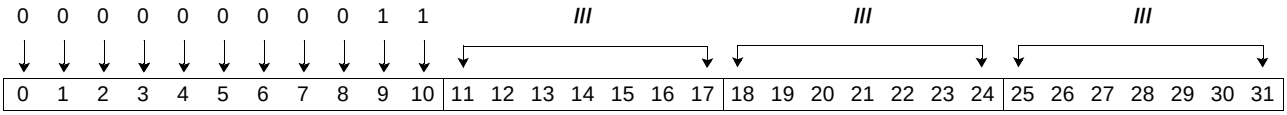


This instruction has no effect on the execution of the program other than to cause the processor to wait until all pending store instructions have completed before fetching the next sequential instruction. This instruction must be used following a store instruction that modifies the instruction stream.

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 — Required — v 1.0

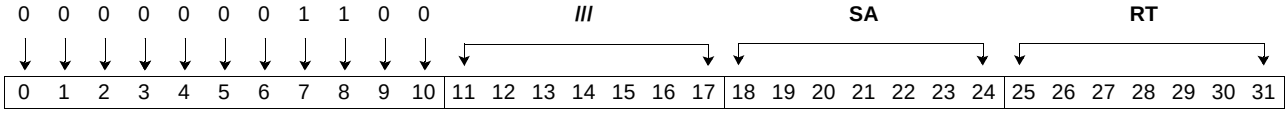
dsync



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 Required v 1.0

mfspr rt,sa



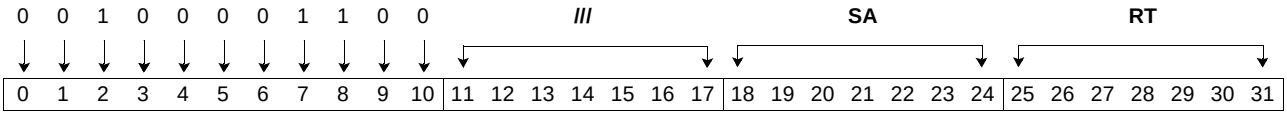
Special-Purpose Register SA is copied into register RT. If SPR SA is not defined, zeros are supplied.

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
```

Move to Special-Purpose Register — Required — v 1.0

mtspr **sa, rt**



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.

```
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.

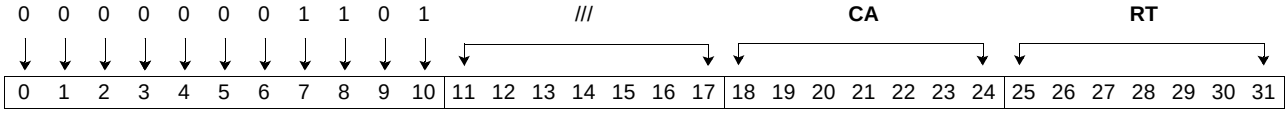
An implementation can implement any number of channels up to 128. Each channel has a channel number in the range 0-127. Channel numbers have no particular significance, and there is no relationship between the direction of a channel and its number.

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 — Required — v 1.0

rdch **rt,ca**



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.

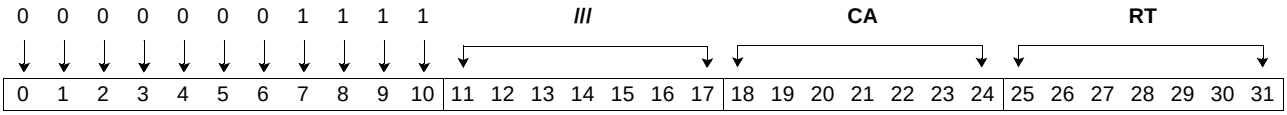
Note: The SPU ISA defines the **rdch** and **wrch** instructions as 128-bit operations. An implementation might define 32-bit wide channels. In that case, the 32-bit value occupies the preferred slot; the other slots return zeros.

```
if readable(Channel(CA)) then
        RT ← Channel(CA)
else
        Stop after executing zero or more instructions after the rdch.
```

Read Channel Count

Required **v 1.0**

rchcnt **rt,ca**

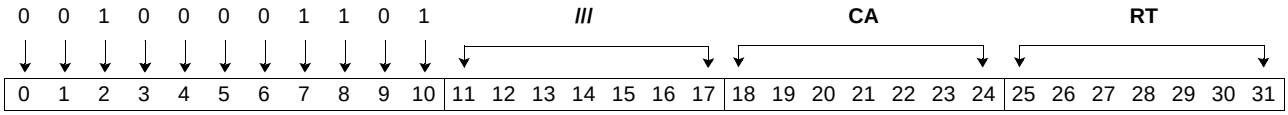


The channel capacity of channel CA is placed into the preferred slot of register RT. The channel capacity of unimplemented channels is zero.

$RT^{0:3} \leftarrow$ Channel Capacity(CA)
$RT^{4:15} \leftarrow 0$

Write Channel — Required — v 1.0

wrch **ca,rt**



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.

```
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

Feature Bit Setting		Result
[D]	[E]	
0	0	Status does not change.
0	1	Interrupt processing is enabled.
1	0	Interrupt processing is disabled.
1	1	Causes undefined behavior.

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

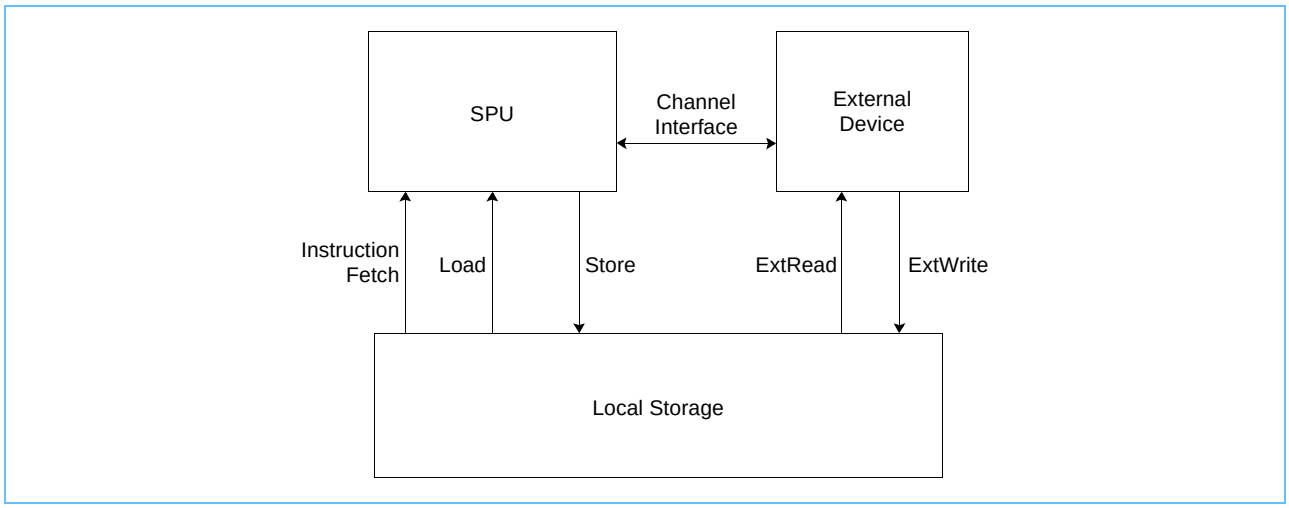


Table 13-1 defines five transactions serviced by local storage. The SPU ISA does not define the behavior of the external device or how the external device accesses local storage. When this document refers to an external write of local storage, it assumes the external device delivers data to local storage such that a subsequent SPU load from local storage can retrieve the data.

Table 13-1. Local Storage Accesses

Name	Description
Load	SPU load instruction gets data from local storage read.
Store	SPU store instruction sends data to local storage write.
Fetch	SPU instruction fetch gets data from local storage read.
ExtWrite	External device sends data to local storage write.
ExtRead	External device gets data from local storage read.

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.c** 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

Instruction	Consistency Effects	Instruction Serialization Effects
dsync	Ensures that subsequent external reads access data written by prior stores. Ensures that subsequent loads access data written by external writes.	Forces load and store access of local storage because of instructions before the **dsync** to be completed before completion of **dsync**. Forces read channel operations because of instructions before the **dsync** to be completed before completion of the **dsync**. Forces load and store access of local storage because of instructions after the **dsync** to occur after completion of the **dsync**. Forces read and write channel operations because of instructions after the **dsync** to occur after completion of the **dsync**.
sync	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.	Forces all access of local storage and channels because of instructions before the **sync** to be completed before completion of **sync**. Forces all access of local storage and channels because of instructions after the **sync** to occur after completion of the **sync**.
sync.c	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.	Forces all access of local storage and channels because of instructions before the **sync.c** to be completed before completion of **sync.c**. Forces all access of local storage and channels because of instructions after the **sync.c** to occur after completion of the **sync.c**.

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.

Table 13-3. Synchronizing Multiple Accesses to Local Storage

Writer	Local Storage Access to be Synchronized with the Local Storage Write				
	Store	Load	Fetch	ExtRead	ExtWrite
Store	nothing	nothing	**sync**	**dsync**	**dsync**
ExtWrite	**dsync**	**dsync**	**sync**	N/A	N/A

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."

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 shows how an SPU program can reliably send to an external device, synchronizing only through the local storage. In this example, an SPU sends data through a buffer at address C to an external reader using a marker in local storage at address D. The SPU begins by storing the data to be transferred. It then executes a **dsync** instruction to force the data into local storage before it stores the marker. The **dsync** instruction also prevents the marker store from being reordered amongst the data stores. After the marker store, the SPU program must execute a **dsync** instruction again to force the marker into local storage.

Table 13-5 shows how data can move from an external writer to the SPU program using local-storage-based synchronization. The SPU program starts by polling for the marker that indicates that data is ready. The polling loop begins with a **dsync** instruction that forces subsequent load instructions to get data from the now current local storage state. When the marker is found, the SPU program must execute a **dsync** instruction

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

External Device		SPU	Comment
		Store data to C	
		dsync	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.
		Store marker to D	
		dsync	Force the store to D to be visible in local storage to external readers.
eloop:	Read D		
	If not marker, goto eloop		
	Read C		

Table 13-5. Receiving Data and Synchronizing through Local Storage

External Device		SPU	Comment
	Write data to A		This is the order in which the external device modifies local storage. The ordering is not controlled by the SPU ISA.
	Write marker to B		
		loop: **dsync**	Force a subsequent load to access local storage, so that the load arriving from B will get new data from local storage.
		Load from B	
		If not marker, goto loop	Ensure A and B are both written to local storage.
		dsync	Force a subsequent load to execute after the load from B. Without this **dsync**, the load from A could be performed before the load from B and get local storage contents before the write to A.
		Load from A	Must get data from the write to A.

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

External Device	SPU	Comment
SPU receives data through local storage address A		
Write data to A		
Send message to channel B		The ordering is not controlled by the SPU ISA.
	rdch B	Wait for message
	dsync	Ensure load from A is executed after **rdch**, and access the data in local storage
	load from A	Must get data
SPU sends data through local storage address C		
	Store data to C	
	dsync	Ensure data is in local storage
	wrch D	Send message
Receive message from channel D		
Read data from C		The ordering is not controlled by the SPU ISA.

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)

Mnemonic	Instruction	Page
a	Add Word	60
absdb	Absolute Differences of Bytes	92
addx	Add Extended	66
ah	Add Halfword	58
ahi	Add Halfword Immediate	59
ai	Add Word Immediate	61
and	And	97
andbi	And Byte Immediate	99
andc	And with Complement	98
andhi	And Halfword Immediate	100
andi	And Word Immediate	101
avgb	Average Bytes	91
bg	Borrow Generate	70
bgx	Borrow Generate Extended	71
bi	Branch Indirect	178
bihnz	Branch Indirect If Not Zero Halfword	189
bihz	Branch Indirect If Zero Halfword	188
binz	Branch Indirect If Not Zero	187
bisl	Branch Indirect and Set Link	181
bisled	Branch Indirect and Set Link if External Data	180
biz	Branch Indirect If Zero	186
br	Branch Relative	174
bra	Branch Absolute	175
brasl	Branch Absolute and Set Link	177
brhnz	Branch If Not Zero Halfword	184
brhz	Branch If Zero Halfword	185
brnz	Branch If Not Zero Word	182
brsl	Branch Relative and Set Link	176
brz	Branch If Zero Word	183
cbd	Generate Controls for Byte Insertion (d-form)	40
cbx	Generate Controls for Byte Insertion (x-form)	41
cdd	Generate Controls for Doubleword Insertion (d-form)	46
cdx	Generate Controls for Doubleword Insertion (x-form)	47
ceq	Compare Equal Word	160
ceqb	Compare Equal Byte	156

Table A-1. Instructions Sorted by Mnemonic (Page 2 of 6)

Mnemonic	Instruction	Page
ceqbi	Compare Equal Byte Immediate	157
ceqh	Compare Equal Halfword	158
ceqhi	Compare Equal Halfword Immediate	159
ceqi	Compare Equal Word Immediate	161
cflts	Convert Floating to Signed Integer	221
cfltu	Convert Floating to Unsigned Integer	223
cg	Carry Generate	67
cgt	Compare Greater Than Word	166
cgtb	Compare Greater Than Byte	162
cgtbi	Compare Greater Than Byte Immediate	163
cgth	Compare Greater Than Halfword	164
cgthi	Compare Greater Than Halfword Immediate	165
cgti	Compare Greater Than Word Immediate	167
cgx	Carry Generate Extended	68
chd	Generate Controls for Halfword Insertion (d-form)	42
chx	Generate Controls for Halfword Insertion (x-form)	43
clgt	Compare Logical Greater Than Word	172
clgtb	Compare Logical Greater Than Byte	168
clgtbi	Compare Logical Greater Than Byte Immediate	169
clgth	Compare Logical Greater Than Halfword	170
clgthi	Compare Logical Greater Than Halfword Immediate	171
clgti	Compare Logical Greater Than Word Immediate	173
clz	Count Leading Zeros	83
cntb	Count Ones in Bytes	84
csflt	Convert Signed Integer to Floating	220
cuflt	Convert Unsigned Integer to Floating	222
cwd	Generate Controls for Word Insertion (d-form)	44
cwx	Generate Controls for Word Insertion (x-form)	45
dfa	Double Floating Add	203
dfceq	Double Floating Compare Equal	226
dfcgt	Double Floating Compare Greater Than	228
dfcmeq	Double Floating Compare Magnitude Equal	227
dfcmgt	Double Floating Compare Magnitude Greater Than	229
dfm	Double Floating Multiply	207
dfma	Double Floating Multiply and Add	209
dfms	Double Floating Multiply and Subtract	213
dfnma	Double Floating Negative Multiply and Add	214

Table A-1. Instructions Sorted by Mnemonic (Page 3 of 6)

Mnemonic	Instruction	Page
dfnms	Double Floating Multiply and Subtract	213
dfs	Double Floating Subtract	205
dftsv	Double Floating Test Special Value	230
dsync	Synchronize Data	243
eqv	Equivalent	114
fa	Floating Add	202
fceq	Floating Compare Equal	231
fcgt	Floating Compare Greater Than	233
fcmeq	Floating Compare Magnitude Equal	232
fcmgt	Floating Compare Magnitude Greater Than	234
fesd	Floating Extend Single to Double	225
fi	Floating Interpolate	219
fm	Floating Multiply	206
fma	Floating Multiply and Add	208
fms	Floating Multiply and Subtract	212
fnms	Floating Negative Multiply and Subtract	210
frds	Floating Round Double to Single	224
frest	Floating Reciprocal Estimate	215
frsqest	Floating Reciprocal Absolute Square Root Estimate	217
fs	Floating Subtract	204
fscrrd	Floating-Point Status and Control Register Write	235
fscrwr	Floating-Point Status and Control Register Read	236
fsm	Form Select Mask for Words	87
fsmb	Form Select Mask for Bytes	85
fsmbi	Form Select Mask for Bytes Immediate	55
fsmh	Form Select Mask for Halfwords	86
gb	Gather Bits from Words	90
gbb	Gather Bits from Bytes	88
gbh	Gather Bits from Halfwords	89
hbr	Hint for Branch (r-form)	192
hbra	Hint for Branch (a-form)	193
hbrr	Hint for Branch Relative	194
heq	Halt If Equal	150
heqi	Halt If Equal Immediate	151
hgt	Halt If Greater Than	152
hgti	Halt If Greater Than Immediate	153
hlgt	Halt If Logically Greater Than	154

Table A-1. Instructions Sorted by Mnemonic (Page 4 of 6)

Mnemonic	Instruction	Page
hlgti	Halt If Logically Greater Than Immediate	155
il	Immediate Load Word	52
ila	Immediate Load Address	53
ilh	Immediate Load Halfword	50
ilhu	Immediate Load Halfword Upper	51
iohl	Immediate Or Halfword Lower	54
iret	Interrupt Return	179
lnop	No Operation (Load)	240
lqa	Load Quadword (a-form)	34
lqd	Load Quadword (d-form)	32
lqr	Load Quadword Instruction Relative (a-form)	35
lqx	Load Quadword (x-form)	33
mfspr	Move from Special-Purpose Register	244
mpy	Multiply	72
mpya	Multiply and Add	76
mpyh	Multiply High	77
mpyhh	Multiply High High	79
mpyhha	Multiply High High and Add	80
mpyhhau	Multiply High High Unsigned and Add	82
mpyhhu	Multiply High High Unsigned	81
mpyi	Multiply Immediate	74
mpys	Multiply and Shift Right	78
mpyu	Multiply Unsigned	73
mpyui	Multiply Unsigned Immediate	75
mtspr	Move to Special-Purpose Register	245
nand	Nand	112
nop	No Operation (Execute)	241
nor	Nor	113
or	Or	102
orbi	Or Byte Immediate	104
orc	Or with Complement	103
orhi	Or Halfword Immediate	105
ori	Or Word Immediate	106
orx	Or Across	107
rchcnt	Read Channel Count	249
rdch	Read Channel	248
rot	Rotate Word	129

Table A-1. Instructions Sorted by Mnemonic (Page 5 of 6)

Mnemonic	Instruction	Page
roth	Rotate Halfword	127
rothi	Rotate Halfword Immediate	128
rothm	Rotate and Mask Halfword	136
rothmi	Rotate and Mask Halfword Immediate	137
roti	Rotate Word Immediate	130
rotm	Rotate and Mask Word	138
rotma	Rotate and Mask Algebraic Word	147
rotmah	Rotate and Mask Algebraic Halfword	145
rotmahi	Rotate and Mask Algebraic Halfword Immediate	146
rotmai	Rotate and Mask Algebraic Word Immediate	148
rotmi	Rotate and Mask Word Immediate	139
rotqbi	Rotate Quadword by Bits	134
rotqbii	Rotate Quadword by Bits Immediate	135
rotqby	Rotate Quadword by Bytes	131
rotqbybi	Rotate Quadword by Bytes from Bit Shift Count	133
rotqbyi	Rotate Quadword by Bytes Immediate	132
rotqmbi	Rotate and Mask Quadword by Bits	143
rotqmbii	Rotate and Mask Quadword by Bits Immediate	144
rotqmby	Rotate and Mask Quadword by Bytes	140
rotqmbybi	Rotate and Mask Quadword Bytes from Bit Shift Count	142
rotqmbyi	Rotate and Mask Quadword by Bytes Immediate	141
selb	Select Bits	115
sf	Subtract from Word	64
sfh	Subtract from Halfword	62
sfhi	Subtract from Halfword Immediate	63
sfi	Subtract from Word Immediate	65
sfx	Subtract from Extended	69
shl	Shift Left Word	120
shlh	Shift Left Halfword	118
shlhi	Shift Left Halfword Immediate	119
shli	Shift Left Word Immediate	121
shlqbi	Shift Left Quadword by Bits	122
shlqbii	Shift Left Quadword by Bits Immediate	123
shlqby	Shift Left Quadword by Bytes	124
shlqbybi	Shift Left Quadword by Bytes from Bit Shift Count	126
shlqbyi	Shift Left Quadword by Bytes Immediate	125
shufb	Shuffle Bytes	116

Table A-1. Instructions Sorted by Mnemonic (Page 6 of 6)

Mnemonic	Instruction	Page
stop	Stop and Signal	238
stopd	Stop and Signal with Dependencies	239
stqa	Store Quadword (a-form)	38
stqd	Store Quadword (d-form)	36
stqr	Store Quadword Instruction Relative (a-form)	39
stqx	Store Quadword (x-form)	37
sumb	Sum Bytes into Halfwords	93
sync	Synchronize	242
wrch	Write Channel	250
xor	Exclusive Or	108
xorbi	Exclusive Or Byte Immediate	109
xorhi	Exclusive Or Halfword Immediate	110
xori	Exclusive Or Word Immediate	111
xsbh	Extend Sign Byte to Halfword	94
xshw	Extend Sign Halfword to Word	95
xswd	Extend Sign Word to Doubleword	96

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

Rightmost 4 Bits of the Effective Address	Created Mask
0000	03 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0001	10 03 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0010	10 11 03 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0011	10 11 12 03 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0100	10 11 12 13 03 15 16 17 18 19 1a 1b 1c 1d 1e 1f
0101	10 11 12 13 14 03 16 17 18 19 1a 1b 1c 1d 1e 1f
0110	10 11 12 13 14 15 03 17 18 19 1a 1b 1c 1d 1e 1f
0111	10 11 12 13 14 15 16 03 18 19 1a 1b 1c 1d 1e 1f
1000	10 11 12 13 14 15 16 17 03 19 1a 1b 1c 1d 1e 1f
1001	10 11 12 13 14 15 16 17 18 03 1a 1b 1c 1d 1e 1f
1010	10 11 12 13 14 15 16 17 18 19 03 1b 1c 1d 1e 1f
1011	10 11 12 13 14 15 16 17 18 19 1a 03 1c 1d 1e 1f
1100	10 11 12 13 14 15 16 17 18 19 1a 1b 03 1d 1e 1f
1101	10 11 12 13 14 15 16 17 18 19 1a 1b 1c 03 1e 1f
1110	10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 03 1f
1111	10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 03

Table B-2. Halfword Insertion: Rightmost 4 Bits of the Effective Address and Created Mask

Rightmost 4 Bits of the Effective Address	Created Mask
000x	0203 1213 1415 1617 1819 1a1b 1c1d 1e1f
001x	1011 0203 1415 1617 1819 1a1b 1c1d 1e1f
010x	1011 1213 0203 1617 1819 1a1b 1c1d 1e1f
011x	1011 1213 1415 0203 1819 1a1b 1c1d 1e1f
100x	1011 1213 1415 1617 0203 1a1b 1c1d 1e1f
101x	1011 1213 1415 1617 1819 0203 1c1d 1e1f
110x	1011 1213 1415 1617 1819 1a1b 0203 1e1f
111x	1011 1213 1415 1617 1819 1a1b 1c1d 0203

Table B-3. Word Insertion: Rightmost 4 Bits of the Effective Address and Created Mask

Rightmost 4 Bits of the Effective Address	Created Mask
00xx	00010203 14151617 18191a1b 1c1d1e1f
01xx	10111213 00010203 18191a1b 1c1d1e1f
10xx	10111213 14151617 00010203 1c1d1e1f
11xx	10111213 14151617 18191a1b 00010203

Table B-4. Doubleword Insertion: Rightmost 4 Bits of Effective Address and Created Mask

Rightmost 4 Bits of the Effective Address	Created Mask
0xxx	0001020304050607 18191a1b1c1d1e1f
1xxx	1011121303151617 0001020304050607

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	See Cell Broadband Engine Architecture.
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.
PPE	PowerPC Processor Element. A general-purpose processor in the Cell Broadband Engine.
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

ROL	relative offset low
RTL	register transfer language
shufb	shuffle bytes instruction
signal	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.
SIMD	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.
SNaN	Signalling NaN.
snoop	To compare an address on a bus with a tag in a cache, to detect operations that violate memory coherency.
SPR	Special-purpose register.
SPU	Synergistic Processor Unit. The part of an SPE that executes instructions from its local storage (LS).
SRAM	static random access memory
sync	Synchronize command.
synchronization	The process of arranging storage operations to complete in the order of occurrence.
UNF	Underflow
vector	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.
word	Four bytes.
wrch	Write to channel instruction.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H, I, J, K

L

M

N

S

T

U, V, W

X

Z

Documentation Questionnaire

Your comments and suggestions help us improve the documentation. We are interested in your feedback even if you are a casual user of this document. Click on the Attachments tab in this PDF file and double-click on the file named CBE_quest_auto_spuisa.pdf to open it. You can complete the form online and return it by e-mail.

Thank you for taking the time to answer this short questionnaire.

.