

SPARC64™ X / X+ Specification

Distribution: Public
Privilege Levels: Nonprivileged

Ver 29.0
2015/01/27

Fujitsu Limited

Fujitsu Limited
4-1-1 Kamikodanaka
Nakahara-ku, Kawasaki, 211-8588
Japan

Copyright© 2007 - 2015 Fujitsu Limited, 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan. All rights reserved.

This product and related documentation are protected by copyright and distributed under licenses restricting their use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Fujitsu Limited and its licensors, if any.

The product(s) described in this book may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

SPARC® is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Oracle and / or its affiliates.

SPARC64™ is a registered trademark of SPARC International, Inc., licensed exclusively to Fujitsu Limited.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Fujitsu and the Fujitsu logo are trademarks of Fujitsu Limited.

This publication is provided “as is” without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

This publication could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein; these changes will be incorporated in new editions of the publication. Fujitsu Limited may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

Index

1. Document Overview	8
1.1. Fonts and Notations	8
1.1.1. Font	8
1.1.2. Notation	8
1.1.3. Meaning of <i>reserved</i> and —	9
1.1.4. Access attributes	9
1.1.5. Informational Notes	9
2. Definitions	11
3. Architectural Overview	12
4. Data Formats	13
4.1. Densely Packed Decimal (DPD) Floating-Point Numbers	13
4.1.1. Field	13
4.1.2. Combination field (G)	14
4.1.3. Trailing significand field (T)	14
4.1.4. Cohort	15
4.1.5. Normal and denormal DPD floating-point numbers	15
4.1.6. Numbers that can be encoded by the DPD format	15
4.1.7. Rounding modes	16
4.2. Packed BCD (Binary Coded Decimal)	16
4.2.1. Fields	16
4.3. Oracle floating-point numbers	17
4.3.1. Fields	17
4.3.2. Sign (S)	18
4.3.3. Exponent (exp)	18
4.3.4. Mantissa (significand)	18
4.3.5. Special values	20
4.3.6. Normal and denormal numbers	20
4.3.7. Numbers that can be encoded as Oracle floating-point numbers	20
4.3.8. Rounding modes	21
4.3.9. Extended exponent part (exp10)	21
5. Register	23
5.1. Reserved Register Fields	23
5.2. General-Purpose R Registers	23
5.2.1. General-Purpose Integer Registers	23
5.2.2. Windowed R Registers	23
5.2.3. Special R Registers	23
5.3. Floating-Point Registers	23
5.3.1. Floating-Point Register Number Encoding	24
5.3.2. Using double-precision registers for single-precision operations	24
5.3.3. Specifying registers for SIMD instructions	25
5.4. Floating-Point State Register (FSR)	26
5.5. Ancillary State Registers	31
5.5.1. 32-bit Multiply/Divide Register (Y) (ASR 0)	31
5.5.2. Integer Condition Codes Register (CCR) (ASR 2)	31
5.5.3. Address Space Identifier (ASI) Register (ASR 3)	31
5.5.4. Tick (TICK) Register (ASR 4)	31
5.5.5. Program Counters (PC, NPC) (ASR 5)	32
5.5.6. Floating-Point Registers State (FPRS) Register (ASR 6)	32
5.5.7. Performance Control Register (PCR) (ASR 16)	32

5.5.8.	Performance Instrumentation Counter (PIC) Register (ASR 17)	33
5.5.9.	General Status Register (GSR) (ASR 19)	34
5.5.11.	System Tick (STICK) Register (ASR 24)	34
5.5.13.	Pause Register (PAUSE) (ASR 27)	35
5.5.14.	Extended Arithmetic Register (XAR) (ASR 29)	35
5.5.15.	Extended Arithmetic Register Status Register (XASR) (ASR 30)	37
6.	Instruction Set Overview	40
6.1.	Instruction Execution	40
6.2.	Instruction Formats	40
6.3.	Instruction Categories	40
6.3.9	Floating-Point Operate (FPop) Instructions	40
6.3.11	Reserved Opcodes and Instruction Fields	41
7.	Instructions	42
7.1.	ADD	51
7.2.	Align Address	52
7.4.	Three-Dimensional Array Addressing	53
7.5.	Byte Mask and Shuffle	54
7.6.	Branch on Integer Condition Codes (Bicc)	55
7.7.	Branch on Integer Condition Codes with Prediction (BPcc)	56
7.8.	Branch on Integer Register with Prediction (BPr)	57
7.9.	Call and Link	58
7.10.	Compare and Swap	59
7.12.	Edge Handling Instructions	61
7.13.	Edge Handling Instructions (noCC)	62
7.14.	Convert Integer to Floating-Point	63
7.15.	Convert Between Floating-Point Formats	64
7.16.	Convert Floating-Point to Integer	66
7.17.	Floating-Point Absolute Value	67
7.18.	Floating-Point Add and Subtract	68
7.19.	Align Data	70
7.20.	Branch on Floating-Point Condition Codes (FBfcc)	71
7.21.	Branch on Floating-Point Condition Code with Prediction (FBPfcc)	72
7.22.	Floating-Point Compare	73
7.23.	Floating-Point Conditional Compare to Register	74
7.24.	SIMD Compare (comforms to UA2011)	76
7.25.	Floating-Point Divide	77
7.26.	Floating-Point Exponential Auxiliary	78
7.27.	FEXPAND	80
7.28.	Flush Instruction Memory	81
7.29.	Flush Register Windows	82
7.30.	Floating-Point Multiply-Add/Subtract	83
7.31.	Floating-Point Minimum and Maximum	91
7.32.	Floating-Point Move	93
7.33.	Move Floating-Point Register on Condition (FMOVcc)	94
7.34.	Move Floating-Point Register on Integer Register Condition (FMOVr)	95
7.35.	Partitioned Multiply Instructions	96
7.36.	Floating-Point Multiply	97
7.37.	Floating-Point Negative	98
7.38.	FPACK	99
7.39.	Fixed-point Partitioned Add	100
7.40.	Integer Multiply-Add	102
7.41.	FPMERGE	103
7.42.	Fixed-point Partitioned Subtract (64-bit)	104
7.43.	F Register Logical Operate	106
7.44.	Floating-Point Reciprocal Approximation	109
7.45.	Move Selected Floating-Point Register on Floating-Point Register's Condition	112
7.46.	Floating-Point Square Root	113
7.47.	Floating-Point Trigonometric Functions	114
7.48.	Illegal Instruction Trap	119
7.49.	Integer Logical Operation	120
7.51.	Jump and Link	121
7.52.	Load Integer	122
7.53.	Load Integer from Alternate Space	123
7.54.	Block Load	124

7.55.	Load Floating-Point	126
7.56.	Load Floating-Point from Alternate Space	129
7.57.	Short Floating-Point Load	132
7.58.	Load-Store Unsigned Byte	133
7.59.	Load-Store Unsigned Byte to Alternate Space	134
7.60.	Load Integer Twin Word	135
7.61.	Load Integer Twin Word from Alternate Space	136
7.62.	Load Integer Twin Extended Word from Alternate Space	138
7.63.	Load Floating-Point State Register	140
7.64.	Memory Barrier	141
7.65.	Move Integer Register on Condition (MOVcc)	142
7.66.	Move Integer Register on Register Condition (MOVr)	143
7.67.	Multiply Step	144
7.68.	Multiply and Divide (64-bit)	146
7.69.	No Operation	147
7.72.	Partitioned Add	148
7.73.	Pixel Component Distance (with Accumulation)	149
7.74.	Population Count	150
7.75.	Prefetch	151
	7.75.1. Prefetch Variants	151
	7.75.2. Weak versus Strong Prefetches	152
7.76.	Read Ancillary State Register (RDASR)	154
7.79.	Return	155
7.80.	SAVE and RESTORE	156
7.82.	Signed Divide (64-bit÷32-bit)	157
7.83.	SETHI	158
7.85.	Set Interval Arithmetic Mode	159
7.87.	Shift	160
7.88.	Signed Multiply (32-bit)	161
7.89.	Sleep	162
7.91.	Store Barrier	163
7.92.	Store Integer	164
7.93.	Store Integer into Alternate Space	165
7.94.	Block Initializing Store	166
7.95.	Block Store	167
7.96.	Store Floating-Point	169
7.97.	Store Floating-Point into Alternate Space	171
7.98.	Store Floating-Point Register on Register Condition (for SPARC64™ X)	174
7.99.	Store Partial Floating-Point	177
7.100.	Store Short Floating-Point	178
7.101.	Store Integer Twin Word	179
7.102.	Store Integer Twin Word into Alternate Space	180
7.103.	Store Floating-Point State Register	181
7.104.	Subtract	182
7.105.	Swap Register with Memory	183
7.106.	Set XAR (SXAR)	184
7.107.	Tagged Add and Subtract	185
7.108.	Trap on Integer Condition Code (Tcc)	186
7.109.	Unsigned Divide (64-bit÷32-bit)	187
7.110.	Unsigned Multiply (32-bit)	188
7.111.	Write Ancillary State Register (WRASR)	189
7.114.	Cache Line Fill with Undetermined Values	190
7.115.	DES support instructions	192
7.116.	AES support instructions	197
7.117.	Decimal Floating-Point Operations	204
7.118.	Oracle Floating-Point Operations	209
7.119.	Decimal Floating-Point Compare	217
7.120.	Oracle Decimal Floating-Point Compare	219
7.121.	Decimal Floating-Point Convert	220
7.122.	Shift Mask Or (for SPARC64™ X)	222
7.123.	SIMD Compare (for SPARC64™ X)	226
7.124.	Leading Zero Detect	230
7.125.	Fixed-point Partitioned Add (64-bit)	231
7.126.	Fixed-point Partitioned Subtract (64-bit)	232
7.127.	SIMD Unsigned Compare	233
7.128.	Floating-Point Lexicographic Compare	234

7.129. Floating-Point Negative Add.....	235
7.130. Floating-Point Negative Multiply	237
7.131. WRPAUSE(PAUSE).....	239
7.132. Load Entire Floating-Point State Register.....	241
7.133. Compare and Branch (CBcond)	242
7.134. Partitioned Move Selected Floating-Point Register on Floating-Point Register's Condition	243
7.135. 64-bit Integer Compare on Floaing-Point Register	246
7.136. 64-bit Integer Shift on Floating-Point Register	247
7.137. Store Floating-Point Register on Register Condition (Extension of SPARC64™ X+)	248
7.138. Shift Mask Or (Extension of SPARC64™ X+).....	252
7.139. SIMD Compare (Extension of SPARC64™ X+)	257
7.140. Fixed-Point Partitioned Add (128-bit).....	261
7.141. Integer Minimum and Maximum	262
7.142. Move Integer Register to Floating-Point Register (for SPARC64™ X+)	264
8. IEEE Std. 754-1985 Requirements for SPARC-V9.....	265
8.1. Nonstandard Floating-Point Mode	265
8.1.1. <i>fp_exception_other</i> (ftt = <i>unfinished_FPop</i>)	265
8.1.2. Behavior when FSR.ns = 1	268
9. Memory Models	272
10. Address Space Identifiers	273
10.3. ASI Assignment	273
10.3.1. Supported ASIs.....	273
10.3.2. ASI access exceptions	275
11. Performance Instrumentation.....	277
11.1. Overview.....	277
11.1.1. Sample Pseudo-codes.....	278
11.2. Description of PA Events	280
11.2.1. Instruction and Trap Statistics	286
11.2.2. MMU and L1 cache Events.....	292
11.2.3. L2 cache Events	293
11.2.4. Bus Transaction Events	295
11.3. Cycle Accounting.....	296
12. Traps.....	298
12.1. Virtual Processor Privilege Modes	298
12.5. Trap list and priorities	298
12.5.1. Trap Descriptions.....	302
12.5.2. Special cases for priority	310
13. Memory Management Unit	311
13.1. Address types	311
13.4. TSB Translation Table Entry (TTE).....	311
13.6. Context Registers.....	312
13.8. Page sizes	313
14. Opcode Maps	314
15. Assembly Language Syntax	323
15.1. Notation Used	323
15.1.1. Other Operand Syntax	323
15.2. HPC-ACE Notation.....	323
15.2.1. Suffixes for HPC-ACE Extensions.....	324

Preface

This document defines the logical specification of SPARC64™ X / SPARC64™ X+ and is based on Oracle SPARC Architecture 2011(UA2011). Differences from UA2011 are noted in this document or as references to other documents.

This specification refers to the following documents:

- Oracle SPARC Architecture 2011. Draft D0.9.6, May 2014.
<http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf>
We refer to this document as UA2011.
- SPARC64 VIIIfx Extensions. Ver 15, 26 Apr. 2010.
<http://img.jp.fujitsu.com/downloads/jp/jhpc/sparc64viii-fx-extensions.pdf>
We refer to this document as SPARC64 VIIIfx Extensions.
- SPARC® Joint Programming Specification (JPS1): Commonality
Release 1.0.4, 31 May 2002.
<http://www.fujitsu.com/downloads/PRMPWR/JPS1-R1.0.4-Common-pub.pdf>
We refer to this document as JPS1.

1. Document Overview

1.1. Fonts and Notations

1.1.1. Font

- Arial font is used for registers and register fields (`REG` and `REG.field`, respectively). This font is also used when referring to the field of an ASI register.
- Courier font is used for ASI names (`ASI_NAME`), which are prefixed by `ASI_`. We avoid the use of the construction `ASI_NAME.field`.
- Italic Arial font is used for exceptions (*exception_name*).
- Uppercase Courier font is used for instructions (`INSTRUCTION`).
- Courier font is used for CPU states (`CPU_state`).
- Italic Times Roman font or “—” is used for reserved, which indicates that a register field is reserved for future expansion.

1.1.2. Notation

The notation used in this document generally follows the notation used in JPS1.

Specifically,

- Numbers are decimal unless otherwise indicated by a numeric subscript (for example, `10002`).
- Spaces may be inserted in long binary or hex numbers (for example, `1000 000016`) to improve readability.
- Verilog notation may be used for some numbers. For example, the prefixes “`{bit_width}'B`” and “`{bit_width}'h`” indicate binary and hexadecimal numbers, respectively. When Verilog notation is used, there is no numeric subscript indicating the base.
- Numbered integer and floating-point registers are written as `R[number]` and `F[number]`, respectively.
- Instruction names and various objects may contain the symbols `{}`, `|`, `*` and `n`.
 - A character string enclosed by `{}` is optional. For example, `ASI_PRIMARY{_LITTLE}` expands to `ASI_PRIMARY` and `ASI_PRIMARY_LITTLE`.
 - If there are `|` symbols inside the curly braces `{}`, one of the character strings separated by the vertical bars must be selected. For example, `FMUL{s|d}` expands to `FMULs` and `FMULd`. An empty character string makes the alternatives inside the braces optional. For example, `F{N}SMULd` is equivalent to `F{N}SMULd`.
 - The `*` and `n` symbols indicate character string and numeric substitution, respectively, for all possible values. For example, `DAE *` expands to `DAE_invalid_asi`, `DAE_nc_page`, `DAE_nfo_page`, `DAE_privilege_violation`, and `DAE_side_effect_page`. `spill_n_normal` expands to `spill_0_normal`, `spill_1_normal`,

spill_2_normal, *spill_3_normal*, *spill_4_normal*, *spill_5_normal*, *spill_6_normal*, and *spill_7_normal*.

- Bit strings are of the form <a> and <a:b>.
- The double colon (::) operator concatenates two bit strings.
- ASCII characters are used.

1.1.3. Meaning of *reserved* and —

reserved or — indicates that a bit field is reserved for future expansion and has an undefined value. *reserved* is used when future expansion is expected; a brief description of the field is provided. — is used when the usage is undecided. No description is provided for fields marked with —.

1.1.4. Access attributes

Registers and register fields may have the access attributes shown in the table below.

Table 1-1 Access attributes

Access attribute	Object	Operation	
		Read	Write
	Field	Undefined value	Ignored.
R	Register and Field	The value is read.	Ignored.
RO	Register and Field	The value is read.	Not permitted.
R0	Field	Zero is read.	Ignored.
W	Register and Field	Undefined value	The value is written.
WO	Register and Field	Not permitted.	The value is written.
RW	Register and Field	The value is read.	The value is written.
RW1C	Field	The value is read.	Writing 1 clears the field. ⁱ
RWQF	Field	The value is read.	Register value indicates condition, and a write of the same value is preserved in the register. Otherwise, the write is ignored.
RWS/R0WS	Register	The value is read. Or, zero is read.	Causes side effect. Value to be written is ignored.

1.1.5. Informational Notes

This document contains several different types of information notes.

Compatibility Note Compatibility notes explain compatibility differences versus SPARC V8/V9, JPS1, SPARC64 VIIIfx Extensions, and UA2011.

Note Notes provide general information.

ⁱ The bit range that is reset to 0 depends on the field.

Programming Note	Programming notes provide information for writing software.
-------------------------	---

2. Definitions

For additional definitions, please refer to Chapter 2 of UA2011.

CPUID :

A CPUID is the unique logical ID of a strand in a system. The CPUID contains the LSBID (logical system board ID) and Chip ID (physical processor ID within a system board).

LSB (Logical System Board) :

A physical partition is a set of one or more system boards that work together as a single system. An LSB is a system board in a physical partition and is identified by the allocated logical ID.

VCPU (Virtual Processor):

A virtual processor (refer to Chapter 2 of UA2011). SPARC64™ X and SPARC64™ X+ have two VCPUs per physical CPU core.

3. Architectural Overview

Feature

- HPC-ACE and SMT are supported. VA is 64-bits wide and has no hole bit.
- RA is normally 64 bits wide.
- Instructions only on local ROM can be executed for non-cacheable space.
- NWINDOWS = 8
- MAXPTL = 2

Present parameter

- 16 cores/chip and 2SMT/core
- L1 instruction cache : 64KB/4way ; L1 data cache : 64KB/4way ; Unified L2 cache : 24MB/24way ; line size of all cache memories : 128 bytes
- For main TLB, set-associative TLB only, instruction : 1024entries/4way ; data : 1024entries/4way ; page size : 4 sizes (8KB, 64KB, 4MB, 256MB)

4. Data Formats

Refer to UA2011 for Integer Data Formats, Floating-Point Data Formats, and VIS instruction set SIMD Data Formats.

Refer to 5.3 Floating-Point Registers in this document for the HPC-ACE SIMD Data Format.

4.1. Densely Packed Decimal (DPD) Floating-Point Numbers

SPARC64™ X / SPARC64™ X+ support decimal floating-point numbers encoded with the DPD (Densely Packed Decimal) format defined by IEEE754-2008 and instructions that operate on DPD floating-point numbers.

4.1.1. Field

The value of a DPD floating-point number is given by the following expression, where S, the exponent, and the significand are integers. S has a value of 0 or 1.

$$(-1)^S \times \text{significand} \times 10^{\text{exponent}}$$

A DPD floating-point number is encoded by a sign field S, a combination field G, and a trailing significand field T. The exponent is stored in the combination field G, and the significand is split between G and the trailing significand field T. The combination field G has additional structure, with two bit ranges that are 5-bits wide and W bits wide.

SPARC64™ X / SPARC64™ X+ support the DPD format for both single-precision and double-precision floating-point numbers.

Table 4-1 DPD format field widths

	Single precision	Double precision
Entire data	32 bits	64 bits
S	1 bit	1 bit
G (W + 5)	11 bits	13 bits
W	6 bits	8 bits
T	20 bits	50 bits

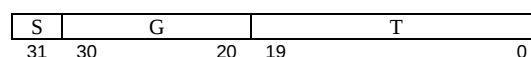


Figure 4-1 DPD floating-point single-precision data format

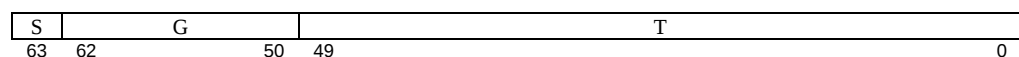


Figure 4-2 DPD floating point double-precision data format

4.1.2. Combination field (G)

The combination field G is divided into the upper five bits (GU<4:0>) and the remaining lower bits (GL<(W-1):0>). GU contains the two uppermost bits of the exponent and the most significant digit of the significand. The most significant digit of the significand is the leftmost digit (LMD). GU also indicates if the data is not a number (NaN, ∞). Table 4-2 shows the encoding of the GU bit range.

Table 4-2 Encoding of upper five bits in the combination field (GU)

GU<4:0>	Upper two bits of exponent part	Leftmost digit of significand (LMD)	Remarks
11111 ₂	—	—	SNaN if GL<W-1> = 1. QNaN if GL<W-1> = 0. GL<(W-2):0> is ignored. T is the payload.
11110 ₂	—	—	$+\infty$ or $-\infty$ GL<(W-1):0> is ignored.
1110x ₂ 110xx ₂	GU<2:1> (00 ₂ , 01 ₂ , 10 ₂)	8 + GU<0> (8 or 9)	GL<(W-1):0> are the lower bits of the exponent.
10xxx ₂ 0xxxx ₂	GU<4:3> (00 ₂ , 01 ₂ , 10 ₂)	$4 \times \text{GU}<2> + 2 \times \text{GU}<1> + \text{GU}<0>$ (0 – 7)	GL<(W-1):0> are the lower bits of the exponent.

When GU is 00000₂, 01000₂ or 10000₂, the significand is zero. Therefore, +0 is expressed as S = 0, T = 0, and GU = 00000₂, 01000₂, or 10000₂. -0 is expressed as S = 1, T = 0, and GU = 00000₂, 01000₂ or 10000₂. Note that there are different representations of zero due to the three possible values of GU<4:3> and the bit width of GL, which depends on the precision of the data. (The set of possible encodings is called a cohort.)

4.1.3. Trailing significand field (T)

There are two or more sets of ten bits (referred to as declet) in the trailing significand field T. Each declet encodes a number from 0 to 999. The following tables show how to convert between declets and three-digit decimal numbers. In these tables, a declet is shown as b<9:0>. The decimal number is divided into three parts, which are weighted by 100, 10, and 1, respectively. Each part is four bits. The hundreds part is shown as h<3:0>. The tens part is shown as t<3:0>. The ones part is shown as o<3:0>. Bit i in any of these ranges (b, h, t, and o) is written by concatenating the name of the bit range and the bit number. For example, b9 is equivalent to b<9>.

Table 4-3 Converting from declets (b<9:0>) to three-digit decimal numbers ($100 \times h + 10 \times t + o$)

b<9:0>	Humdreds (h3h2h1h0)	Tens (t3t2t1t0)	Ones (o3o2o1o0)
xxxxxx0xxx	$4 \times b9 + 2 \times b8 + b7$	$4 \times b6 + 2 \times b5 + b4$	$4 \times b2 + 2 \times b1 + b0$
xxxxxx100x	$4 \times b9 + 2 \times b8 + b7$	$4 \times b6 + 2 \times b5 + b4$	8+b0
xxxxxx101x	$4 \times b9 + 2 \times b8 + b7$	8+b4	$4 \times b6 + 2 \times b5 + b0$
xxxxxx110x	8+b7	$4 \times b6 + 2 \times b5 + b4$	$4 \times b9 + 2 \times b8 + b0$
xxx00x111x	8+b7	8+b4	$4 \times b9 + 2 \times b8 + b0$
xxx01x111x	8+b7	$4 \times b9 + 2 \times b8 + b4$	8+b0
xxx10x111x	$4 \times b9 + 2 \times b8 + b7$	8+b4	8+b0
xxx11x111x	8+b7	8+b4	8+b0

Table 4-4 Converting from three-digit decimal numbers ($100 \times h + 10 \times t + o$) to declets ($b\langle 9:0 \rangle$)

Hundreds h3 h2 h1 h0	Tens t3 t2 t1 t0	Ones o3 o2 o1 o0	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0xxx	0xxx	0xxx	h2	h1	h0	t2	t1	t0	0	o2	o1	o0
0xxx	0xxx	100x	h2	h1	h0	t2	t1	t0	1	0	0	o0
0xxx	100x	0xxx	h2	h1	h0	o2	o1	t0	1	0	1	o0
0xxx	100x	100x	h2	h1	h0	1	0	t0	1	1	1	o0
100x	0xxx	0xxx	o2	o1	h0	t2	t1	t0	1	1	0	o0
100x	0xxx	100x	t2	t1	h0	0	1	t0	1	1	1	o0
100x	100x	0xxx	o2	o1	h0	0	0	t0	1	1	1	o0
100x	100x	100x	x	x	h0	1	1	t0	1	1	1	o0

Note that a declet $b\langle 9:0 \rangle$ can encode 1024 numbers, while a three-digit decimal number only encodes 1000 numbers. In other words, some decimal numbers convert to more than one declet of equivalent value. The bottom row of Table 4-4 shows the cases where $b\langle 9:8 \rangle$ may assume different values for the same decimal number.

The following table explicitly shows the decimal numbers with multiple declets.

$b\langle 9:0 \rangle$	Decimal number	$b\langle 9:0 \rangle$	Decimal number
06E ₁₆ , 16E ₁₆ , 26E ₁₆ , 36E ₁₆	888	0EE ₁₆ , 1EE ₁₆ , 2EE ₁₆ , 3EE ₁₆	988
06F ₁₆ , 16F ₁₆ , 26F ₁₆ , 36F ₁₆	889	0EF ₁₆ , 1EF ₁₆ , 2EF ₁₆ , 3EF ₁₆	989
07E ₁₆ , 17E ₁₆ , 27E ₁₆ , 37E ₁₆	898	0FE ₁₆ , 1FE ₁₆ , 2FE ₁₆ , 3FE ₁₆	998
07F ₁₆ , 17F ₁₆ , 27F ₁₆ , 37F ₁₆	899	0FF ₁₆ , 1FF ₁₆ , 2FF ₁₆ , 3FF ₁₆	999

4.1.4. Cohort

The encoding of a real number in the DPD floating-point format is not unique. For example, $1.000 \times 10^2 = 10.00 \times 10^1 = 100.0 \times 10^0$. The set of equivalent encodings is called a cohort. The number of cohort members for a DPD floating-point number depends on the value. For example, 1.000×10^2 has 7 cohort members in single precision and 16 members in double precision. As discussed in section 4.1.2, zero has multiple encodings due to the 3 possible values of GU and the width of GL. Note that +0 and -0 are different numbers and are not part of the same cohort.

When comparing DPD floating-point numbers, some members of a cohort are considered equivalent.

4.1.5. Normal and denormal DPD floating-point numbers

A DPD floating-point number is normal if there is a cohort member with LMD greater than 0.

A number is denormal if the cohort member with the smallest exponent has an LMD of 0.

+/-0 and +/-∞ are neither normal nor denormal numbers.

4.1.6. Numbers that can be encoded by the DPD format

Table 4-5 shows the numbers that can be expressed as DPD floating-point numbers.

Table 4-5 Range of DPD floating-point numbers

	Single precision	Double precision
Number of significand digits	7	16
Exponent	−101–90 (bias = 101)	−398–369 (bias = 398)
Normal numbers		
Maximum absolute value (Nmax)	$(10^7-1) \times 10^{90}$	$(10^{16}-1) \times 10^{369}$
Minimum absolute value (Nmin)	1×10^{-95}	1×10^{-383}
Denormal numbers		
Maximum absolute value (Dmax)	$(106-1) \times 10^{101}$	$(10^{15}-1) \times 10^{398}$
Minimum absolute value (Dmin)	1×10^{101}	1×10^{398}
0	S Sign Exponent Any Significand 0	S Sign Exponent Any Significand 0
Infinity	S Sign GU 11110 ₂ GL Ignored T Any (payload)	S Sign GU 11110 ₂ GL Ignored T Any (payload)
NaN		
SNaN	S Ignored GU 11111 ₂ GL 1xxxxx ₂ T Any (payload)	S Ignored GU 11111 ₂ GL 1xxxxxxx ₂ T Any (payload)
QNaN	S Ignored GU 11111 ₂ GL 0xxxxx ₂ T Any (payload)	S Ignored GU 11111 ₂ GL 0xxxxxxx ₂ T Any (payload)

4.1.7. Rounding modes

There are five DPD rounding modes, which conform to IEEE754-2008. The rounding mode is specified by the value of FSR.drd (page 26) or GSR.dirnd (page 34).

- Nearest (even, if tie)
- Round toward 0
- Round toward $+\infty$
- Round toward $-\infty$
- Nearest (away from 0, if tie)

4.2. Packed BCD (Binary Coded Decimal)

SPARC64™ X / SPARC64™ X+ support instructions to convert between a BCD number and the equivalent DPD floating-point number. No instructions that operate directly on BCD data are defined. The BCD data may be signed or unsigned.

4.2.1. Fields

D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
63 60	59 56	55 52	51 48	47 44	43 40	39 36	35 32	31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0

Figure 4-3 Unsigned BCD data format

D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	S
63 60	59 56	55 52	51 48	47 44	43 40	39 36	35 32	31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0

Figure 4-4 Signed BCD data format

A number 0-9 is encoded by four bits in the BCD number. Each field "D" in Figure 4-3 and Figure 4-4 is one decimal digit. The signed BCD data format encodes 15 decimal digits, and the unsigned BCD data format encodes 16 decimal digits. If any field "D" has a value of A₁₆ – F₁₆, the data is not a BCD number.

In a signed BCD number, the least significant four bits S encode the sign of the number. Table 4-6 shows the relationship between the value of S and the sign. Note that SPARC64™ X / SPARC64™ X+ encode the plus sign as C₁₆ and the minus sign as D₁₆.

Table 4-6 Sign of BCD data

S	Sign	S	Sign
0 ₁₆	+	8 ₁₆	+
1 ₁₆	+	9 ₁₆	+
2 ₁₆	+	A ₁₆	+
3 ₁₆	+	B ₁₆	–
4 ₁₆	+	C ₁₆	+
5 ₁₆	+	D ₁₆	–
6 ₁₆	+	E ₁₆	+
7 ₁₆	+	F ₁₆	+

4.3. Oracle floating-point numbers

Compatibility Note The specification for Oracle floating-point numbers may change in the future. This format should be used only for libraries targeting the SPARC64™ X and SPARC64™ X+ platforms.

SPARC64™ X / SPARC64™ X+ support Oracle floating-point numbers and instructions that operate on these numbers.

4.3.1. Fields

An Oracle floating-point number consists of a sign S, an exponent exp, and a significand. S is an integer with value 0 or 1, exp is an integer with value -65 to 62, and the significand is a number with a fixed number of digits and value 0 to 99.999999999999. The value is given by the following expression.

$$(-1)^{(1-S)} \times \textit{significand} \times 100^{\textit{exp}}$$

Table 4-7 Oracle floating-point fields widths

	Oracle floating-point number
Entire data	64 bits
S	1 bit
exp	7 bits
Significand	56 bits

S	exp	Significand
63	62	56 55 0

Figure 4-5 Oracle floating-point data format

4.3.2. Sign (S)

The field S encodes the sign. Table 4-8 shows the possible values of sign S.

Table 4-8 Sign encoding

S	Sign
0	Negative
1	Positive

4.3.3. Exponent (exp)

The exponent is stored in the field exp, which has a value in the range -65 to 62 . The maximum exponent encoded by exp is infinity. The encoding of the exponent depends on the sign S. The field exp is encoded in ascending order if the sign is positive and in descending order if the sign is negative. The calculation of the exponent from integer exp<6:0> is shown in Table 4-9.

Table 4-9 Exponent calculation

S	Exponent
0 (negative)	$62 - \text{exp}\langle 6:0 \rangle$
1 (positive)	$\text{exp}\langle 6:0 \rangle - 65$

Table 4-10 shows the explicit encoding of the exponent.

Table 4-10 Exponent encoding

S	exp<6:0>	Exponent
0 (negative)	0	62 (or, $-\infty$)
	1	61
	..	
	62	0
	..	
	126	-64
	127	-65
1 (positive)	0	-65
	1	-64
	..	
	65	0
	..	
	126	61
	127	62 (or, $+\infty$)

4.3.4. Mantissa (significand)

The significant consists of seven bytes. Each byte encodes an integer 0–99. The value of the significand in relation to these seven integers (digits) is given by the following expression.

$$\sum_{i=0}^6 digit_i \times 100^{-i}$$

digit ₀	digit ₁	digit ₂	digit ₃	digit ₄	digit ₅	digit ₆
55	48 47	40 39	32 31	24 23	16 15	8 7 0

Figure 4-6 Significand format

The encoding of these bytes depends on the sign S. The digits of a positive value are encoded in ascending order. The digits of a negative value are encoded in descending order. Each integer is expressed as digit<7:0> and is calculated as shown in Table 4-11

Table 4-11 Integer Calculation

S	Integer
0 (negative)	101·digit<7:0>
1 (positive)	digit<7:0>·1

Table 4-12 shows the explicit encoding of each digit.

Table 4-12 Encoding of digit<7:0 >

S	digit<7:0>	Number
0 (negative)	0 – 1	Outside the range (treated as 0)
	2	99
	3	98
	4	97
	..	
	99	2
	100	1
	101	0
	102 – 255	Outside the range (treated as 0)
1 (positive)	0	Outside the range (treated as 0)
	1	0
	2	1
	3	2
	.	
	98	97
	99	98
	100	99
	101 – 255	Outside the range (treated as 0)

Note When a number with all digits outside the range is specified as an operand, it is generally treated as 0. In certain combinations where a digit has the value 0 or 101, the number may be treated as infinity. See Section 4.3.5.

4.3.5. Special values

The values 0, $-\infty$, $+\infty$ and dNAN are special Oracle floating-point numbers. These values are expressed as a combination of several fields. The combination of these fields is shown below.

When an operation results in the special value 0, $-\infty$, $+\infty$, or dNAN, the resulting Oracle floating-point number has the format shown in Table 4-13.

Table 4-13 Output of special values

Special value	S	exp	digit ₀	digit ₁₋₆	Remarks
0	1 (positive)	0	0	0	Negative 0 is never an output.
$-\infty$	0 (negative)	0	0	0	
$+\infty$	1 (positive)	127	101	0	
dNAN	1 (positive)	0	0	0	Same as 0. It is not possible to distinguish between dNAN and 0 from the result.

To use a special value as the operand of an operation, specify the Oracle floating-point number as described in Table 4-14. To input $-\infty$ or $+\infty$, specify S, exp, and digit₀ as described in Table 4-14. To input 0, specify all digits with values treated as 0. The value dNAN cannot be specified.

Table 4-14 Input of special values

Special value	S	exp	digit ₀	digit ₁₋₆	Remarks
0	—	—	*0*	*0*	*0* is a value 0 – 1 or 101 – 255.
$-\infty$	0 (negative)	0	0	—	Negative digit 0 is outside the range, as shown in Table 4-12.
$+\infty$	1 (positive)	127	101	—	Positive digit 101 is outside the range, as shown in Table 4-12.
dNAN	—	—	—	—	Cannot be specified.

4.3.6. Normal and denormal numbers

An Oracle floating-point number is normal if the value of digit₀ of the significand is larger than 0. The number is denormal if the value of digit₀ of the significand is 0.

The Oracle floating-point numbers 0 and $\pm\infty$ are neither normal nor denormal.

Note In SPARC64™ X / SPARC64™ X+, most operations with Oracle floating-point numbers always output a normal number or one of the special values defined in Section 4.3.5. Operations can be denormal. A few instructions output the result without normalizing the value. Refer to the specification of each instruction for details.

4.3.7. Numbers that can be encoded as Oracle floating-point numbers

Table 4-15 shows the numbers that can be expressed as Oracle floating-point numbers.

Table 4-15 Range of Oracle floating-point numbers

	Oracle floating-point number
Number of significand digits (decimal number)	7 (14)
Exponent	-65 – 62 (bias = 65)
Normal numbers	
Maximum absolute value (Nmax)	$99.999999999999 \times 100^{62}$
Minimum absolute value (Nmin)	1×100^{-65}
Denormal numbers	
Maximum absolute value (Dmax)	$0.999999999999 \times 100^{62}$
Minimum absolute value (Dmin)	1×100^{71}
Special values	
0	Refer to Section 4.3.5.
$-\infty$	
$+\infty$	
dNaN	

4.3.8. Rounding modes

There are five rounding modes for Oracle floating-point numbers. The rounding mode is specified by the value of FSR.drd (page 26) or GSR.irnd (page 34).

- Nearest (even if tie)
- Round toward 0
- Round toward $+\infty$
- Round toward $-\infty$
- Nearest (away from 0, if tie)

Note The least significant bit (LSB) of the significand is rounded after normalization. The LSB is the rightmost bit. When rounding, normalization ignores the limits imposed by the encoding of the exponent. If the rounded result is denormal, 0 is returned.

Example) $0.200000000001e^{-65}$

- 1) Normalizing this value gives the number $20.000000000100 e^{-66}$. (Remember that the exponent is a power of 100.) Note that the minimum possible exponent that can be encoded is -65.
 - 2) Then the LSB of the normalized number is 0 and rounding does not change the value. After forcing the rounded number to satisfy the minimum value of the exponent, we recover the original number, whose most significant bit is 0. The result is denormal and 0 is returned.
-

Note Underflow is decided after rounding for Oracle floating-point numbers, unlike IEEE 754, where underflow is decided before rounding.

4.3.9. Extended exponent part (exp10)

Certain instructions take an extended exponent as an operand. The extended exponent specifies the exponent as a power of 10 instead of 100 (see the formula in Section 4.3.1). It

consists of sign S and the extended exponent exp10. The extended exponent is used only to specify the exponent and has no meaning as a numerical value.

Table 4-16 Extended exponent field widths

	Extended exponent
Entire data	64 bits
S	1 bit
exp10	8 bits
reserved	55 bits

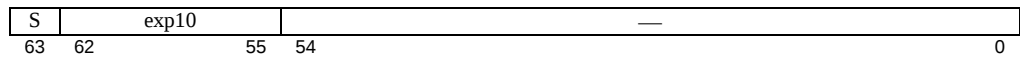


Figure 4-7 Extended exponent format

The encoding of the extended exponent depends on the sign S. The extended exponent exp10 is encoded in ascending order if positive and in descending order if negative. The extended exponent is expressed by the integer exp10<7:0> and has the value indicated in Table 4-17.

Table 4-17 Calculation of exponent radix 10

S	Exponent radix 10
0 (negative)	$125 - \text{exp10}<7:0>$
1 (positive)	$\text{exp10}<7:0> - 130$

Explicit encodings for the extended exponent exp10 are shown in Table 4-18.

Table 4-18 Encoding of extended exponent exp10<7:0 >

S	exp10<7:0>	Exponent radix 10
0 (negative)	0	125
	1	124
	..	
	125	0
	..	
	254	-129
	255	-130
1 (positive)	0	-130
	1	-129
	..	
	130	0
	..	
	254	124
	255	125

5. Register

5.1. Reserved Register Fields

Refer to Section 5.1 in UA2011.

Compatibility Note To preserve compatibility with previous platforms, some *reserved* fields will read as 0.

Programming Note When comparing values, *reserved* fields should be masked and excluded from the comparison.

5.2. General-Purpose R Registers

5.2.1. General-Purpose Integer Registers

Refer to Section 5.2.1 in UA2011.

Global registers are referred to as R[0] – R[7] or g[0] – g[7] in this specification. There is no notation for indicating which set of global registers is currently selected by the GL register.

5.2.2. Windowed R Registers

Refer to Section 5.2.2 in UA2011.

The number of windowed register sets, N_REG_WINDOWS, is 8.

5.2.3. Special R Registers

Refer to Section 5.2.3 in UA2011.

5.3. Floating-Point Registers

In addition to the floating-point registers defined in Section 5.3 Floating-Point Registers of UA2011, new double-precision floating-point registers Fd[64] – Fd[126] and Fd[256] – Fd[382] are added. Only even-numbered registers can be accessed. The XASR register is added to display the state of the additional registers. See “Extended Arithmetic Register Status Register (XASR)” (page 37) for details.

Fd[0] – Fd[126] are called the Basic Floating-Point Registers, and Fd[256] – Fd[382] are called the Extended Floating-Point Registers. In addition, Fd[0] – Fd[62] are also called V9 Floating-Point Registers.

5.3.1. Floating-Point Register Number Encoding

Refer to Section 5.3.1 in UA2011.

We expand the encoding of floating-point registers defined in UA2011 to support the addition of the HPC-ACE floating-point registers.

The XAR register contains the `urs1`, `urs2`, `urs3`, and `urd` fields, which extend the `rs1`, `rs2`, `rs3`, and `rd` fields in an instruction word. A decoded HPC-ACE register number is a 9-bit number. The upper 3 bits are specified in the XAR and are concatenated with the decoded 6-bit register number. When an instruction uses HPC-ACE floating-point registers, it must use double-precision floating-point registers. Then the least significant bit of the register number is always 0, and 128 even-numbered registers `Fd[0] – Fd[126]` and `Fd[256] – Fd[382]` can be specified by this encoding.

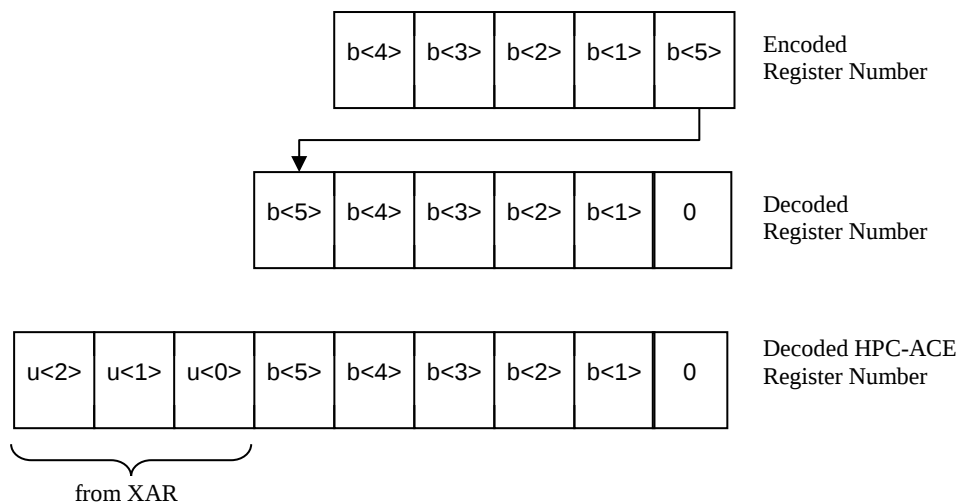


Figure 5-1 HPC-ACE Floating-Point Register Number Encoding

5.3.2. Using double-precision registers for single-precision operations

In SPARC64™ X / SPARC64™ X+, double-precision registers can be used to perform single-precision operations. This applies not only to the registers added in SPARC64™ X / SPARC64™ X+ but also to the double-precision registers defined in SPARC V9. To use a double-precision register for a single-precision operation, it is sufficient to set `XAR.v = 1` at execution time. Thus, a SIMD single-precision operation always uses double-precision registers.

When using a double-precision register for a single-precision operation, the following behavior differs from the SPARC V9 specification:

- The encoding of the instruction field is the same as for a double-precision operand in TABLE 5-3 of UA2011. Consequently, only even-numbered registers `Fd[2n]` ($n = 0 - 63, 128 - 191$) can be used.
- The upper 4 bytes of the register (bits `<63:32>`) are treated as a single-precision value, and the lower 4 bytes (bits `<31:0>`) are ignored.
- Execution results and load data are written in the upper 4 bytes, and zeroes are written in the lower 4 bytes.

Programming Note When `XAR.v = 1` and `XAR.urs1 = 0`, the SPARC V9 double-precision register specified by `rs1` is used to perform a single-precision operation. There are similar cases for `rs2`, `rs3`, and `rd`. In these situations, bits `<31:0>` of the register overlap an odd-numbered register, and will be written over with zeroes.

Endian conversion is done for each single-precision word; that is, endian conversion is done in 4-byte units.

5.3.3. Specifying registers for SIMD instructions

When `XAR.v = 1` and `XAR.SIMD = 1`, the majority of instructions that use the floating-point registers become SIMD instructions. One SIMD instruction executes two floating-point operations. Registers used for SIMD instructions must be register pairs of the form `Fd[2n]` and `Fd[2n+256]` ($n = 0 - 63$). The `Fd[2n]` register number is specified by the instruction. An *illegal_action* exception is signalled when an unusable register is specified.

The SIMD instructions listed below are special:

- **FMADD:** Registers `Fd[2n+256]` can be specified for `rs1` and `rs2`. Refer to Section 7.30 for details.
- **FSHIFTORX:** Registers `Fd[2n+256]` can be specified for `rs1`. Refer to Section 7.122 and Section 7.138 for details.
- **FAESENCLX, FAESDECLX, FAESDECX, and FAESDECLX:** Registers `Fd[2n+256]` can be used for `rs1`. Refer to Section 7.116 for details.

Programming Note Single-precision floating-point instructions support SIMD execution; however, double-precision registers must be used. See Section 5.3.2 Using double-precision registers for single-precision operations (page 24) for details.

Of the existing floating-point instructions, the following instructions do not support SIMD execution. See Table 7-3 for the list of instructions that do support SIMD execution.

- `FDIV(S, D)`, `FSQRT(S, D)`
- Most VIS instructions that are not logical operations (`FOR`, `FAND`, etc.)

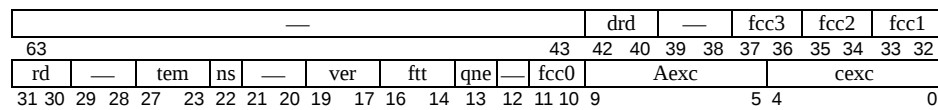
Note On SPARC64™ X+, some VIS instructions like `FPADD16{ | S }` that are not logical operations support SIMD execution.

- Instructions that reference and/or update `fcc`, `icc`, `xcc` (`FBfcc`, `FBPfcc`, `FCMP`, `FCMPE`, `FMOVcc`, etc.)
- `FMOVr`
- Instructions that have both floating-point and integer operands (`FCMPU{LE|NE|GT|EQ}8`, etc.)
- Decimal Floating-Point Operations, Compare, and Convert
- Oracle Floating-Point Operations and Oracle Decimal Floating-Point Compare

One SIMD floating-point instruction specifies two operations. The floating-point operation that stores its result in `Fd[2n]` is called the basic operation. The floating-point operation that stores its result in `Fd[2n+256]` is called the extended operation.

Endian conversion is performed separately for the basic and extended floating-point registers.

5.4. Floating-Point State Register (FSR)



Bit	Field	Access	Explanation
42:40	drd	RW	Specifies the rounding method for decimal floating-point operations. Refer to “drd” (page 26) for details.
37:36	fcc3	RW	Displays the result of a floating-point compare instruction. Refer to “fccn” (page 27) for details.
35:34	fcc2	RW	Displays the result of a floating-point compare instruction. Refer to “fccn” (page 27) for details.
33:32	fcc1	RW	Displays the result of a floating-point compare instruction. Refer to “fccn” (page 27) for details.
31:30	rd	RW	Specifies the rounding method for floating-point operations.
27:23	tem	RW	Controls whether a trap is generated for an IEEE-754 floating-point exception. Refer to “tem” (page 28) for details.
22	ns	RW	Specifies whether execution results conform to IEEE-754. Refer to “ns” (page 27) for details.
19:17	ver	R	Identifies the version of the floating-point processing unit. This field is 0 in the initial version of SPARC64™ X / SPARC64™ X+.
16:14	ftt	R	Displays information about a floating-point exception trap. Refer to “ftt” (pages 27-28) for details.
13	qne	R	Always 0.
11:10	fcc0	RW	Display the result of a floating-point compare instruction. Refer to “fccn” (page 27) for details.
9:5	aexc	RW	Accumulates all IEEE-754 floating-point exceptions that occur while floating-point exception traps are disabled. Refer to “aexc” (page 28) for details.
4:0	cexc	RW	Displays the IEEE-754 floating-point exceptions for the most recently executed FPop instruction, regardless of whether floating-point exception traps are disabled. Refer to “cexc” (page 28) for details.

drd

Bits 42-40 select the rounding direction for decimal floating-point results. The `drd` field is implemented in accordance with IEEE 754-2008 and its separate from the `rd` field, which is used for binary floating-point results. Five rounding methods defined in IEEE 754-2008 are supported. If `GSR.dim = 1`, then the value of `FSR.drd` is ignored and decimal floating-point results are instead rounded according to `GSR.dirnd`. Refer to Section 5.5.9 (page 34) for details.

Table 5-1 Decimal Rounding Direction Field of FSR

drd	Round Toward
0	Nearest (even, if tie)
1	0
2	$+\infty$
3	$-\infty$
4	Nearest (away from 0, if tie)
5-7	<i>reserved</i> The rounding result is undefined.

fccn

Refer to Section 5.4.1 in UA2011.

Additionally, execution of the following instructions updates one of the fccn fields in the FSR.

- FCMP and FCMPE
- FCMP{td|Etd|od}
- FLCMP and FPCMP{64X|U64X} (SPARC64™ X+ only)

ns

The field ns specifies whether floating-point operations conform to IEEE754.

On SPARC64™ X, when ns = 0, all operation results and exceptions conform to IEEE754. When ns = 1, instead of generating a trap, a subnormal input or output is replaced by 0 (the sign is the same as the subnormal number).

On SPARC64™ X+, when XASR.fed = 0 and ns = 0, all operation results and exceptions conform to IEEE754. When XASR.fed = 1 or ns = 1, instead of generating a trap, a subnormal input or output is replaced by 0 (the sign is the same as the subnormal number).

Refer to Section 8, “IEEE std. 754-1985 Requirements for SPARC-V9” (page 265) for details.

Programming Note If the SIAM instruction is executed to set GSR.im = 1, settings for FSR.ns will be ignored and floating-point operations will behave as if FSR.ns = 0.

Compatibility Note XASR.fed is supported only on SPARC64™ X+. When XASR.fed = 1, FSR.ns is ignored and floating-point operations behave as if FSR.ns = 1. Values set by the SIAM instruction are ignored.

ftt (on SPARC64™ X or on SPARC64™ X+ with XASR.fed = 0)

Refer to Section 5.4.6 in UA2011.

Compatibility Note Floating-point arithmetic exception disable mode is added in SPARC64™ X+. When XASR.fed = 0, the behavior of floating-point exception traps is the same as SPARC64™ X.

Note When an *fp_exception_ieee_754* trap occurs for a non-SIMD instruction, the bit corresponding to the exception is set in *cexc*. For a SIMD instruction, one or two bits are set in *cexc*.

ftt (on SPARC64™ X+ with XASR.fed = 1)

In SPARC64™ X+, *fp_exception_ieee_754* and *fp_exception_other(unfinished_FPop)* traps are not generated when XASR.fed = 1. Also, because quad FPOps are not implemented, hardware generates an *illegal_instruction* exception rather than *fp_exception_other (invalid_fp_register)*. Then no floating-point exception traps are generated when XASR.fed = 1.

FSR.tem and FSR.ns are ignored.

Programming Note When XASR.fed = 1, values set by the SIAM instructions are ignored and floating-point operations behave as if FSR.ns = 1.

When FSR.tem = 0_0000₂ and FSR.ns = 1 is set, traps for *fp_exception_ieee_754* and *fp_exception_other (unfinished_FPop)* are also not generated. However, the handling of FSR.aexc and FSR.cexc differs. The behavior of FSR.aexc and FSR.cexc observed by user software when XASR.fed = 1 is described below.

- FSR.aexc is not be updated
- FSR.cexc is cleared when instructions that update FSR are executed.

When an unexpected floating-point exception occurs while XASR.fed = 1, there is no way to determine that an exception occurred, other than the result of the operation.

tem, aexc, cexc

tem

nvm	ofm	ufm	dzm	nxm
27	26	25	24	23

aexc

nva	ofa	ufa	dza	nxa
9	8	7	6	5

cexc

nvc	ofc	ufc	dzc	nxc
4	3	2	1	0

These three fields display the five floating-point exceptions defined by IEEE 754 and control trap generation. Each field is 5 bits, where each bit corresponds to an exception defined by IEEE 754. The arrangement of bits is the same for all three fields. Table 5-3 shows the meanings of these bits.

- The tem field controls trap generations for IEEE 754 floating-point exceptions. If a floating-point instruction generates one or more exceptions and the tem bit corresponding to any of the exceptions is 1, then the exception causes a trap. A tem bit of 0 prevents the corresponding exception from generating a trap.
- The aexc field accumulates IEEE 754 floating-point exceptions that occur while floating-point exception traps are disabled.

- The cexc field displays IEEE 754 floating-point exceptions generated by the most recently executed FPop instruction. The cexc bits corresponding to the exceptions are set, and the other bits are set to zero.

Programming Note If a floating-point exception generates a trap, the recovery software should set cexc appropriately before returning.

Table 5-2 shows the values of ftt, aexc and cexc corresponding to various floating-point exception conditions.

Table 5-2 Floating-Point Exceptions and Updates to the FSR

Events	ftt	cexc	aexc
IEEE 754 floating-point exceptions are not generated.	0	0	unchanged
IEEE 754 floating-point exceptions are generated but traps are masked.	0	Bits in the cexc field corresponding to the exceptions are set.	The new cexc field is ORed into the aexc field.
IEEE 754 floating-point exceptions and traps are generated.	1	Bits in the cexc field corresponding to the exceptions are set. For non-SIMD instructions, only one bit corresponding to the highest-priority exception is set. For SIMD instructions, one or two bits are set.	unchanged
The <i>fp_exception_other</i> exception and trap are generated	2 ⁱⁱ	unchanged	unchanged

Table 5-3 Fields in aexc, cexc

Field	Exception	Notation	
		enabled	disabled
nva, nvc	An operand is improper for the operation to be performed. For example, $0.0 \div 0.0$ and $\infty - \infty$ are invalid. (1 = invalid operand(s), 0 = valid operand(s))	NV	nv
ofa, ofc	The result, rounded as if the exponent range were unbounded, would be larger in magnitude than the destination format's largest finite number. (1 = overflow, 0 = no overflow)	OF	of
ufa, ufc	The rounded result is inexact and would be smaller in magnitude than the smallest normalized number in the indicated format; (1 = underflow, 0 = no underflow) Underflow is never indicated when the correct unrounded result is 0. Otherwise: • If FSR.tem.ufm = 0: Underflow occurs if a nonzero result is tiny and a loss of accuracy occurs. • If FSR.tem.ufm = 1: Underflow occurs if a nonzero result is tiny.	UF	uf
dza, dzc	$X \div 0.0$, where X is not 0.0 nor NaN. (1 = division by zero, 0 = no division by zero)	DZ	dz
nxa, nxc	The rounded result of an operation differs from the infinitely precise unrounded result. (1 = inexact result, 0 = exact result)	NX	nx

ⁱⁱ Hardware never sets ftt = 6 (*invalid_fp_register*).

Floating-point operations which cause an overflow (of) or underflow (uf) condition may also cause an "inexact" (nx) condition. For non-SIMD instructions, only one bit in `cexc` is set if the corresponding trap is enabled in the `tem` field. Otherwise, if the exceptions are masked, all bits corresponding to generated exceptions are set. Table 5-4 summarizes how `FSR.cexc` bits are set for various exceptions and masks.

Table 5-4 Setting of `FSR.cexc` bits for non-SIMD instructions

Condition						Result			
Exception(s) detected in floating-point operation			Trap Enable Mask bits (in FSR.tem)			fp_exception_ieee_754 Trap Occurs?	Current Exception bits (in FSR.cexc)		
of	uf	nx	ofm	ufm	nxm		ofc	ufc	nxc
—	—	—	x	x	x	no	0	0	0
—	—	✓	x	x	0	no	0	0	1
—	✓iii	✓iii	x	0	0	no	0	1	1
✓iv	—	✓iv	0	x	0	no	1	0	1
—	—	✓	x	x	1	yes	0	0	1
—	✓iii	✓iii	x	0	1	yes	0	0	1
—	✓	—	x	1	x	yes	0	1	0
—	✓	✓	x	1	x	yes	0	1	0
✓iv	—	✓iv	1	x	x	yes	1	0	0
✓iv	—	✓iv	0	x	1	yes	0	0	1

Updates to `cexc` and `aexc` for SIMD instructions

For SIMD instructions, two bits might be set in `cexc` when traps are enabled.

Basic and extended operations are performed simultaneously. However, because the source operands are different, either operation or both could cause exceptions.

When only one operation causes an exception, the behavior is the same as for a non-SIMD instruction. When both operations cause exceptions, `cexc`, `aexc` and `ftt` are updated and traps are generated as shown below. For the purposes of illustration, let's say the exception caused by the basic operation updates the hypothetical `basic.aexc` and `basic.cexc` fields. The exception caused by the extended operation updates the hypothetical `extend.aexc` and `extend.cexc` fields.

- When `fp_exception_ieee_754` exceptions are detected for both basic and extended operations:
 - Both exceptions are masked and no exception is signaled:
The logical OR of `basic.cexc` and `extend.cexc` is displayed in `FSR.cexc`. The logical OR of `basic.cexc` and `extend.cexc` is accumulated in `FSR.aexc`.

$$\text{FSR.cexc} \leftarrow \text{basic.cexc} \mid \text{extend.cexc}$$

$$\text{FSR.aexc} \leftarrow \text{FSR.aexc} \mid \text{basic.cexc} \mid \text{extend.cexc}$$
 - Either the basic or extended operation signals an exception:
The logical OR of `basic.cexc` and `extend.cexc` is displayed in `FSR.cexc`. `FSR.aexc` is unchanged.

$$\text{FSR.cexc} \leftarrow \text{basic.cexc} \mid \text{extend.cexc}$$

ⁱⁱⁱ Except for `FRCPA{s|d}`, when the underflow trap is disabled (`FSR.tem.ufm = 0`), underflow (uf) is always accompanied by inexact (nx). For `FRCPA{s|d}`, when the underflow trap is disabled (`FSR.tem.ufm = 0`), underflow (uf) is not accompanied by inexact (nx).

^{iv} Overflow (of) is always accompanied by inexact (nx).

- Both basic and extended operations signal exceptions:

The logical OR of basic.cexc and extend.cexc is displayed in FSR.cexc. FSR.aexc is unchanged.

$$\text{FSR.cexc} \leftarrow \text{basic.cexc} \mid \text{extend.cexc}$$

- When *fp_exception_ieee_754* is detected for one operation and *fp_exception_other* is detected for the other operation, the *fp_exception_other* exception is signalled with *ftt = unfinished_FPop*. Both FSR.aexc and FSR.cexc are unchanged.

Programming Note When an *fp_exception_other* exception is generated, it is impossible for hardware to determine whether an *fp_exception_ieee_754* exception occurred simultaneously. System software must run an emulation routine to detect the second exception and update the necessary registers.

- When *fp_exception_other* exceptions are detected for both basic and extended operations, an *fp_exception_other* with *ftt = unfinished_FPop* is signalled. Both FSR.aexc and FSR.cexc are unchanged.

tem, cexc, aexc on SPARC64™ X+ with XASR.fed = 1

On SPARC64™ X+ with XASR.fed = 1, the value of tem is ignored and no floating-point exception traps are generated. Changing the value of tem has no effect on this behavior. In this case, the aexc and cexc fields are updated as follows.

- The aexc field is unchanged.
- The cexc field is cleared when an instruction that updates FSR is executed.

FSR Conformance

A SPARC V9 implementation may choose to implement the tem, cexc, and aexc fields in hardware in either of two ways (both of which comply with IEEE Std 754-1985). On SPARC64™ X / SPARC64™ X+, all three fields are implemented.

5.5. Ancillary State Registers

5.5.1. 32-bit Multiply/Divide Register (Y) (ASR 0)

Refer to Section 5.5.1 in UA2011.

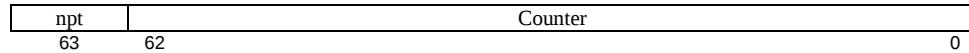
5.5.2. Integer Condition Codes Register (CCR) (ASR 2)

Refer to Section 5.5.2 in UA2011.

5.5.3. Address Space Identifier (ASI) Register (ASR 3)

Refer to Section 5.5.3 in UA2011.

5.5.4. Tick (TICK) Register (ASR 4)



The counter field of the TICK register is a 63-bit counter (SPARC V9 Impl. Dep. #105b) that counts processor clock cycles. Bit 63 of the TICK register is the nonprivileged trap (npt) bit, which controls access to the TICK register by nonprivileged software.

Compatibility Note Each thread in SPARC64™ X / SPARC64™ X+ has its own copy of the npt field and the counter field.

Nonprivileged software can read the TICK register, but only when nonprivileged access to TICK is enabled (TICK.npt = 0). If nonprivileged access is disabled (TICK.npt = 1), an attempt by nonprivileged software to read the TICK register causes a *privileged_action* exception. Table 5-5 shows the exceptions generated by reading or writing the TICK register.

Table 5-5 Exceptions when reading or writing the TICK register

RDTICK	(WRTICK doesn't exist)	RDPR	WRPR
OK (if TICK.npt = 0) <i>privileged_action</i> (if TICK.npt = 1)	—	<i>privileged_opcode</i>	<i>privileged_opcode</i>

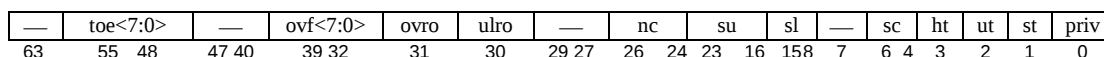
5.5.5. Program Counters (PC, NPC) (ASR 5)

Refer to Section 5.5.5 in UA2011.

5.5.6. Floating-Point Registers State (FPRS) Register (ASR 6)

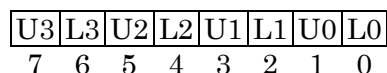
Refer to Section 5.5.6 in UA2011.

5.5.7. Performance Control Register (PCR) (ASR 16)



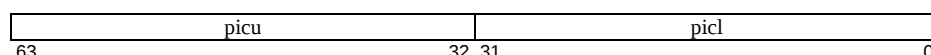
Bit	Field	R/W	Description
55:48	toe<7:0>	RW	Controls whether an overflow exception is generated for performance counters. A write updates the field, and a read returns the current settings. If PCR.toe<i> = 1 and the counter corresponding to PCR.ovf<i> overflows, PCR.ovf<i> is set to 1 and a <i>pic_overflow</i> exception is generated. If PCR.toe<i> = 0 and the counter corresponding to PCR.ovf<i> overflows, PCR.ovf<i> is set to 1 but a <i>pic_overflow</i> exception is not generated. When PCR.ovf<i> is already 1 and PCR.toe<i> is changed to 1 from 0, a <i>pic_overflow</i> exception is not generated.
39:32	ovf<7:0>	RW	Overflow Clear/Set/Status. A read by RDPCR returns the overflow status of the counters (if ovf = 1, the corresponding counter has overflowed). PCR.ovf<2n> and PCR.ovf<2n+1> refer to the lower counter (PIC<31:0>) and upper counter (PIC<63:32>), respectively, of the n-th

counter pair selected by PCR.sc. A write of 0 to an ovf bit clears the overflow status of the corresponding counter. Writing a 1 via software does not cause a *pic_overflow* exception.



31	ovro	RW	Overflow Read-Only. A write to the PCR register with write data containing a value of ovro = 0 updates the PCR.ovf field with the ovf write data. If the write data contains a value of ovro = 1, the ovf write data is ignored and the PCR.ovf field is not updated. A read of the PCR.ovro field returns 0. The PCR.ovro field allows PCR to be updated without changing the overflow status. Hardware maintains the most recent state in PCR.ovf such that a subsequent read of the PCR returns the current overflow status.
30	ulro	RW	SU/SL Read-Only. A write to the PCR register with write data containing a value of ulro = 0 updates the PCR.su and PCR.sl fields with the su/sl write data. If the write data contains a value of ulro = 1, the su/sl write data is ignored and the PCR.su and PCR.sl fields are not updated. A read of the PCR.ulro field returns 0.
26:24	nc	RO	Indicates the number of counter pairs. On SPARC64™ X / SPARC64™ X+, nc has a value of 3 (indicating 4 counter pairs). Writes to the PCR.nc are ignored.
23:16	su	RW	Selects the event counted by PIC<63:32>. A write updates the field, and a read returns the current setting.
15:8	sl	RW	Selects the event counted by PIC<31:0>. A write updates the field, and a read returns the current setting.
6:4	sc	RW	PIC Pair Selection. A write updates which PIC counter pair is selected, and a read returns the current selection.
3	ht	RO	If PCR.ht = 1, events are counted in hypervisor mode. PCR.ht can be read. Writes to PCR.ht are ignored.
2	ut	RW	Non-privileged Mode. When PSTATE.priv = 0 and PCR.ut = 1, events are counted.
1	st	RW	System Mode. When PSTATE.priv = 1 and PCR.st = 1, events are counted.
0	priv	RW	Privileged. If PCR.priv = 1, executing a RDPCR, WRPCR, RDPIC, or WRPIC instruction in non-privileged mode (PSTATE.priv = 0) causes a <i>privileged_action</i> exception. If PCR.priv = 0, a RDPCR, WRPCR, RDPIC, or WRPIC instruction can be executed in non-privileged mode. If PCR.priv = 0, a non-privileged (PSTATE.priv = 0) attempt to update PCR.priv (that is, to write a value of 1) via a WRPCR instruction causes a <i>privileged_action</i> exception.

5.5.8. Performance Instrumentation Counter (PIC) Register (ASR 17)



Bit	Field	Access	Explanation
63:32	picu	RW	32-bit counter for the event selected by the su field of the Performance Control Register (PCR).
31:0	picl	RW	32-bit counter for the event selected by the sl field of the Performance Control Register (PCR).

5.5.9. General Status Register (GSR) (ASR 19)

Refer to Section 5.5.7 in UA2011.

mask				dim	dirnd	im	irnd	reserved				scale	align			
63			32	31	30	28	27	26	25	24		8	7	3	2	0
Bit	Field	Access	Explanation													
63:32	mask	RW	Refer to UA2011.													
31	dim	RW	Interval Mode for decimal floating-point numbers. When dim = 1, the value in FSR.drd is ignored. The processor rounds floating-point results according to GSR.dirnd.													
30:28	dirnd	RW	Rounding direction to use in interval mode for decimal floating-point numbers. GSR.dirnd is valid when GSR.dim = 1. Refer to FSR.drd (page 26) for details.													
27	im	RW	Refer to UA2011.													
26:25	irnd	RW	Refer to UA2011.													
24:8	reserved	RO	Reserved (undefined)													
7:3	scale	RW	Refer to UA2011.													
2:0	align	RW	Refer to UA2011.													

Note A read of a *reserved* field returns an undefined value. Zeros must be written to *reserved* fields to preserve compatibility with future implementations.

5.5.11. System Tick (STICK) Register (ASR 24)

npt	Counter														
63	62														0

The counter field of the STICK register is a 63-bit counter that increments at a rate determined by a clock signal external to the processor. Bit 63 of the STICK register is the nonprivileged trap (NPT) bit, which controls access to the STICK register by nonprivileged software. A clock signal external to the processor is not defined in this specification.

Compatibility Note Each thread in SPARC64™ X / SPARC64™ X+ has its own copy of the npt field and the counter field.

Nonprivileged software can read the STICK register by using the RDSTICK instruction, but only when nonprivileged access to STICK is enabled (STICK.npt = 0). If nonprivileged access is disabled (STICK.npt = 1), an attempt by nonprivileged software to read the STICK register causes a *privileged_action* exception. Table 5-6 shows the exceptions generated when reading or writing the STICK register.

Table 5-6 Exceptions when reading or writing the STICK register

RDSTICK	WRSTICK
OK (if STICK.npt = 0) <i>privileged_action</i> (if STICK.npt = 1)	<i>illegal_instruction</i> (differs from TICK register)

Compatibility Note In JPS1, writing the STICK register in nonprivileged mode generates a *privileged_opcode* exception.

A read of STICK.counter<6:0> always returns 0xf.

5.5.13. Pause Register (PAUSE) (ASR 27)

—			pause		—	
63			15		3	
14			2		0	
Bit	Field	Access	Description			
63:15	<i>reserved</i>	WO	reserved			
14:3	pause	WO	Pause VCPU for the specified number of processor cycles.			
2:0	<i>reserved</i>	WO	ignored			

A virtual processor's PAUSE register is used to pause execution on the virtual processor for the number of cycles specified by the WRPAUSE or PAUSE instruction. Software initiates a pause by writing the number of cycles to the bits PAUSE<14:0>. The lowest 3 bits of the PAUSE register are ignored. Then the maximum duration that can be specified by a WRPAUSE or PAUSE instruction is 32760 virtual processor cycles.

5.5.14. Extended Arithmetic Register (XAR) (ASR 29)

0	f_v	0	f_simd	f_urd	f_urs1	f_urs2	f_urs3	s_v	0	s_simd	s_urd	s_urs1	s_urs2	s_urs3											
63	32	31	30	29	28	27	25	24	22	21	19	18	16	15	14	13	12	11	9	8	6	5	3	2	0

Bit	Field	Access	Description
31	f_v	RW	Indicates whether the contents of fields beginning with f_ are valid. If f_v = 1, the contents of the f_ fields are applied to the instruction that executes first. After the 1st instruction completes, all f_ fields are cleared.
28	f_simd	RW	If f_simd = 1, the 1st instruction is executed as a SIMD instruction. If f_simd = 0, execution is non-SIMD.
27:25	f_urd	RW	Extends the rd field of the 1st instruction.
24:22	f_urs1	RW	Extends the rs1 field of the 1st instruction.
21:19	f_urs2	RW	Extends the rs2 field of the 1st instruction.
18:16	f_urs3	RW	Extends the rs3 field of the 1st instruction.
15	s_v	RW	Indicates whether the contents of fields beginning with s_ are valid. If s_v = 1, the contents of the s_ fields are applied to the instruction that executes second. After the 2nd instruction completes, all s_ fields are cleared.
12	s_simd	RW	If s_simd = 1, the 2nd instruction is executed as a SIMD instruction. If s_simd = 0, execution is non-SIMD.
11:9	s_urd	RW	Extends the rd field of the 2nd instruction.
8:6	s_urs1	RW	Extends the rs1 field of the 2nd instruction.

5:3	s_urs2	RW	Extends the rs2 field of the 2nd instruction.
2:0	s_urs3	RW	Extends the rs3 field of the 2nd instruction.

The XAR register extends the fields in an instruction word. It holds the upper 3 bits of an instruction's register number fields (rs1, rs2, rs3, rd) and indicates whether the instruction is a SIMD instruction.

The register contains fields for two separate instructions. There are V (valid) bits for the first and second instructions; all other fields for a given instruction are valid only when v = 1. These register fields are mainly used to specify floating-point registers, except the *_urs3<1> fields, which are also used to disable hardware prefetch for integer and floating-point load/store instructions.

When a trap occurs, the contents of the XAR are saved to the TXAR[TL] register, and all fields in the XAR are set to 0. The saved value corresponds to the value of the XAR just before the instruction that caused the trap was executed.

Note If a TCC initiates a trap, the contents of the XAR just before the Tcc instruction was executed are saved.

Aliases of XAR fields in this specification

The fields described in Table 5-7 have the following aliases.

Table 5-7 Alias for memory access

Aliases	Field	Usage
XAR.f_dis_hw_pf	XAR.f_urs3<1>	Disable hardware prefetch
XAR.s_dis_hw_pf	XAR.s_urs3<1>	Disable hardware prefetch
XAR.f_negate_mul	XAR.f_urd<2>	For SIMD FMA
XAR.s_negate_mul	XAR.s_urd<2>	For SIMD FMA
XAR.f_rs1_copy	XAR.f_urs3<2>	For SIMD FMA
XAR.s_rs1_copy	XAR.s_urs3<2>	For SIMD FMA

XAR operations

Only some instructions can reference the XAR register. In this document, instructions that can reference XAR are called “XAR-eligible instructions”. Refer to Table 7-3 (page 43) for details on which instructions are XAR eligible.

- An attempt to execute an instruction that is not XAR-eligible while XAR.v = 1 causes an *illegal_action* exception.
- XAR-eligible instructions have the following behavior.
 - If XAR.v = 1, the XAR.urs1, XAR.urs2, XAR.urs3 and XAR.urd fields are concatenated with the instruction fields rs1, rs2, rs3, and rd respectively, to specify floating-point registers. The XAR.urs3<1> fields may instead be used to disable hardware prefetch for integer and floating-point load/store instructions. Floating-point registers are referenced by 9-bit register numbers; the XAR fields specify the upper 3 bits. A double-precision encoded 5-bit instruction field is decoded to generate the lower 6 bits of the register number. Refer to “5.3.1 Floating-Point Register Number Encoding” (page 24) for details.

- If `XAR.f_v = 1`, the `XAR.f_urs1`, `XAR.f_urs2`, `XAR.f_urs3` and `XAR.f_urd` fields are used.
- If `XAR.f_v = 0` and `XAR.s_v = 1`, the `XAR.s_urs1`, `XAR.s_urs2`, `XAR.s_urs3` and `XAR.s_urd` fields are used.
- The values of the `f_` or `s_` fields are only valid once. After the instruction referencing the XAR register completes, the referenced fields are set to 0.
- XAR-eligible instructions cause *illegal_action* exceptions in the following cases.
 - `XAR.urs1 ≠ 0` is specified for an instruction that does not use `rs1`. There are similar cases for `rs2`, `rs3` and `rd`.
`XAR.urs1<1> ≠ 0` is specified for an instruction that uses `rs1`. There are similar cases for `rs2`, `rs3` and `rd`.
`XAR.urs2 ≠ 0` is specified for an instruction whose `rs2` field holds an immediate value (such as `simm13` or `fcn`).
 - A register number greater than or equal to `F[256]` is specified for the `rd` field of an `FDIV{S|D}` or `FSQRT{S|D}` instruction.
 - `XAR.simd = 1` for an instruction (including integer arithmetic) that does not support SIMD execution.
 - `XAR.simd = 1`, and a register number greater than or equal to `F[256]` is specified. Some instructions (`F{N}MADD{s|d}`, `F{N}MSUB{s|d}`, `FAES*X` and so on) are exceptions to this rule; register numbers greater than or equal to `F[256]` can be specified. Refer to the specification for each instruction.
 - `XAR.urs3<2> ≠ 0` for a `ld/st/atomic` instruction.

When the XAR specifies register numbers for only one instruction, either the `f_` or `s_` fields can be used.

Programming Note If the `WRXAR` instruction is used, either `XAR.f_v` or `XAR.s_v` can be set to 1. The `SXAR1` instruction sets `XAR.f_v` to 1.

If `XAR.f_v = 0`, the `f_simd`, `f_urs1`, `f_urs2`, `f_urs3`, and `f_urd` fields are ignored even when the fields contain non-zero values. The value of each field after execution is undefined. If `XAR.s_v = 0`, the `s_simd`, `s_urs1`, `s_urs2`, `s_urs3`, and `s_urd` fields are ignored even when the fields contain non-zero values. The value of each field after execution is undefined.

5.5.15. Extended Arithmetic Register Status Register (XASR) (ASR 30)

<SPARC64™ X>

<i>reserved</i>						<i>xfd<5:4></i>	<i>reserved</i>	<i>xfd<1:0></i>
63		6	5	4	3	2	1	0

Bit	Field	Access	Description
63:6	<i>reserved</i>	RO, RW	<i>reserved</i> (undefined). <63:9> is RO, <8:6> is RW.
5:4	<i>xfd<5:4></i>	RW	Updating a floating-point register (<code>F[382] - F[256]</code>) sets the appropriate bit to 1. Refer to <i>xfd</i> (page 39) for details.
3:2	<i>reserved</i>	RW	<i>reserved</i> (undefined)
1:0	<i>xfd<1:0></i>	RW	Updating a floating-point register (<code>F[126] - F[0]</code>) sets the appropriate bit to 1. Refer to <i>xfd</i> (page 39) for details.

<SPARC64™ X+>

<i>reserved</i>		<i>fed</i>	<i>reserved</i>		<i>xfd</i> <5:4>	<i>reserved</i>	<i>xfd</i> <1:0>			
63	37	36	35	6	5	4	3	2	1	0

Bit	Field	Access	Description
63:37	<i>reserved</i>	RO	Reserved (undefined).
36	<i>fed</i>	RW	Floating-Point Exception Disable mode. No floating-point exception traps are generated (supported from SPARC64™ X+).
35:6	<i>reserved</i>	RO, RW	<i>reserved</i> (undefined). <35:9> is RO, <8:6> is RW
5:4	<i>xfd</i> <5:4>	RW	Updating a floating-point register (F[382] - F[256]) sets the appropriate bit to 1. Refer to <i>xfd</i> (page 39) for details.
3:2	<i>reserved</i>	RW	<i>reserved</i> (undefined)
1:0	<i>xfd</i> <1:0>	RW	Updating a floating-point register (F[126] - F[0]) sets the appropriate bit to 1. Refer to <i>xfd</i> (page 39) for details.

Note A read of a *reserved* field returns an undefined value. Zeros must be written to *reserved* field to preserve compatibility for future implementation.

fed (supported on SPARC64™ X+)

Setting the *fed* field masks all floating-point exceptions. When *XASR.fed* = 0, the behavior of floating-point exceptions are the same as SPARC64™ X. This field is updated by the *WRXASR* instruction.

All floating-point exceptions are masked when *XASR.fed* = 1. That is, corresponding traps are not generated. In addition, *FSR.aexc* is not updated and *FSR.ftt* is cleared by 0, regardless of the values of *FSR.tem* and *FSR.ns*. Refer to *FSR* (pages 27, 28) for details. Also, the *FSHIFTORX* instruction does not generate an *illegal_instruction* trap.

Exception	<i>XASR.fed</i> = 0	<i>XASR.fed</i> = 1
<i>fp_exception_ieee</i>	Behavior specified by <i>FSR.tem</i>	Trap is not generated. If an instruction that updates <i>FSR</i> is executed <i>FSR.cexc</i> and <i>FSR.ftt</i> are cleared <i>FSR.aexc</i> is not updated
<i>fp_exception_other</i> (<i>unfinished_FPop</i>)	Behavior specified by <i>FSR.ns</i>	Trap is not generated.
<i>illegal_instruction</i> (<i>FSHIFTORX</i>)	<i>illegal_instruction</i> trap is generated depending on the value of <i>Fd[rs3]</i>	Trap is not generated. The value in <i>Fd[rd]</i> is undefined.

Operation results for *fed* = 1 are the same as *fed* = 0, *FSR.tem* = 0_0000₂ and *FSR.ns* = 1 except for the behavior of the *FSHIFTORX* instruction.

The use of this flag is determined solely by the compiler. In other words, user-privileged software routines generated by the compiler, and compiler startup routines or libraries can use this field.

The Compiler can freely choose to alter this flag or leave it untouched. Nonprivileged software not generated by the compiler (e.g., assembly language) should not alter this flag.

When modifying this field, it is caller's responsibility to clear the flag before jumping to routines that are not generated by the compiler, such as OS library routines.

Note Minimizing the period where XASR.fed = 1 is recommended.

xfd

The xfd fields are used to determine whether any of the floating-point registers need to be saved during a context switch. Updating a register sets the appropriate bit to 1.

- There is no flag indicating an update to an integer register.
- Updating a floating-point register sets the appropriate XASR.xfd<i> = 1. The floating-point registers and corresponding xfd bits are shown below.

xfd bits	Corresponding floating-point registers
0	F[0] - F[62]
1	F[64] - F[126]
2	<i>Reserved</i>
3	<i>Reserved</i>
4	F[256] - F[318]
5	F[320] - F[382]
6	<i>Reserved</i>
7	<i>Reserved</i>

Programming Note Updating a V9 floating-point register sets the xfd[0] bit of the XASR and also updates the V9 FPRS. For example, updating F[15] sets both FPRS.dl = 1 and XASR.xfd<0> = 1.

Programming Note The fields XASR.xfd<7:6> and XASR.xfd<3:2> are undefined.

6. Instruction Set Overview

6.1. Instruction Execution

Refer to Section 6.1 in UA2011.

6.2. Instruction Formats

Refer to Section 6.2 in the SPARC64 VIIIfx extensions.

6.3. Instruction Categories

Refer to Section 6.3 in UA2011.

6.3.4.3 CALL and JMPL Instructions

Compatibility Note When PSTATE.am = 1, the upper 32bits of %o7 are set to 0.

6.3.4.6 Trap Instruction (Tcc)

Compatibility Note Traps numbered 128 ~ 255 trap to hypervisor mode. It is not possible to trap directly from nonprivileged mode to hypervisor mode. The base address of the trap vector is HTBA.

6.3.9 Floating-Point Operate (FPop) Instructions

FPop refers to floating-point execution instructions (except for FBfcc, FBPfcc, and load/store instructions). FPop1 and FPop2 are defined in Table 14-5 ~ Table 14-7 (page 317) and are FPop instructions. FPop also includes IMPDEP1 and IMPDEP2, which are described in the table below.

IMPDEP1	IMPDEP2
FADDtd, FSUBtd, FMULtd, FDIVtd, FCMP{E}td, FQUAtd, FADDod, FSUBod, FMULod, FDIVod, FCMPod, F{R}QUAod, FXADDod{LO HI}, FXMULodLO, FbuxT0td, FtdTObux, FbsxT0td, FtdTObsx, FodT0td, FtdTOod, FCMP{LE LT GE GT EQ NE}{E}{s d}, FMAX{s d}, FMIN{s d}, FRCPA{s d}, FRSQRTA{s d}, FTRISSEld, FTRISMULd, FEXPAd	F{N}MADD{s d}, F{N}MSUB{s d}, FTRIMADDd, FSELMOV{s d}

6.3.11 Reserved Opcodes and Instruction Fields

Compatibility Note An *illegal_instruction* exception is generated when an attempt is made to execute an instruction where one or more reserved fields in the instruction word are not 0. In JPS1, this behavior was not clearly described in the footnote.

7. Instructions

This chapter describes instructions defined in SPARC64™ X / SPARC64™ X+. Refer to Chapter 7 of UA2011 for instructions not defined in this chapter.

Compatibility Note When the specification of an instruction in JPS1 differs from UA2011, SPARC64™ X / SPARC64™ X+ conform to UA2011. There are several differences between JPS1 and UA2011 that are not a result of the differences between sun4u and sun4v (for example, handling of *reserved* fields and the exceptions generated by quadruple-precision floating-point instructions).

Instruction definitions for SPARC64™ X / SPARC64™ X+ include the following descriptions.

- Table of opcodes for instructions defined in the subsection. This table also includes values for unique field(s) whether HPC-ACE features can be used with the instruction, and assembly language notation.
- Illustration of the applicable instruction format(s). Fields marked “*reserved*” are reserved for future expansion and must be set to 0 by software on SPARC64™ X / SPARC64™ X+. Refer to Section 1.1.3 (page 9) for details about the meaning of *reserved* and the em dash (—).
- Description of the instruction and restrictions.

Note Exceptional conditions are summarized in the table at the end of the subsection. Exceptions may be described in the description when further explanation is needed.

- For floating-point arithmetic instructions, a table relating input operands to the arithmetic result. The table also relates input operands to the arithmetic exceptions (OF, UF, DZ, NX, NV) defined by IEEE 754.
- Table of exceptions that can occur when the instruction is executed. Exceptions are listed in descending priority order. The highest-priority exception is listed first.

The following exceptions are not described in this table.

- *IAE_** exceptions can occur for all instructions
- *illegal_instruction* exceptions occur for unimplemented instructions

XAR fields are shown without the *f_* or *s_* prefix, except when describing conditions that cause an *illegal_action* exception.

No timing information is described.

Table 7-3 shows the list of all instructions supported by SPARC64™ X / SPARC64™ X+. Certain instructions are marked with mnemonic superscripts. These superscripts are also used in Chapter 14, “Opcode Maps” (page 314). Table 7-1 lists these mnemonic superscripts and their meanings.

Table 7-1 Meaning of mnemonic superscripts

Superscript	Meaning
D	Instruction should not be used (Deprecated)
N	Incompatible instruction
P _{ASI}	Privileged operation when bit 7 of ASI is 0.
P _{ASR}	Privileged operation depending on the ASR number
P _{NPT}	Privileged operation when PSTATE.PRIV = 0 and {S}TICK.NPT = 1
P _{PIC}	Privileged operation when PCR.PRIV = 1
P _{PCR}	Privileged access when PCR.PRIV = 1
+	Instruction not supported on SPARC64™ X.

The description of an instruction may use the notation described in Table 7-2 when referring to specific operands.

Table 7-2 Register notation for rs1 (same for rs2, rs3, and rd)

Notation	Meaning	
	XAR.v = 0	XAR.v = 1
R[rs1]	Integer register encoded by the rs1 field of the instruction word	Integer register encoded by the rs1 field of the instruction word
Fs[rs1]	Single-precision floating-point register encoded by the rs1 field of the instruction word	Single-precision floating-point register encoded by XAR.urs1 and the rs1 field of the instruction word
Fd[rs1]	Double-precision floating-point register encoded by the rs1 field of the instruction word	Double-precision floating-point register encoded by XAR.urs1 and the rs1 field of the instruction word
F[rs1]	Floating-point register encoded by the rs1 field of the instruction word (no distinction made between single precision, double precision, or quadruple precision)	Floating-point register encoded by XAR.urs1 and the rs1 field of the instruction word (no distinction made between single precision, double precision, or quadruple precision)

In the Table 7-3, the columns for HPC-ACE extension show which HPC-ACE features can be used with an instruction on SPARC64™ X / SPARC64™ X+.

- **Regs.** XAR-eligible instruction. The extended floating-point registers can be used. For memory access instructions, hardware prefetch can be disabled.

An instruction which has a ☆ in this column can specify Fd[0] - Fd[126] for the rd register but not Fd[256] - Fd[382] .

- **SIMD** Instruction can be specified as a SIMD instruction.

Instructions without checks in either of these two columns are not XAR eligible. Instructions that are not XAR eligible are described in "XAR operations" (page 36).

Table 7-3 Instruction set of SPARC64™ X / SPARC64™ X+

Instruction	HPC-ACE extension		Page
	Regs.	SIMD	
ADD (ADDcc)			51
ADDC (ADDCcc)			51
ALIGNADDRESS{_LITTLE}			52
AND (ANDcc)			120
ANDN (ANDNcc)			120
ARRAY{8 16 32}			53
BMASK			54
BPcc			56
BPr			57
BSHUFFLE			54
Bicc ^D			55
CALL			58
CASA ^{PASI} , CASXA ^{PASI}	✓		59
CWB{NE E G LE GE L GU LEU CC CS POS NEG VC VS} ⁺			242
CXB{NE E G LE GE L GU LEU CC CS POS NEG VC VS} ⁺			242
EDGE{8 16 32}{L}N			62
EDGE{8 16 32}{L}cc			61
FABSq	✓		67
FABS{s d}	✓	✓	67
FADDod	☆		209
FADDq	✓		68
FADD{s d}	✓	✓	68
FADDtd	☆		204
FAESDECLX	✓	✓	197
FAESDECX	✓	✓	197
FAESENCLX	✓	✓	197
FAEENCX	✓	✓	197
FAESKEYX	✓	✓	197
FALIGNDATA			70
FANDNOT{1 2}{s}	✓	✓	106
FAND{s}	✓	✓	106
FBPfcc			72
FBfcc ^D			71
FCMP{E}{s d q}	✓		73
FCMP{E}td	✓		217
FCMP{LE LT GE GT EQ NE}{E}{s d}	✓	✓	74
FCMP{LE NE GT EQ}{16 32}			76
FCMP{LE GT}{8X 16X 32X X}	☆		226
FPCMP{LE GT}{8X 16X 32X 64X} ⁺	✓	✓	257
FCMPod	✓		219
FLCMP{s d} ⁺	✓		234
FDESENCX	✓	✓	192
FDESIIPX	✓	✓	192
FDESIPX	✓	✓	192
FDESKEYX	✓	✓	192
FDESPC1X	✓	✓	192

Instruction	HPC-ACE extension		Page
FDIVod	☆		209
FDIV{s d q}	☆		77
FDIVtd	☆		204
FEXPAd	✓	✓	78
FEXPAND			80
FLUSH			81
FLUSHW			82
FMADD{s d}	✓	✓	83
FMAX{s d}	✓	✓	91
FMIN{s d}	✓	✓	91
FMOVq	✓		93
FMOVcc			94
FMOVR			95
FMOV{s d}	✓	✓	93
FMSUB{s d}	✓	✓	83
FMUL8x16			96
FMUL8x16{AU AL}			96
FMUL8{SU UL}x16			96
FMULD8{SU UL}x16			96
FMULod	☆		209
FMULq	✓		97
FMUL{s d}	✓	✓	97
FMULtd	☆		204
FNAND{s}	✓	✓	106
FNEGq	✓		98
FNEG{s d}	✓	✓	98
FNMADD{s d}	✓	✓	83
FNMSUB{s d}	✓	✓	83
FNADD{s d} ⁺	✓	✓	235
FN MUL{s d} ⁺	✓	✓	237
FNsMULd ⁺	✓	✓	237
FNOR{s}	✓	✓	106
FNOT{1 2}{s}	✓	✓	106
FONE{s}	✓	✓	106
FORNOT{1 2}{s}	✓	✓	106
FOR{s}	✓	✓	106
FPACK{16 32 FIX}			99
FPADD{16 32}{S}			100
FPADD{16 32}{S} ⁺	✓	✓	100
FPADD64 ⁺	✓	✓	231
FPADD128XHI ⁺	✓	✓	261
FPMADDX{HI}	✓	✓	102
FPMAX{u}{32 64} ⁺	✓	✓	262
FPMIN{u}{32 64} ⁺	✓	✓	262
FPMERGE			103
FPSUB{16 32}{S}			104

Instruction	HPC-ACE extension		Page
FPSUB{16 32}{S} ⁺	✓	✓	104
FPSUB64 ⁺	✓	✓	232
F{R}QUAod	☆		209
FQUAtd	☆		204
FRCPA{s d}	✓	✓	109
FRSQRTA{s d}	✓	✓	109
FPSELMOV{8 16 32}X ⁺	✓	✓	243
FSELMOV{s d}	✓	✓	112
FSHIFTORX	✓	✓	222
FSHIFTORX ⁺	✓	✓	252
FSQRT{s d q}	☆		113
FSRC{1 2}{s}	✓	✓	106
FSUBod	☆		209
FSUBq	✓		68
FSUB{s d}	✓	✓	68
FSUBtd	☆		204
FTRIMADDd	✓	✓	114
FTRISMULd	✓	✓	114
FTRISSEld	✓	✓	114
FUCMP{LE NE GT EQ}{8X 16X 32X X}	☆		226
FPCMPU{LE NE GT EQ}{8X 16X 32X 64X} ⁺	✓	✓	257
FPCMPU{LE NE GT EQ}8 ⁺			233
FPCMP{64 U64}X ⁺	✓		246
FXADDod{LO HI}	☆		209
FXMULodLO	☆		209
FXNOR{s}	✓	✓	106
FXOR{s}	✓	✓	106
FZERO{s}	✓	✓	106
FdMULq	✓		97
F{bsx bux od}T0td	☆		220
FqT0{i x}	✓		66
FsMULd	✓	✓	97
F{i x}T0q	✓		63
F{i x}T0{s d}	✓	✓	63
F{s d}T0q	✓		64
F{s d}T0{i x}	✓	✓	66
FsT0d, FdT0s	✓	✓	64
FtdT0{bsx bux od}	☆		220
FqT0{s d}	✓		64
ILLTRAP			119
JMPL			121
LDBLOCKF	✓		124
LDF, LDDF	✓	✓	126
LDQF	✓		126
LDFA ^{PASI} , LDDFA ^{PASI}	✓	✓	129

Instruction	HPC-ACE extension		Page
LDQFA ^{PASI}	✓		129
LDFSR ^D	✓		140
LDSHORTF			132
LDSTUB	✓		133
LDSTUBA ^{PASI}	✓		134
LDTW ^D	✓		135
LDTWA ^{D,PASI}	✓		136
LDTXA ^N	✓		138
LDXEFSR+	✓		241
LDXFSR	✓		140
LD{S U}{B H W}, LDX	✓		122
LD{S U}{B H W}A ^{PASI} , LDXA ^{PASI}	✓		123
LZD			230
MEMBAR			141
MOVcc			142
MOVr			143
MOVwTos ⁺	✓		264
MOVxTod ⁺	✓		264
MULScc ^D			144
MULX			146
NOP			147
OR (Orcc)			120
ORN (ORNcc)			120
PADD32			148
PAUSE ⁺			239
PDIST			149
POPC			150
PREFETCH, PREFETCHA ^{PASI}	✓		151
RDASI			154
RDCCR			154
RDFPRS			154
RDGSR			154
RDPC			154
RDPCR ^{PCR}			154
RDPIC ^{PIC}			154
RDSTICK ^{NPT}			154
RTICK ^{NPT}			154
RDXASR			154
RDY ^D			154
RDASR ^{PASR}			154
RESTORE			156
RETURN			155
ROLX			160
SAVE			156
SDIAM			159

Instruction	HPC-ACE extension		Page
SDIV ^D (SDIVcc ^D)			157
SDIVX			146
SETHI			158
SIAM			159
SLEEP			162
SLL, SLLX			160
FPSLL64X+	✓	✓	247
SMUL ^D (SMULcc ^D)			161
SRA, SRAX			160
FPSRA64X+	✓	✓	247
SRL, SRLX			160
FPSRL64X+	✓	✓	247
STBAR ^D			163
STBI ^N	✓		166
STBLOCKF	✓		167
STF, STDF	✓	✓	169
STQF	✓		169
STFA ^{PASI} , STDFA ^{PASI}	✓	✓	171
STQFA ^{PASI}	✓		171
STFSR ^D , STXFSR	✓		181
STPARTIALF			177
STSHORTF			178
ST{B H W X}	✓		164
ST{B H W X}A ^{PASI}	✓		165
ST{D}FR	✓	✓	174
ST{D}FR+	✓	✓	248
STTW ^D	✓		179
STTWA ^{D,PASI}	✓		180
SUB (SUBcc)			182
SUBC (SUBCcc)			182
SWAP ^D , SWAPA ^{D,PASI}	✓		183
SXAR{1 2}			184
TADDcc (TADDccTV ^D)			185
TSUBcc (TSUBccTV ^D)			185
Tcc			186
UDIV ^D (UDIVcc ^D)			187
UDIVX			146
UMUL ^D (UMULcc ^D)			188
WRASI			189
WRASR ^{PASR}			189
WRCCR			189
WRFPRS			189
WRGSR			189
WRPAUSE+			189

Instruction	HPC-ACE extension		Page
WRPCR ^{PCR}			189
WRPIC ^{PIC}			189
WRXAR			189
WRXASR			189
WRY ^D			189
XFILL ^N	✓		190
XNOR (XNORcc)			120
XOR (XORcc)			120

In SPARC64™ X / SPARC64™ X+, certain instructions are defined as the combination of a specific ASI number with one of the instructions LDDFA, LDTWA, LDXA, STDFA, STTWA, STXA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, or LDUWA. This combination is interpreted as a separate instruction, rather than an access to an alternate space. Table 7-4 shows these instructions. Refer to the instruction definition for details.

An empty column means that the combination of that ASI number with an instruction is not interpreted as a separate instruction. Those ASI numbers are invalid for LDDFA, LDTWA, LDXA, STDFA, STTWA, STXA, LDSBA, LDSHA, LDSWA, LDUBA, LDUHA, and LDUWA.

Table 7-4 Instructions defined as load/store to alternate space with special ASI.

		ASI number						
		16 ₁₆ , 17 ₁₆ , 1E ₁₆ , 1F ₁₆ , F0 ₁₆ , F1 ₁₆ , F8 ₁₆ , F9 ₁₆	E0 ₁₆ , E1 ₁₆	F2 ₁₆ , F3 ₁₆	22 ₁₆ , 23 ₁₆ , 27 ₁₆ , 2A ₁₆ , 2B ₁₆ , 2F ₁₆ , E2 ₁₆ , E3 ₁₆ , EA ₁₆ , EB ₁₆	C0 ₁₆ - C5 ₁₆ , C8 ₁₆ - CD ₁₆	D0 ₁₆ - D3 ₁₆ , D8 ₁₆ - DB ₁₆	E4 ₁₆ E5 ₁₆
LDDFA	i = 0	LDBLOCKF					LDSHORTF	
	i = 1							
STDFA	i = 0	STBLOCKF	STBLOCKF	XFILL ^N		STPARTIALF	STSHORTF	
	i = 1							
LDTWA ^{D, P_{ASI}}	i = 0				LDTXA ^N			
	i = 1							
STTWA ^{D, P_{ASI}}	i = 0			XFILL ^N	STBI ^N			
	i = 1							
LDXA	i = 0							
	i = 1							
STXA	i = 0			XFILL ^N	STBI ^N			
	i = 1							
LDSBA	i = 0							
	i = 1							
LDSHA	i = 0							
	i = 1							
LDSWA	i = 0							
	i = 1							
LDUBA	i = 0							
	i = 1							
LDUHA	i = 0							
	i = 1							
LDUWA	i = 0							
	i = 1							

7.1. ADD

Refer to Section 7.1 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	All	$XAR.v = 1$

7.2. Align Address

Refer to Section 7.5 in UA2011.

Note ALIGNADDR_LITTLE generates the opposite-endian byte ordering for a subsequent FALIGNDATA operation.

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	FPRS.fef = 0 or PSTATE.pef = 0
<i>illegal_action</i>	All	XAR.v = 1

7.4. Three-Dimensional Array Addressing

Refer to Section 7.8 in UA2011.

Exception	Target instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1

7.5. Byte Mask and Shuffle

Refer to Section 7.10 in UA2011.

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	FPRS.fef = 0 or PSTATE.pef = 0
<i>illegal_action</i>	All	XAR.v = 1

7.6. Branch on Integer Condition Codes (Bicc)

Refer to Section 7.9 in UA2011.

Note The Trap on Control Transfer feature is implemented on SPARC64™ X / SPARC64™ X+ (page 302).

Exception	Target instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1
<i>control_transfer_instruction</i>	Excluding BN	The branch is taken and PSTATE.tct = 1. The conditional branch BA is always taken.

7.7. Branch on Integer Condition Codes with Prediction (BPcc)

Refer to Section 7.11 in UA2011.

Note The Trap on Control Transfer feature is implemented on SPARC64™ X / SPARC64™ X+ (page 302).

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0. (cc0 = 1)
<i>illegal_action</i>	All	XAR.v = 1
<i>control_transfer_instruction</i>	Excluding BPN	The branch is taken and PSTATE.tct = 1. The conditional branch BA is always taken.

Related Branch on Integer Register with Prediction (BPr) (page 57)

7.8. Branch on Integer Register with Prediction (BPr)

Refer to Section 7.12 in UA2011.

Note The Trap on Control Transfer feature is implemented on SPARC64™ X / SPARC64™ X+ (page 302).

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	When any of the following are true • $rcond = 000_2$ or 100_2 • $iw<28> = 1$
<i>illegal_action</i>	All	$XAR.v = 1$
<i>control_transfer_instruction</i>	All	The branch is taken and $PSTATE.tct = 1$.

Related

Branch on Integer Condition Codes with Prediction (BPcc) (page 56)

7.9. Call and Link

Refer to Section 7.13 in UA2011.

Note When PSTATE.am = 1, the upper 32 bits of the PC are masked (set to 0) and written to R[15]. R[15] is updated immediately, and the delay instruction can use the modified value of R[15].

Note The Trap on Control Transfer feature is implemented in SPARC64™ X / SPARC64™ X+ (page 302).

Exception	Target instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1
<i>control_transfer_instruction</i>	All	PSTATE.tct = 1

Related JMPL (page 121)

7.10. Compare and Swap

Opcode	op3	Operation	HPC-ACE	Assembly Language Syntax
			Regs. SIMD	
CASA ^{PASI}	11 1100 ₂	Compare and Swap Word from Alternate Space	✓	<code>casa [reg_{rs1}] imm_{asi}, reg_{rs2}, reg_{rd}</code> <code>casa [reg_{rs1}] %asi, reg_{rs2}, reg_{rd}</code>
CASXA ^{PASI}	11 1110 ₂	Compare and Swap Extended Word from Alternate Space	✓	<code>casxa [reg_{rs1}] imm_{asi}, reg_{rs2}, reg_{rd}</code> <code>casxa [reg_{rs1}] %asi, reg_{rs2}, reg_{rd}</code>

Refer to Section 7.16 in UA2011.

The compare-and-swap instructions can be used with any of the following ASIs, subject to the privilege mode rules described for the *privileged_action* exception. Use of any other ASI with these instructions causes a *DAE_invalid_asi* exception.

ASIs valid for CASA and CASXA	
ASI_NUCLEUS	ASI_NUCLEUS_LITTLE
ASI_AS_IF_USER_PRIMARY	ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY	ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_REAL	ASI_REAL_LITTLE
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A reserved field is not 0. (i = 1 and iw<12:5> ≠ 0)
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>mem_address_not_aligned</i>	CASXA	When the address indicated by R[rs1] is not aligned on eight-byte boundary
	CASA	When the address indicated by R[rs1] is not aligned on four-byte boundary
<i>privileged_action</i>	All	PSTATE.priv = 0 and either of the following is true • i = 0 and ASI<7> = 0 • i = 1 and imm_asi<7> = 0
	All	PSTATE.priv = 1 and either of the following is true • i = 0 and 3016 ≤ ASI ≤ 7F16 • i = 1 and 3016 ≤ imm_asi ≤ 7F16
<i>VA_watchpoint</i>	All	Refer to 12.5.1.62.
<i>DAE_invalid_asi</i>	All	Refer to 12.5.1.5
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	Attempted access to non-cacheable space. Refer to 12.5.1.6
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7
<i>DAE_side_effect_page</i>	All	Refer to 12.5.1.9

7.12. Edge Handling Instructions

Refer to Section 7.23 in UA2011.

Exception	Target instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1

7.13. Edge Handling Instructions (noCC)

Refer to Section 7.24 in UA2011.

Exception	Target instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1

7.14. Convert Integer to Floating-Point

Opcode	opf	Operation	HPC-ACE		Assembly Language	
			Regs.	SIMD	Syntax	
FiT0s	0 1100 0100 ₂	Convert 32-bit Integer to Single	✓	✓	fitos	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FiT0d	0 1100 1000 ₂	Convert 32-bit Integer to Double	✓	✓	fitod	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FiT0q	0 1100 1100 ₂	Convert 32-bit Integer to Quad	✓		fitoq	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FxT0s	0 1000 0100 ₂	Convert 64-bit Integer to Single	✓	✓	fxtos	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FxT0d	0 1000 1000 ₂	Convert 64-bit Integer to Double	✓	✓	fxtod	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FxT0q	0 1000 1100 ₂	Convert 64-bit Integer to Quad	✓		fxtq	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.36 and Section 7.68 in UA2011.

Note Rounding is performed as specified by FSR.rd or GSR.irnd.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FiT0s, FiT0d, FxT0s, FxT0d	A <i>reserved</i> field is not 0.
		FiT0q, FxT0q	Always. For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are used for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		FiT0s, FiT0d, FxT0s, FxT0d	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
		FiT0q, FxT0q	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_ieee_754</i>	NX	FxT0s, FxT0d, FiT0s	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FqT0x, FqT0i	rs2<1> ≠ 0

7.15. Convert Between Floating-Point Formats

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FsT0d	0 1100 1001 ₂	Convert Single to Double	✓	✓	fstod	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FsT0q	0 1100 1101 ₂	Convert Single to Quad	✓		fstoq	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FdT0s	0 1100 0110 ₂	Convert Double to Single	✓	✓	fdtos	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FdT0q	0 1100 1110 ₂	Convert Double to Quad	✓		fdtoq	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FqT0s	0 1100 0111 ₂	Convert Quad to Single	✓		fqtos	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FqT0d	0 1100 1011 ₂	Convert Quad to Double	✓		fqtod	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.66 in UA2011.

Note Rounding is performed as specified by FSR.rd or GSR.irnd.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FsT0d, FdT0s	A <i>reserved</i> field is not 0.
		FsT0q, FdT0q, FqT0s, FqT0d	Always. For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		FsT0d, FdT0s	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
		FsT0q, FdT0q, FqT0s, FqT0d	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_ieee_754</i>	OF, UF, NX	FqT0s, FqT0d, FdT0s	Conforms to IEEE754.
	NV	All	F[rs2] is sNaN.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FsT0q, FdT0q	rd<1> ≠ 0
		FqT0s, FqT0d	rs2<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		FsT0d, FdT0s	Refer to Chapter 8.

Compatibility Note *fp_exception_other* (FSR.ftt = *invalid_fp_register*) conforms to UA2011. In JPS1, the *fp_exception_other* (FSR.ftt = *unimplemented_FPop*) exception was detected when executing quadruple-precision instructions.

7.16. Convert Floating-Point to Integer

Opcode	opf	Operation	HPC-ACE		Assembly Language	
			Regs.	SIMD	Syntax	
FsTOx	0 1000 0001 ₂	Convert Single to 64-bit Integer	✓	✓	fstox	<i>freg_{rs2}, freg_{rd}</i>
FdTOx	0 1000 0010 ₂	Convert Double to 64-bit Integer	✓	✓	fdtox	<i>freg_{rs2}, freg_{rd}</i>
FqTOx	0 0100 0011 ₂	Convert Quad to 64-bit Integer	✓		fqtox	<i>freg_{rs2}, freg_{rd}</i>
FsTOi	0 1101 0001 ₂	Convert Single to 32-bit Integer	✓	✓	fstoi	<i>freg_{rs2}, freg_{rd}</i>
FdTOi	0 1101 0010 ₂	Convert Double to 32-bit Integer	✓	✓	fdtoi	<i>freg_{rs2}, freg_{rd}</i>
FqTOi	0 1101 0011 ₂	Convert Quad to 32-bit Integer	✓		fqtoi	<i>freg_{rs2}, freg_{rd}</i>

Refer to Section 7.65 in UA2011.

Note The result is always rounded towards zero. FSR.rd and GSR.irnd are ignored.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FsTOx, FdTOx, FsTOi, FdTOi	A <i>reserved</i> field is not 0.
		FqTOx, FqTOi	Always. For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		FsTOx, FdTOx, FsTOi, FdTOi	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
		FqTOx, FqTOi	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_ieee_754</i>	NV, NX	All	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FqTOx, FqTOi	rs2<1> ≠ 0

Compatibility Note *fp_exception_other* (FSR.ftt = *invalid_fp_register*) conforms to UA2011. In JPS1, the *fp_exception_other* (FSR.ftt = *unimplemented_FPop*) exception was detected when executing quadruple precision instructions.

7.17. Floating-Point Absolute Value

Opcode	opf	Operation	HPC-ACE		Assembly Language	
			Regs.	SIMD	Syntax	
FABSS	0 0000 1001 ₂	Absolute value Single	✓	✓	fabss	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FABSd	0 0000 1010 ₂	Absolute value Double	✓	✓	fabsd	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FABSq	0 0000 1011 ₂	Absolute value Quad	✓		fabsq	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.25 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FABSS, FABSd	A <i>reserved</i> field is not 0.
	FABSq	Always. For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	FABSS, FABSd	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	FABSq	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	FABSq	When either of the following is true • rs2<1> ≠ 0 • rd<1> ≠ 0

7.18. Floating-Point Add and Subtract

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FADDs	0 0100 0001 ₂	Add Single	✓	✓	fadds	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FADDd	0 0100 0010 ₂	Add Double	✓	✓	faddd	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FADDq	0 0100 0011 ₂	Add Quad	✓		faddq	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSUBs	0 0100 0101 ₂	Subtract Single	✓	✓	fsubs	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSUBd	0 0100 0110 ₂	Subtract Double	✓	✓	fsubd	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSUBq	0 0100 0111 ₂	Subtract Quad	✓		fsubq	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.26 and Section 7.67 in UA2011.

Note Rounding is performed as specified by FSR.rd or GSR.irnd.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FADDs, FADDd, FSUBs, FSUBd	A <i>reserved</i> field is not 0.
		FADDq, FSUBq,	Always For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		FADDs, FADDd, FSUBs, FSUBd	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
		FADDq, FSUBq,	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_ieee_754</i>	OF, UF, NX, NV	All	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FADDq FSUBq	When any of the following are true • rs1<1> ≠ 0 • rs2<1> ≠ 0 • rd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		FADDs, FADDd, FSUBs, FSUBd	Refer to Chapter 8.

Compatibility Note *fp_exception_other* (FSR.ftt = *invalid_fp_register*) conforms to UA2011. In JPS1, the *fp_exception_other* (FSR.ftt = *unimplemented_FPop*) exception was detected when executing quadruple-precision instructions.

7.19. Align Data

Refer to Section 7.27 in UA2011.

Compatibility Note This instruction is referred to as “FALIGNDATAg” in UA2011.

Exception	Target instruction	Condition
<i>fp_disabled</i>	FALIGNDATA	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	FALIGNDATA	XAR.v = 1

7.20. Branch on Floating-Point Condition Codes (FBfcc)

Refer to Section 7.28 in UA2011.

Note The Trap on Control Transfer feature is implemented on SPARC64™ X / SPARC64™ X+ (page 302).

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1
<i>control_transfer_instruction</i>	Excluding FBN	The branch is taken and PSTATE.tct = 1. The conditional branch FBA is always taken.

7.21. Branch on Floating-Point Condition Code with Prediction (FBPfcc)

Refer to Section 7.29 in UA2011.

Note The Trap on Control Transfer feature is implemented on SPARC64™ X / SPARC64™ X+ (page 302).

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1
<i>control_transfer_instruction</i>	Excluding FBPN	When the branch is taken and PSTATE.tct = 1. The conditional branch FBPA is always taken.

7.22. Floating-Point Compare

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FCMPs	0 0101 0001 ₂	Compare Single	✓		fcmps %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FCMPd	0 0101 0010 ₂	Compare Double	✓		fcmpd %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FCMPq	0 0101 0011 ₂	Compare Quad	✓		fcmpq %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FCMPES	0 0101 0101 ₂	Compare Single and Exception if Unordered.	✓		fcmpes %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FCMPed	0 0101 0110 ₂	Compare Double and Exception if Unordered.	✓		fcmped %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FCMPEq	0 0101 0111 ₂	Compare Quad and Exception if Unordered.	✓		fcmpeq %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>

Refer to Section 7.31 in UA2011.

Note The “compare and cause exception if unordered” (FCMPES, FCMPed, and FCMPEq) instructions cause an *fp_exception_ieee_754* invalid (NV) exception if either operand is a sNaN or qNaN.

Note FCMP causes an *fp_exception_ieee_754* invalid (NV) exception if either operand is a sNaN.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FCMPs, FCMPd, FCMPES, FCMPed	A <i>reserved</i> field is not 0.
		FCMPq, FCMPEq	Always For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	When XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd ≠ 0
<i>fp_exception_ieee_754</i>	NV	All	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FCMPq, FCMPEq	When either of the following is true • rs1<1> ≠ 0 • rs2<1> ≠ 0

7.23. Floating-Point Conditional Compare to Register

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FCMPEQd	1 0110 0000 ₂	Fd[rs1] = Fd[rs2]	✓	✓	fcmpeqd	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPEQEd	1 0110 0010 ₂	Fd[rs1] = Fd[rs2] Exception if Unordered	✓	✓	fcmpeqed	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPLEEd	1 0110 0100 ₂	Fd[rs1] ≤ Fd[rs2] Exception if Unordered	✓	✓	fcmppeed	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPLTEd	1 0110 0110 ₂	Fd[rs1] < Fd[rs2] Exception if Unordered	✓	✓	fcmplteq	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPNEd	1 0110 1000 ₂	Fd[rs1] ≠ Fd[rs2]	✓	✓	fcmpned	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPNEEd	1 0110 1010 ₂	Fd[rs1] ≠ Fd[rs2] Exception if Unordered	✓	✓	fcmpneed	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPGTed	1 0110 1100 ₂	Fd[rs1] > Fd[rs2] Exception if Unordered	✓	✓	fcmpgted	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPGEEd	1 0110 1110 ₂	Fd[rs1] ≥ Fd[rs2] Exception if Unordered	✓	✓	fcmpgeed	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPEQs	1 0110 0001 ₂	Fs[rs1] = Fs[rs2]	✓	✓	fcmpeqs	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPEQEs	1 0110 0011 ₂	Fs[rs1] = Fs[rs2] Exception if Unordered	✓	✓	fcmpeques	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPLEEs	1 0110 0101 ₂	Fs[rs1] ≤ Fs[rs2] Exception if Unordered	✓	✓	fcmplees	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPLTEs	1 0110 0111 ₂	Fs[rs1] < Fs[rs2] Exception if Unordered	✓	✓	fcmpltes	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPNEs	1 0110 1001 ₂	Fs[rs1] ≠ Fs[rs2]	✓	✓	fcmpnes	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPNEEs	1 0110 1011 ₂	Fs[rs1] ≠ Fs[rs2] Exception if Unordered	✓	✓	fcmpnees	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPGTes	1 0110 1101 ₂	Fs[rs1] > Fs[rs2] Exception if Unordered	✓	✓	fcmpgtes	freg _{rs1} , freg _{rs2} , freg _{rd}
FCMPGEEs	1 0110 1111 ₂	Fs[rs1] ≥ Fs[rs2] Exception if Unordered	✓	✓	fcmpgees	freg _{rs1} , freg _{rs2} , freg _{rd}

10 ₂	rd			op3 = 11 0110 ₂	rs1			opf			rs2		
31 30 29	25 24			19 18	14 13			5 4			0		

Description The above instructions compare the values in the floating-point registers specified by F[rs1] and F[rs2]. If the condition specified by the instruction is met, then the floating-point register specified by F[rd] is written entirely with ones. If the condition is not met, then F[rd] is written entirely with zeroes.

When the source operands are sNaN or qNaN, generated exceptions and instruction results are described below. The “exception” column indicates the value set in FSR.cexc when an *fp_exception_ieee_754* exception occurs. The “F[rd]” column indicates the value stored in F[rd] when no exception occurs.

Instruction	SNaN		QNaN	
	Exception	F[rd]	Exception	F[rd]
FCMPGTE{s d}, FCMPLTE{s d}, FCMPGEE{s d}, FCMPLEE{s d}	NV	all0	NV	all0
FCMPEQE{s d}	NV	all0	NV	all0
FCMPNEE{s d}	NV	all1	NV	all1
FCMPEQ{s d}	NV	all0	—	all0
FCMPNE{s d}	NV	all1	—	all1

Programming Note These instructions can be efficiently used with FSELMOV{s|d}, STFR, STDFR, and the VIS logical instructions.

Exception		Target instruction	Condition
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	When XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>fp_exception_ieee_754</i>	NV	All	Unordered

7.24. SIMD Compare (comforms to UA2011)

Refer to Section 7.54 and Section 7.55 in UA2011.

Refer to Section 7.127 regarding the SIMD Unsigned Compare instructions,
FPCMPU{EQ|NE|LE|GT}8.

Compatibility Note There are three kinds of SIMD compare instructions.

1) SIMD compare instructions conforming to UA2011 (described in this section and 7.127)

- The comparison result is stored in the least significant bits of R[rd].
- Source (Fd[rs1] and Fd[rs2]) and destination (R[rd]) registers cannot be extended by XAR.
- The instruction mnemonic is FPCMP*{8|16|32} or FUCMP*{8|16|32}.
- FPCMP{NE|EQ}8 are not defined on SPARC64™ X / SPARC64™ X+.

2) SIMD compare instructions as implemented on SPARC64™ X / SPARC64™ X+ (described in 7.123)

- The comparison result is stored in the most significant bits of Fd[rd].
- Source (Fd[rs1] and Fd[rs2]) registers can be extended by XAR. The destination (Fd[rd]) register can be extended by XAR, but only basic floating-point registers (Fd[0] – Fd[126]) can be specified. HPC-ACE SIMD operations are not supported.
- The instruction mnemonic is FCMP*{8|16|32|64}X or FUCMP*{8|16|32|}X.
- FCMP{NE|EQ}{8|16|32|}X are not defined on SPARC64™ X / SPARC64™ X+.

3) SIMD compare instructions as implemented on SPARC64™ X+ (described in 7.139)

- The comparison result is stored in the most significant bits of Fd[rd].
 - Source (Fd[rs1] and Fd[rs2]) and destination (Fd[rd]) registers can be extended by XAR. HPC-ACE SIMD operations are supported.
 - The instruction mnemonic is FPCMP*{8|16|32|64}X or FPCMPU*{8|16|32|64}X.
 - FPCMP{NE|EQ}{8|16|32|64}X are not defined on SPARC64™ X / SPARC64™ X+.
-

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

7.25. Floating-Point Divide

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FDIVs	0 0100 1101 ₂	Divide Single	Only basic : rd.		fdivs	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FDIVd	0 0100 1110 ₂	Divide Double	Only basic : rd.		fdivd	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FDIVq	0 0100 1111 ₂	Divide Quad	Only basic : rd.		fdivq	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>

Refer to Section 7.32 in UA2011.

Note Rounding is performed as specified by FSR.rd or GSR.irnd field.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FDIVq	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<2:1> ≠ 0
<i>fp_exception_ieee_754</i>	OF, UF, DZ, NV, NX	All	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FDIVq	When either of the following is true • rs1<1> ≠ 0 • rs2<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		FDIVs, FDIVd	Refer to Chapter 8.

7.26. Floating-Point Exponential Auxiliary

Instruction	opf	Operation	HPC-ACE		Assembly Language	
			Regs	SIMD	Syntax	
FEXPAd	1 0111 1100 ₂	Exponential Auxiliary	✓	✓	fexpad	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0110 ₂	—	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description The FEXPAd instruction accelerates the series approximation of the exponential function $\exp(x)$. A table lookup is performed based on the lower bits of Fd[rs2], and the result is stored in Fd[rd].

$$\text{Fd}[\text{rd}] = 1'b0 :: \text{Fd}[\text{rs2}]<16:6> :: \text{Texp}[\text{Fd}[\text{rs2}]<5:0>]$$

If the FEXPAd instruction is executed, FSR.cexc and FSR.ftt are set to 0. FSR.aexc is not updated.

Texp is table of 64 entries that maintains the 52-bit significand of a double-precision number.

Table 7-5 Table of Texp [k]

k	Texp[k]	k	Texp[k]	k	Texp[k]	k	Texp[k]
0	0x0000000000000000	16	0x306FE0A31B715	32	0x6A09E667F3BCD	48	0xAE89F995AD3AD
1	0x02C9A3E778061	17	0x33C08B26416FF	33	0x6DFB23C651A2F	49	0xB33A2B84F15FB
2	0x059B0D3158574	18	0x371A7373AA9CB	34	0x71F75E8EC5F74	50	0xB7F76F2FB5E47
3	0x0874518759BC8	19	0x3A7DB34E59FF7	35	0x75FEB564267C9	51	0xBCC1E904BC1D2
4	0x0B5586CF9890F	20	0x3DEA64C123422	36	0x7A11473EB0187	52	0xC199BDD85529C
5	0x0E3EC32D3D1A2	21	0x4160A21F72E2A	37	0x7E2F336CF4E62	53	0xC67F12E57D14B
6	0x11301D0125B51	22	0x44E086061892D	38	0x82589994CCE13	54	0xCB720DCEF9069
7	0x1429AAEA92DE0	23	0x486A2B5C13CD0	39	0x868D99B4492ED	55	0xD072D4A07897C
8	0x172B83C7D517B	24	0x4BFDAD5362A27	40	0x8ACE5422AA0DB	56	0xD5818DCFBA487
9	0x1A35BEB6FCB75	25	0x4F9B2769D2CA7	41	0x8F1AE99157736	57	0xDA9E603DB3285
10	0x1D4873168B9AA	26	0x5342B569D4F82	42	0x93737B0CDC5E5	58	0xDFC97337B9B5F
11	0x2063B88628CD6	27	0x56F4736B527DA	43	0x97D829FDE4E50	59	0xE502EE78B3FF6
12	0x2387A6E756238	28	0x5AB07DD485429	44	0x9C49182A3F090	60	0xEA4AFA2A490DA
13	0x26B4565E27CDD	29	0x5E76F15AD2148	45	0xA0C667B5DE565	61	0xEFA1BEE615A27
14	0x29E9DF51FDEE1	30	0x6247EB03A5585	46	0xA5503B23E255D	62	0xF50765B6E4540
15	0x2D285A6E4030B	31	0x6623882552225	47	0xA9E6B5579FDBF	63	0xFA7C1819E90D8

FEXPAd does not treat Fd[rs2] as a floating-point number. Even if Fd[rs2] is NaN, it is not treated as a special value.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	When XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.27. FEXPAND

Refer to Section 7.33 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

7.28. Flush Instruction Memory

Refer to Section 7.38 in UA2011.

Note The specifications of the FLUSH instruction in JPS1 and UA2011 are slightly different. Differences between these specifications are noted. SPARC64™ X / SPARC64™ X+ mainly conform to the JPS1 specification.

Compatibility Note In both JPS1 and UA2011, the least significant 3 address bits are ignored, but JPS1 expects software to specify 0 for the least significant 2 address bits.

Note SPARC64™ X / SPARC64™ X+ guarantee consistency between the instruction cache and the data cache, so even if a FLUSH instruction is not executed, the values in both caches eventually become consistent. Therefore, the address specified by the instruction is not used, and exceptions related to the address fields are not generated. However, SPARC64™ X / SPARC64™ X+ execute instructions out of order, so an instruction might be sent the pipeline before the instruction cache is updated. Use the FLUSH instruction on SPARC64™ X / SPARC64™ X+ to guarantee consistency with instructions in the pipeline.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	XAR.v = 1

7.29. Flush Register Windows

Refer to Section 7.39 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	XAR.v = 1
<i>spill_n_normal</i>	
<i>spill_n_other</i>	

7.30. Floating-Point Multiply-Add/Subtract

Instruction	var	size	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FMADDs	00 ₂	01 ₂	Multiply-Add Single	✓	✓	fmadds <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FMADDd	00 ₂	10 ₂	Multiply-Add Double	✓	✓	fmaddd <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FMSUBs	01 ₂	01 ₂	Multiply-Subtract Single	✓	✓	fmsubs <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FMSUBd	01 ₂	10 ₂	Multiply-Subtract Double	✓	✓	fmsubd <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FNMSUBs	10 ₂	01 ₂	Negative Multiply-Subtract Single	✓	✓	fnmsubs <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FNMSUBd	10 ₂	10 ₂	Negative Multiply-Subtract Double	✓	✓	fnmsubd <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FNMADDs	11 ₂	01 ₂	Negative Multiply-Add Single	✓	✓	fnmadds <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FNMADDd	11 ₂	10 ₂	Negative Multiply-Add Double	✓	✓	fnmaddd <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>

10 ₂	rd	op3 = 110111 ₂	rs1	rs3	var	size	rs2
31 30 29	25 24	19 18	14 13	9 8	7 6	5 4	0

SPARC64™ X / SPARC64™ X+ use IMPDEP2 opcodes to implement the Floating-Point Multiply-Add/Subtract (FMA) instructions. FMA instructions support SIMD execution, which is an HPC-ACE feature. This section first describes the behavior of non-SIMD FMA instructions and then explains the use of FMA instructions with HPC-ACE features.

Instruction	Operation
Multiply-Add	$F[rd] = F[rs1] \times F[rs2] + F[rs3]$
Multiply-Subtract	$F[rd] = F[rs1] \times F[rs2] - F[rs3]$
Negative Multiply-Subtract	$F[rd] = - (F[rs1] \times F[rs2] - F[rs3])$
Negative Multiply-Add	$F[rd] = - (F[rs1] \times F[rs2] + F[rs3])$

Non-SIMD execution

FMADD multiplies the floating-point registers specified by F[rs1] and F[rs2], adds the product to the floating-point register specified by F[rs3], and writes the result into the floating-point register specified by F[rd].

FMSUB multiplies the floating-point registers specified by F[rs1] and F[rs2], subtracts the floating-point register specified by F[rs3] from the product, and writes the result into the floating-point register specified by F[rd].

FNMADD multiplies the floating-point registers specified by F[rs1] and F[rs2], subtracts the floating-point register specified by F[rs3] from the product, negates this value, and writes the result into the floating-point register specified by F[rd].

FNMSUB multiplies the floating-point registers specified by F[rs1] and F[rs2], adds the product to the floating-point register specified by F[rs3], negates this value, and writes the result into the floating-point register specified by F[rd].

An FMA instruction is processed as a fused multiply-add/subtract operation. That is, the result of the multiply operation is not rounded and has infinite precision. For FM{ADD|SUB}{s|d}, rounding is done after addition/subtraction. For FNM{ADD|SUB}{s|d}, rounding is done after negation. Thus, at most one rounding error can occur.

Table 7-6 describes how SPARC64™ X / SPARC64™ X+ handle traps generated by

Floating-point Multiply-Add/Subtract instructions. If the multiply detects a denormal source operand while FSR.ns = 0 and the invalid exception trap is not masked, the execution of the instruction is aborted, the invalid (NV) exception bit is set in FSR.cexc, and a trap for the exception is generated. FSR.aexc, is not updated.

Addition/subtraction is only done if the multiply does not generate an invalid exception trap. If addition/subtraction generates an IEEE754 exception trap, the exception is recorded in FSR.cexc. FSR.aexc is not updated. If addition/subtraction detects an IEEE754 exception that is masked, the exception is recorded in FSR.cexc and accumulated in FSR.aexc. The conditions that cause an *unfinished_FPop* exception for Floating-point Multiply-Add/Subtract instructions are the same as the conditions for the FMUL instruction for source operands F[rs1] and F[rs2], and the same as the FADD instruction for source operand F[rs3] and the destination F[rd].

Table 7-6 IEEE754 Exceptions for Floating-Point Multiply-Add/Subtract Instructions

FMUL FADD	IEEE754 trap (NV or NX only) —	No trap IEEE754 trap	No trap No trap
cexc	FMUL exception	FADD exception	FADD exception, masked
aexc	No change	No change	Logical OR of cexc (above) and aexc

Table 7-7 and Table 7-8 describe the values of cexc for various conditions when exceptions are masked by FSR.tem and do not generate a trap. The IEEE exceptions are abbreviated as underflow (uf), overflow (of), invalid operation (inv), and inexact (nx).

Table 7-7 Masked exceptions in cexc when FSR.ns = 0

		FADD			
		None	nx	of nx	inv
FMUL	none	None	nx	of nx	inv
	inv	inv	—	—	inv

Table 7-8 Masked exceptions in cexc when FSR.ns = 1

		FADD				
		None	nx	of nx	uf nx	inv
FMUL	none	None	nx	of nx	uf nx	inv
	inv	inv	—	—	—	inv
	nx	nx	nx	of nx	uf nx	inv nx

In the above tables, conditions marked “—” do not occur.

Programming Note The Floating-Point Multiply-Add/Subtract instructions are implemented using the SPARC V9 IMPDEP2 opcode space. These instructions are specific to SPARC64™ X / SPARC64™ X+ and cannot be used in any programs that will be executed on another SPARC V9 processor.

The results of FMADD{s|d}, FNMADD{s|d}, FMSUB{s|d}, and FNMSUB{s|d} differ from the specification of UA2011 when (1) either F[rs1] or F[rs2] is SNaN or (2) F[rs3] is QNaN.

When the result is exactly 0 and the rounding mode is 3 (towards $-\infty$), the sign of the result is negative. When the result is exactly 0 and the rounding mode is not 3, the sign of the result is positive. The lowercase "n" in QNaNn, QSNaNn and SNaNn used in the following tables refer to the operand number (F[rsn]).

Table 7-9 FMADD{s|d}

		F[rs3]							QNaN3	SNaN3
		$-\infty$	-N	-0	+0	+N	$+\infty$			
F[rs1] × F[rs2]	$-\infty$	$-\infty$						NV dQNaN	$-\infty$ QNaN3	NV QSNaN3
	-N	$\frac{F[rs1] \times F[rs2]}{+ F[rs3]}$	$\frac{Fd[rs1] \times Fd[rs2]}{+0^{ii}}$		$\frac{F[rs1] \times F[rs2]}{+ F[rs3]}^i$	$+\infty$				
	-0	$F [rs3]$	$-\infty$ -0	$-\infty$ +0 ⁱⁱ	$F[rs3]$					
	+0		$-\infty$ +0 ⁱⁱ	$-\infty$ +0						
	+N	$\frac{F[rs1] \times F[rs2]}{+ F[rs3]}^i$	$\frac{Fd[rs1] \times Fd[rs2]}{+0^{ii}}$		$\frac{F[rs1] \times F[rs2]}{+ F[rs3]}$					
	$+\infty$	NV dQNaN								
	QNaN1	$-\infty$ QNaN1								
	QNaN2	$-\infty$ QNaN2								
	QNaN ($\pm 0 \times \pm \infty$)	NV dQNaN					NV QNaN3			
	QSNaN1	NV QSNaN1								
QSNaN2	NV QSNaN2									

ⁱ When the result is 0, footnote (ii) applies.

ⁱⁱ When the rounding mode is towards $-\infty$, the result is -0 .

Table 7-10 FNMADD{s|d}

		F[rs3]							QNaN3	SNaN3
		$-\infty$	-N	-0	+0	+N	$+\infty$			
F[rs1] × F[rs2]	$-\infty$	$\overline{\quad}$ $+\infty$						NV dQNaN	$\overline{\quad}$ QNaN3	NV QNaN3
	-N	$\overline{\quad}$ $-(F[rs1] \times F[rs2])$ $- F[rs3]$	$\overline{\quad}$ $-(Fd[rs1] \times Fd[rs2])$		$\overline{\quad}$ $-(F[rs1] \times F[rs2])$ $- F[rs3]^{iii}$					
	-0	$\overline{\quad}$ $-F[rs3]$	$\overline{\quad}$ $+0$	$\overline{\quad}$ $+0^{iv}$	$\overline{\quad}$ $-F[rs3]$					
	+0	$\overline{\quad}$ $+0^{iv}$	$\overline{\quad}$ -0							
	+N	$\overline{\quad}$ $-(F[rs1] \times F[rs2])$ $- F[rs3]^{iii}$	$\overline{\quad}$ $-(Fd[rs1] \times Fd[rs2])$		$\overline{\quad}$ $-(F[rs1] \times F[rs2])$ $- F[rs3]$					
	$+\infty$	NV dQNaN	$\overline{\quad}$ $-\infty$							
	QNaN1	$\overline{\quad}$ QNaN1								
	QNaN2	$\overline{\quad}$ QNaN2								
	QNaN ($\pm 0 \times$ $\pm \infty$)	NV dQNaN							NV QNaN3	
	QNaN1	NV QNaN1								
QNaN2	NV QNaN2									

ⁱⁱⁱ When the result is 0, footnote (iv) applies.

^{iv} When the rounding mode is towards $-\infty$, the result is -0 .

Table 7-11 FMSUB{s|d}

		F[rs3]							
		$-\infty$	-N	-0	+0	+N	$+\infty$	QNaN3	SNaN3
F[rs1] × F[rs2]	$-\infty$	NV dQNaN	$-\infty$					$-\infty$ QNaN3	NV QNaN3
	-N	$\frac{F[rs1] \times F[rs2]}{F[rs3]^v}$	$\frac{Fd[rs1] \times Fd[rs2]}{F[rs3]}$		$\frac{F[rs1] \times F[rs2]}{F[rs3]}$				
	-0	$-F[rs3]$	$+\text{0}^{vi}$	-0	$-F[rs3]$				
	+0		$+\text{0}$	$+\text{0}^{vi}$					
	+N	$\frac{F[rs1] \times F[rs2]}{F[rs3]}$	$\frac{Fd[rs1] \times Fd[rs2]}{F[rs3]}$		$\frac{F[rs1] \times F[rs2]}{F[rs3]^v}$				
	$+\infty$	$+\infty$					NV dQNaN		
	QNaN1	$QNaN1$							
	QNaN2	$QNaN2$							
	QNaN ($\pm 0 \times \pm \infty$)	NV dQNaN							
	QNaN1	NV QNaN1							
	QNaN2	NV QNaN2							

^v When the result is 0, footnote (vi) applies.

^{vi} When the rounding mode is towards $-\infty$, the result is -0 .

Table 7-12 FNMSUB{s|d}

		F[rs3]						QNaN3	SNaN3	
		$-\infty$	-N	-0	+0	+N	$+\infty$			
F[rs1] × F[rs2]	$-\infty$	NV dQNaN	$-\infty$ $+\infty$						$-\infty$ QNaN3	NV QNaN3
	-N		$-(F[rs1] \times F[rs2]) + F[rs3]^{vii}$	$-(Fd[rs1] \times Fd[rs2])$		$-(F[rs1] \times F[rs2]) + F[rs3]$				
	-0		$F[rs3]$	$+0^{viii}$	$+0$	$F[rs3]$				
	+0			-0	$+0^{viii}$					
	+N		$-(F[rs1] \times F[rs2]) + F[rs3]$	$-(Fd[rs1] \times Fd[rs2])$		$-(F[rs1] \times F[rs2]) + F[rs3]^{vii}$				
	$+\infty$	$-\infty$	NV dQNaN							
	QNaN1	$-\infty$ QNaN1								
	QNaN2	$-\infty$ QNaN2								
	QNaN ($\pm 0 \times \pm \infty$)	NV dQNaN						NV QNaN3		
	QNaN1	NV QNaN1								
QNaN2	NV QNaN2									

SIMD execution In SPARC64™ X / SPARC64™ X+, the basic and extended operations of a SIMD instruction are executed independently. Because the basic operation uses registers in the range Fd[0] – Fd[126], the operation always sets the most significant bits of XAR.urs1, XAR.urs2, XAR.urs3, and XAR.urd to 0 (page 35). This restriction is relaxed for SIMD FMA instructions, so that operations that refer to both basic and extended registers can be executed.

Note The above limitation for SIMD instructions only applies when XAR.simd = 1. When XAR.simd = 0, rs1, rs2, rs3, and rd can use any of the floating-point registers.

For a SIMD FMA instruction, rs1 and rs2 can specify any of the floating-point registers Fd[2n] (n = 0 – 63, 128 – 191). When the basic operation specifies an extended register, the extended operation uses the corresponding basic register. That is, the basic operation uses registers Fd[2n] (n = 0 – 63, 128 – 191), and the extended operation uses Fd[(2n + 256) mod 512] (n = 0 – 63, 128 – 191).

The limitations for rs3 and rd are the same as for other SIMD instructions. The basic operation must use registers Fd[0] – Fd[126], and the extended operation must use Fd[256] – Fd[382]. That is, urs3<2> and urd<2> are never used to specify registers. SIMD

^{vii} When the result is 0, footnote (viii) applies.

^{viii} When the rounding mode is towards $-\infty$, the result is $-\infty$.

FMA instructions use these bits to specify additional execution options; these bits should be 0 for all other SIMD instructions. When $\text{urs3}<2> = 1$, the register(s) specified by rs1 is used for both basic and extended operations. When $\text{urd}<2> = 1$, the sign of the product for the extended operation is reversed.

The meanings of XAR.urs1 , XAR.urs2 , XAR.urs3 , and XAR.urd for a SIMD FMA instruction are summarized below:

- $\text{XAR.urs1}<2>$ $\text{rs1}<8>$ for the basic operation, $\neg\text{rs1}<8>$ for the extended operation
- $\text{XAR.urs2}<2>$ $\text{rs2}<8>$ for the basic operation, $\neg\text{rs2}<8>$ for the extended operation
- $\text{XAR.urs3}<2>$ specifies whether the extended operation uses $\text{rs1}<8>$ or $\neg\text{rs1}<8>$
- $\text{XAR.urd}<2>$ specifies whether the sign of the product is reversed for the extended operation

The $\text{rs1}<8>$ bit described above is a bit in the decoded HPC-ACE register number for a double-precision register. Refer to Figure 5-1 (page 24) for details.

Table 7-13 shows these SIMD operations in more details. See Table 7-14 for the notation used in Table 7-13.

Table 7-13 SIMD operations

Instruction	Basic operation	Extended operation
Fmadd	$\text{frdb} \leftarrow \text{frs1} \times \text{frs2} + \text{frs3b}$	$\text{frde} \leftarrow (-1)^n \times (c ? \text{frs1} : \text{frs1i}) \times \text{frs2i} + \text{frs3e}$
Fmsub	$\text{frdb} \leftarrow \text{frs1} \times \text{frs2} - \text{frs3b}$	$\text{frde} \leftarrow (-1)^n \times (c ? \text{frs1} : \text{frs1i}) \times \text{frs2i} - \text{frs3e}$
Fnmsub	$\text{frdb} \leftarrow -(\text{frs1} \times \text{frs2} - \text{frs3b})$	$\text{frde} \leftarrow -((-1)^n \times (c ? \text{frs1} : \text{frs1i}) \times \text{frs2i} - \text{frs3e})$
Fnmadd	$\text{frdb} \leftarrow -(\text{frs1} \times \text{frs2} + \text{frs3b})$	$\text{frde} \leftarrow -((-1)^n \times (c ? \text{frs1} : \text{frs1i}) \times \text{frs2i} + \text{frs3e})$

Table 7-14 Notation used in Table 7-13

$\text{frs1}:$ $\text{urs1}<2:0>::\text{rs1}<5:1>::1'b0$	$\text{frs1i}:$ $\neg\text{urs1}<2>::\text{urs1}<1:0>::\text{rs1}<5:1>::1'b0$
$\text{frs2}:$ $\text{urs2}<2:0>::\text{rs2}<5:1>::1'b0$	$\text{frs2i}:$ $\neg\text{urs2}<2>::\text{urs2}<1:0>::\text{rs2}<5:1>::1'b0$
$\text{frs3b}:$ $1'b0::\text{urs3}<1:0>::\text{rs3}<5:1>::1'b0$	$\text{frs3e}:$ $1'b1::\text{urs3}<1:0>::\text{rs3}<5:1>::1'b0$
$\text{frdb}:$ $1'b0::\text{urd}<1:0>::\text{rd}<5:1>::1'b0$	$\text{frde}:$ $1'b1::\text{urd}<1:0>::\text{rs3}<5:1>::1'b0$
$c:$ $\text{urs3}<2>$	
$n:$ $\text{urd}<2>$	

Example 1: Multiplication of complex numbers

$$(a1 + i \times b1) \times (a2 + i \times b2) = (a1 \times a2 - b1 \times b2) + i \times (a1 \times b2 + a2 \times b1)$$

```

/*
 * X: address of source complex number
 * Y: address of source complex number
 * Z: address of destination complex number
 */

/* Set up registers */
sxa2
ldd,s    [X], %f0 /* %f0: a1, %f256: b1 */
ldd,s    [Y], %f2 /* %f2: a2, %f258: b2 */
sxa1
fzero,s  %f4 /* clear destination registers */

/* Perform calculation */
sxa2
fmadd,snc %f256, %f258, %f4, %f4
/* %f4 := -%f256 * %f258 - %f4 */
/* %f260 := %f256 * %f2 - %f260 */
fmadd,sc %f0, %f2, %f4, %f4

```

```

/* %f4      := %f0 * %f2 + %f4 */
/* %f260    := %f0 * %f258 + %f260 */

/* Store results */.
sxa1
std,s      %f4, [Z]

```

Example 2: 2x2 matrix multiplication

```

/*
 * A: address of source matrix : a11, a12, a21, a22
 * B: address of source matrix : b11, b12, b21, b22
 * C: address of destination matrix : c11, c12, c21, c22
 */

/* Set up registers */
sxa2
ldd,s      [A], %f0 /* %f0: a11, %f256: a12 */
ldd,s      [A+16], %f2 /* %f2: a21, %f258: a22 */
sxa2
ldd,s      [B], %f4 /* %f4: b11, %f260: b12 */
ldd,s      [B+16], %f6 /* %f6: b21, %f262: b22 */
sxa2
fzero,s%f8 /* %f8: c11, %f264: c12 */
fzero,s%f10 /* %f10: c21, %f266: c22 */

/* Perform calculation */
sxa2
fmadd,sc %f0, %f4, %f8, %f8
/* %f8 := %f0 * %f4 + %f8 */
/* %f264 := %f0 * %f260 + %f264 */
fmadd,sc %f256, %f6, %f8, %f8
/* %f8 := %f256 * %f6 + %f8 */
/* %f264 := %f256 * %f262 + %f264 */
sxa2
fmadd,sc %f2, %f4, %f10, %f10
/* %f10 := %f2 * %f4 + %f10 */
/* %f266 := %f2 * %f260 + %f266 */
fmadd,sc %f258, %f6, %f10, %f10
/* %f10 := %f258 * %f6 + %f10 */
/* %f266 := %f258 * %f262 + %f266 */
/* Store results */.
sxa2
std,s      %f8, [Z]
std,s      %f10, [Z+16]

```

Exception		Target instruction	Condition
<i>illegal_instruction</i>		All	size = 11 ₂ and var ≠ 11 ₂ (Reserved for quadruple-precision FMA instructio)
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>			XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<1> ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_ieee_754</i>	NV, NX, OF, UF	All	
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		All	Refer to Chapter 8.

7.31. Floating-Point Minimum and Maximum

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FMAXd	1 0111 0000 ₂	Select Maximum Double	✓	✓	fmaxd	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMAXs	1 0111 0001 ₂	Select Maximum Single	✓	✓	fmaxs	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMIND	1 0111 0010 ₂	Select Minimum Double	✓	✓	fmind	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMINS	1 0111 0011 ₂	Select Minimum Single	✓	✓	fmins	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0110 ₂	rs1	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description FMAX{s | d} compares the values in the floating-point registers specified by F[rs1] and F[rs2]. If F[rs1] > F[rs2], then F[rs1] is written to the floating-point register specified by F[rd]. Otherwise, F[rs2] is written to F[rd].

FMIN{s | d} compares the values in the floating-point registers specified by F[rs1] and F[rs2]. If F[rs1] < F[rs2], then F[rs1] is written to the floating-point register specified by F[rd]. Otherwise, F[rs2] is written to F[rd].

FMIN and FMAX ignore the sign of a zero value. When the value of F[rs1] is +0 or −0 and the value of F[rs2] is +0, −0, the value of F[rs2] is written to the destination register.

When one of the source operands is QNaN and the other operand is neither QNaN nor SNaN, the value that is not QNaN is stored in F[rd].

Note Unlike other floating-point instructions, FMIN and FMAX do not propagate NaN.

When one or both of the source operands are SNaN, or both of the source operands are QNaN, the value defined in Table B-1 of JPS1 Commonality is written to F[rd]. Furthermore, when one of the source operands is QNaN or SNaN, SPARC64™ X / SPARC64™ X+ detect an *fp_exception_ieee_754* exception.

Table 7-15 FMIN{s|d} and FMAX{s|d}

		F[rs2]						
		−∞	−Fn	−0	+0	+Fn	+∞	QNaN
F[rs1]	−∞	— min(F[rs1], F[rs2]), or max(F[rs1], F[rs2])					NV F[rs1]	NV QNaN2
	−Fn							
	−0							
	+0							
	+Fn							
	+∞							
	QNaN	NV F[rs2]						
SNaN	NV QNaN1							

Exception		Target instruction	Condition
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	When XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>fp_exception_ieee_754</i>	NV	All	Unordered

7.32. Floating-Point Move

Instruction	opf	Operation	HPC-ACE		Assembly Language	
			Regs	SIMD	Syntax	
FMOV _s	0 0000 0001 ₂	Move Single	✓	✓	fmovs	<i>freg_{rs2}, freg_{rd}</i>
FMOV _d	0 0000 0010 ₂	Move Double	✓	✓	fmovd	<i>freg_{rs2}, freg_{rd}</i>
FMOV _q	0 0000 0011 ₂	Move Quad	✓		fmovq	<i>freg_{rs2}, freg_{rd}</i>

10 ₂	rd	op3 = 11 0100 ₂	—	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Refer to Section 7.42 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FMOV _s , FMOV _d	A <i>reserved</i> field is not 0.
	FMOV _q	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	FMOV _s , FMOV _d	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	FMOV _q	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	FMOV _q	When either of the following is true • rs2<1> ≠ 0 • rd<1> ≠ 0

7.33. Move Floating-Point Register on Condition (FMOVcc)

Refer to Section 7.43 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FMOV{S D}icc, FMOV{S D}xcc, FMOV{S D}fcc	A reserved field is not 0.
	FMOVQicc, FMOVQxcc, FMOVQfcc	Always For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
	—	When either of the following is true • opf_cc = 101₂ • opf_cc = 111₂
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	FMOVQicc, FMOVQxcc, FMOVQfcc	When either of the following is true • rs2<1> ≠ 0 • rd<1> ≠ 0

7.34. Move Floating-Point Register on Integer Register Condition (FMOVR)

Refer to Section 7.44 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FMOVR{s d}Z, FMOVR{s d}LEZ, FMOVR{s d}LZ, FMOVR{s d}NZ, FMOVR{s d}GZ, FMOVR{s d}GEZ	A reserved field is not 0.
	FMOVRqZ, FMOVRqLEZ, FMOVRqLZ, FMOVRqNZ, FMOVRqGZ, FMOVRqGEZ	Always For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
	—	When any of the following are true • rcond = 0002 • rcond = 100₂ • opf_low : excluding 0 01012, 0 01102, and 0 01112
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	FMOVRqZ, FMOVRqLEZ, FMOVRqLZ, FMOVRqNZ, FMOVRqGZ, FMOVRqGEZ	When either of the following is true • rs2<1> ≠ 0 • rd<1> ≠ 0

7.35. Partitioned Multiply Instructions

Refer to Section 7.45 in UA2011.

Exception	Target instruction	Condition
<i>fp_disabled</i>	all	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	all	XAR.v = 1

7.36. Floating-Point Multiply

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FMULs	0 0100 1001 ₂	Multiply Single	✓	✓	fmuls	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMULd	0 0100 1010 ₂	Multiply Double	✓	✓	fmuld	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMULq	0 0100 1011 ₂	Multiply Quad	✓		fmulq	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FsMULd	0 0110 1001 ₂	Multiply Single to Double	✓	✓	fsmuld	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FdMULq	0 0110 1110 ₂	Multiply Double to Single	✓		fdmulq	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.46 in UA2011.

Note For FMUL{s|d|q}, rounding is performed as specified by FSR.rd or GSR.irnd.

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FMULq, FdMULq	Always. For these instructions, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		FMULs, FMULd, FsMULd	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
		FMULq, FdMULq	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_ieee_754</i>	NV	All	Conforms to IEEE754.
	OF, UF, NX	FMULs, FMULd, FMULq	
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)		FMULq	When any of the following are true • rs1<1> ≠ 0 • rs2<1> ≠ 0 • rd<1> ≠ 0
		FdMULq	rd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		FMULs, FMULd, FsMULd	Refer to Chapter 8.

7.37. Floating-Point Negative

Instruction	opf	Operation	HPC-ACE		Assembly Language	
			Regs.	SIMD	Syntax	
FNEGs	0 0000 0101 ₂	Negate Single	✓	✓	fnegs	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FNEGd	0 0000 0110 ₂	Negate Double	✓	✓	fnegd	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FNEGq	0 0000 0111 ₂	Negate Quad	✓		fnegq	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.48 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FNEGs, FNEGd	A <i>reserved</i> field is not 0.
	FNEGq	Always. For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	FNEGs, FNEGd	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	FNEGq	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	FNEGq	When either of the following is true • rs2<1> ≠ 0 • rd<1> ≠ 0

7.38. FPACK

Refer to Section 7.51 in UA2011.

Exception	Target Instruction	Condition
<i>illegal_instruction</i>	FPACK16, FPACKFIX	$iw<18:14> \neq 0$
<i>fp_disabled</i>	All	$PSTATE.pef = 0$ or $FPRS.fef = 0$
<i>illegal_action</i>	All	$XAR.v = 1$

7.39. Fixed-point Partitioned Add

<SPARC64™ X>

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPADD16	0 0101 0000 ₂	Four 16-bit addition			<code>fpadd16 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPADD16S	0 0101 0001 ₂	Two 16-bit addition			<code>fpadd16s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPADD32	0 0101 0010 ₂	Two 32-bit addition			<code>fpadd32 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPADD32S	0 0101 0011 ₂	One 32-bit addition			<code>fpadd32s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>

<SPARC64™ X+>

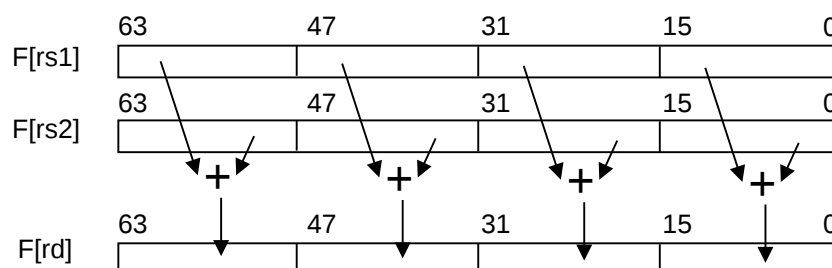
Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPADD16	0 0101 0000 ₂	Four 16-bit addition	✓	✓	<code>fpadd16 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPADD16S	0 0101 0001 ₂	Two 16-bit addition	✓	✓	<code>fpadd16s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPADD32	0 0101 0010 ₂	Two 32-bit addition	✓	✓	<code>fpadd32 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPADD32S	0 0101 0011 ₂	One 32-bit addition	✓	✓	<code>fpadd32s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>

Refer to Section 7.52 in UA2011.

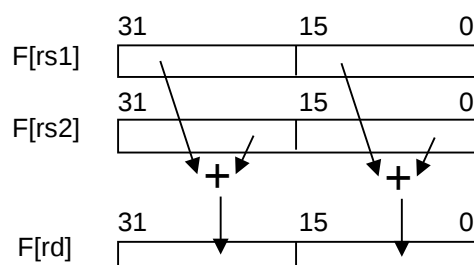
Note FPADD{16|32}{S} do not update any fields in FSR.

Note SIMD is not available for these instructions on SPARC64™ X, but it is available on SPARC64™ X+.

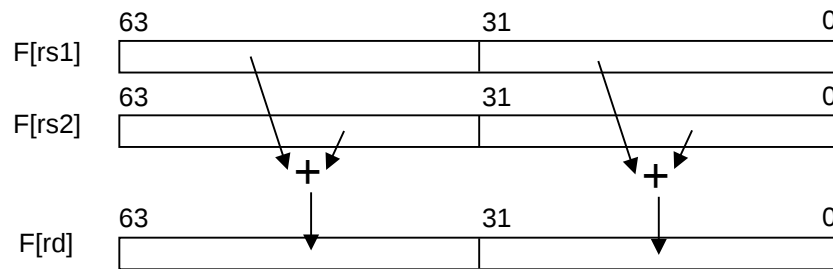
Behavior of FPADD16



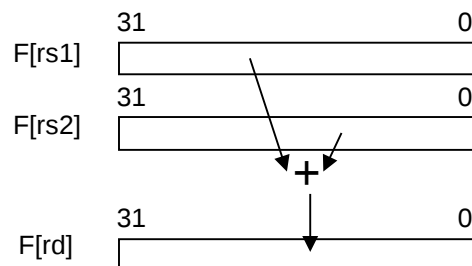
Behavior of FPADD16S



Behavior of FPADD32



Behavior of FPADD32S



< SPARC64™ X >

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

< SPARC64™ X+ >

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.40. Integer Multiply-Add

Instruction	var	size	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPMADDX	00 ₂	00 ₂	Lower 8 bytes of unsigned integer multiply-add	✓	✓	fpmaddx <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>
FPMADDXH I	01 ₂	00 ₂	Upper 8 bytes of unsigned integer multiply-add	✓	✓	fpmaddxhi <i>freg_{rs1}, freg_{rs2}, freg_{rs3}, freg_{rd}</i>

Refer to Section 7.56 in UA2011.

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<1> ≠ 0 • XAR.urd<1> ≠ 0 • XASR.simd = 1 and XAR.urs1<2> ≠ 0 • XASR.simd = 1 and XAR.urs2<2> ≠ 0 • XASR.simd = 1 and XAR.urs3<2> ≠ 0 • XASR.simd = 1 and XAR.urd<2> ≠ 0

7.41. FPMERGE

Refer to Section 7.57 in UA2011.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

7.42. Fixed-point Partitioned Subtract (64-bit)

<SPARC64™ X>

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPSUB16	0 0101 0100 ₂	Four 16-bit Subtract			<code>fpsub16 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPSUB16S	0 0101 0101 ₂	Two 16-bit Subtract			<code>fpsub16s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPSUB32	0 0101 0110 ₂	Two 32-bit Subtract			<code>fpsub32 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPSUB32S	0 0101 0111 ₂	One 32-bit Subtract			<code>fpsub32s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>

<SPARC64™ X+>

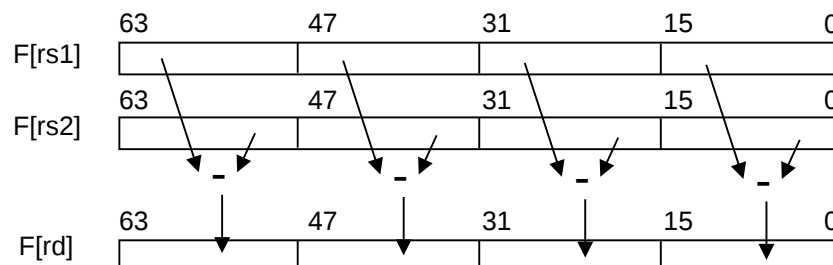
Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPSUB16	0 0101 0100 ₂	Four 16-bit Subtract	✓	✓	<code>fpsub16 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPSUB16S	0 0101 0101 ₂	Two 16-bit Subtract	✓	✓	<code>fpsub16s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPSUB32	0 0101 0110 ₂	Two 32-bit Subtract	✓	✓	<code>fpsub32 <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>
FPSUB32S	0 0101 0111 ₂	One 32-bit Subtract	✓	✓	<code>fpsub32s <i>freg_{rs1}</i>, <i>freg_{rs2}</i>, <i>freg_{rd}</i></code>

Refer to Section 7.58 in UA2011.

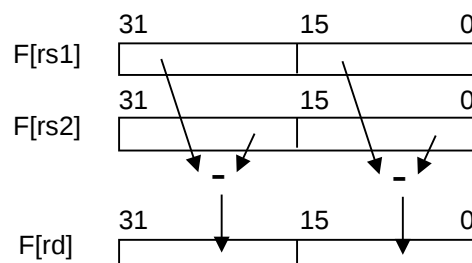
Note FPSUB{16|32}{S} do not update any fields in FSR.

Note SIMD is not available for these instructions on SPARC64™ X, but it is available on SPARC64™ X+.

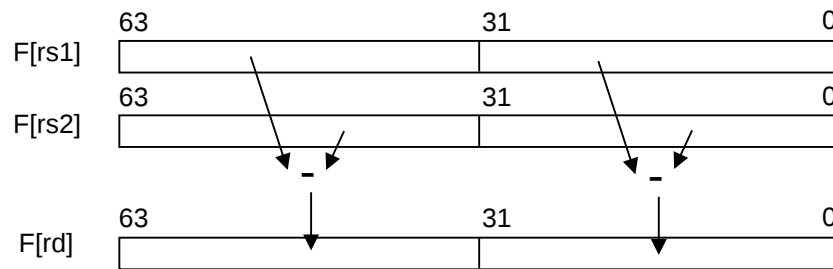
Behavior of FPSUB16



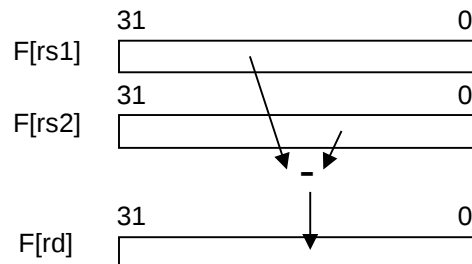
Behavior of FPSUB16S



Behavior of FPSUB32



Behavior of FPSUB32S



< SPARC64™ X >

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

< SPARC64™ X+ >

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.43. F Register Logical Operate

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FZERO	0 0110 0000 ₂	Zero fill	✓	✓	fzero	<i>freg_{rd}</i>
FZEROS	0 0110 0001 ₂	Zero fill, single precision	✓	✓	fzeros	<i>freg_{rd}</i>
FONE	0 0111 1110 ₂	One fill	✓	✓	fone	<i>freg_{rd}</i>
FONES	0 0111 1111 ₂	One fill, single precision	✓	✓	foners	<i>freg_{rd}</i>
FSRC1	0 0111 0100 ₂	Copy Fd[rs1]	✓	✓	fsrc1	<i>freg_{rs1}, freg_{rd}</i>
FSRC1s	0 0111 0101 ₂	Copy Fs[rs1]	✓	✓	fsrc1s	<i>freg_{rs1}, freg_{rd}</i>
FSRC2	0 0111 1000 ₂	Copy Fd[rs2]	✓	✓	fsrc2	<i>freg_{rs2}, freg_{rd}</i>
FSRC2s	0 0111 1001 ₂	Copy Fs[rs2]	✓	✓	fsrc2s	<i>freg_{rs2}, freg_{rd}</i>
FNOT1	0 0110 1010 ₂	Negate (1's complement) Fd[rs1]	✓	✓	fnot1	<i>freg_{rs1}, freg_{rd}</i>
FNOT1s	0 0110 1011 ₂	Negate (1's complement) Fs[rs1]	✓	✓	fnot1s	<i>freg_{rs1}, freg_{rd}</i>
FNOT2	0 0110 0110 ₂	Negate (1's complement) Fd[rs2]	✓	✓	fnot2	<i>freg_{rs2}, freg_{rd}</i>
FNOT2s	0 0110 0111 ₂	Negate (1's complement) Fs[rs2]	✓	✓	fnot2s	<i>freg_{rs2}, freg_{rd}</i>
FOR	0 0111 1100 ₂	Logical OR	✓	✓	for	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FORS	0 0111 1101 ₂	Logical OR, single precision	✓	✓	fors	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FNOR	0 0110 0010 ₂	Logical NOR	✓	✓	fnor	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FNORS	0 0110 0011 ₂	Logical NOR, single precision	✓	✓	fnors	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FAND	0 0111 0000 ₂	Logical AND	✓	✓	fand	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FANDs	0 0111 0001 ₂	Logical AND, single precision	✓	✓	fands	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FNAND	0 0110 1110 ₂	Logical NAND	✓	✓	fnand	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FNANDs	0 0110 1111 ₂	Logical NAND, single precision	✓	✓	fnands	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FXOR	0 0110 1100 ₂	Logical XOR	✓	✓	fxor	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FXORS	0 0110 1101 ₂	Logical XOR, single precision	✓	✓	fxors	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FXNOR	0 0111 0010 ₂	Logical XNOR	✓	✓	fxnor	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FXNORS	0 0111 0011 ₂	Logical XNOR, single precision	✓	✓	fxnors	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FORNOT1	0 0111 1010 ₂	(not Fd[rs1]) or Fd[rs2]	✓	✓	fornot1	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FORNOT1s	0 0111 1011 ₂	(not Fs[rs1]) or Fs[rs2]	✓	✓	fornot1s	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FORNOT2	0 0111 0110 ₂	Fd[rs1] or (not Fd[rs2])	✓	✓	fornot2	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FORNOT2s	0 0111 0111 ₂	Fs[rs1] or (not Fs[rs2])	✓	✓	fornot2s	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FANDNOT1	0 0110 1000 ₂	(not Fd[rs1]) and Fd[rs2]	✓	✓	fandnot1	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FANDNOT1s	0 0110 1001 ₂	(not Fs[rs1]) and Fs[rs2]	✓	✓	fandnot1s	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FANDNOT2	0 0110 0100 ₂	Fd[rs1] and (not Fd[rs2])	✓	✓	fandnot2	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>
FANDNOT2s	0 0110 0101 ₂	Fs[rs1] and (not Fs[rs2])	✓	✓	fandnot2s	<i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>

Refer to Sections 7.60, 7.61, and 7.62 in UA2011.

For the 64-bit versions of these instructions, the names of these instructions on SPARC64™ X / SPARC64™ X+ are different than the instruction names used in UA2011.

UA2011 name	SPARC64™ X / SPARC64™ X+ name
FZER0d	FZERO
FONEd	FONE
FSRC1d	FSRC1

FSRC2d	FSRC2
FNOT1d	FNOT1
FNOT2d	FNOT2
FORd	FOR
FNORd	FNOR
FANDd	FAND
FNANDd	FNAND
FXORd	FXOR
FXNORd	FXNOR
FORNOT1d	FORNOT1
FORNOT2d	FORNOT2
FANDNOT1d	FANDNOT1
FANDNOT2d	FANDNOT2

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_instruction</i>	FZERO, FZER0s, FONE, FONEs	iw<18:14> ≠ 0 or iw<4:0> ≠ 0
	FSRC1, FSRC1s, FNOT1, FNOT1s	iw<4:0> ≠ 0
	FSRC2, FSRC2s, FNOT2, FNOT2s	iw<18:14> ≠ 0
<i>illegal_action</i>	FZERO, FZER0s, FONE, FONEs	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XASR.simd = 1 and XAR.urd<2> ≠ 0
	FSRC1, FSRC1s, FNOT1, FNOT1s	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XASR.simd = 1 and XAR.urs1<2> ≠ 0 • XASR.simd = 1 and XAR.urd<2> ≠ 0
	FSRC2, FSRC2s, FNOT2, FNOT2s	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XASR.simd = 1 and XAR.urs2<2> ≠ 0 • XASR.simd = 1 and XAR.urd<2> ≠ 0
	FOR, FORs, FNOR, FNORs, FAND, FANDs, FNAND, FNANDs, FXOR, FXORs, FXNOR, FXNORs, FORNOT1, FORNOT1s, FORNOT2, FORNOT2s, FANDNOT1, FANDNOT1s, FANDNOT2, FANDNOT2s	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XASR.simd = 1 and XAR.urs1<2> ≠ 0 • XASR.simd = 1 and XAR.urs2<2> ≠ 0 • XASR.simd = 1 and XAR.urd<2> ≠ 0

7.44. Floating-Point Reciprocal Approximation

Instruction	opf	Operation	HPC-ACE		Assembly Language	
			Regs	SIMD	Syntax	
FRCPAD	1 0111 0100 ₂	Reciprocal Approximation Double	✓	✓	frcpad	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FRCPAS	1 0111 0101 ₂	Reciprocal Approximation Single	✓	✓	frcpas	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FRSQRTAD	1 0111 0110 ₂	Reciprocal Approximation of Square Root, Double	✓	✓	frsqrtd	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FRSQRTAS	1 0111 0111 ₂	Reciprocal Approximation of Square Root, Single	✓	✓	frsqrtd	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd			op3 = 11 0110 ₂	—			opf			rs2		
31 30 29	25 24			19 18	14 13			5 4			0		

Description FRCPA{s, d} calculates the reciprocal approximation of the value in the floating-point register specified by F[rs2] and stores the result in the floating-point register specified by F[rd]. Although the result is an approximation, the calculation ignores FSR.rd. The resulting rounding error is less than 1/256, when the result is normalized. In other words,

$$\left| \frac{frcpa(x) - 1/x}{1/x} \right| < \frac{1}{256}$$

Results and exceptions for FRCPA{s, d} are shown in Table 7-16. The upper row in each entry indicates the type(s) of exception if an exception is signalled, and the lower row in each entry indicates the result when an exception is not signalled. For more information on the causes of an *fp_exception_ieee_754* exception, refer to Appendix B in JPS1 Commonality.

Table 7-16 FRCPA{s|d}

op2	Exceptions and results	
	FSR.ns = 0	FSR.ns = 1
$+\infty$	— 0	— 0
$+N$ ($N \geq 2^{126}$ for single, $N \geq 2^{1022}$ for double)	UF ^{ix} approximation of $+1/N$ (denormal) ^x	UF, NX +0
$+N$ ($+N_{\min} \leq N < 2^{126}$ for single, $+N_{\min} \leq N < 2^{1022}$ for double)	— approximation of $+1/N$	— approximation of $+1/N$
$+D$	<i>unfinished_FPop</i> —	DZ $+\infty$
$+0$	DZ $+\infty$	DZ $+\infty$
-0	DZ $-\infty$	DZ $-\infty$
$-D$	<i>unfinished_FPop</i> —	DZ $-\infty$
$-N$ ($+N_{\min} \leq N < 2^{126}$ for single, $+N_{\min} \leq N < 2^{1022}$ for double)	— approximation of $-1/N$	— approximation of $-1/N$
$-N$ ($N \geq 2^{126}$ for single, $N \geq 2^{1022}$ for double)	UF ^{ix} approximation of $-1/N$ (denormal) ^x	UF, NX -0
$-\infty$	— -0	— -0
SNaN	NV QSNaN2	NV QSNaN2
QNaN	— op2	— op2

N	Positive normal number (except for zero, NaN, and infinity)
D	Positive denormal number
Nmin	Minimum value of a positive normal number
dNaN	Sign of QNaN is 0 and all bits of the exponent and significand are 1
QSNaN2	Refer to TABLE B-1 in JPS1 Commonality

FRSQRTA{s|d} calculates the reciprocal approximation of the square root of the value in the floating-point register specified by F[rs2] and stores the result in the floating-point register specified by F[rd]. Although the result is an approximation, the calculation ignores FSR.rd. The resulting rounding error is less than 1/256. In other words,

$$\left| \frac{frsqрта(x) - 1/\sqrt{x}}{1/\sqrt{x}} \right| < \frac{1}{256}$$

Results and exceptions for FRSQRTA{s|d} are shown in Table 7-17. The upper row in each entry indicates the type(s) of exception if an exception is signalled, and the lower row in each entry indicates the result when an exception is not signalled. For more information on the causes of an *fp_exception_ieee_754* exception, refer to Appendix B in JPS1 Commonality

^{ix} When FSR.tem.ufm = 0, NX is not detected.

^x When the result is denormal, the rounding error may be larger than 1/256.

Table 7-17 FRSQRTA{s|d}

op2	Exceptions and results	
	FSR.ns = 0	FSR.ns = 1
$+\infty$	— $+\infty$	— $+\infty$
$+N$	— $+1/(\sqrt{N})$	— $+1/(\sqrt{N})$
$+D$	<i>unfinished_FPop</i> —	— $+0$
$+0$	— $+0$	— $+0$
-0	— $+0$	— $+0$
$-D$	NV dNaN	NV dNaN
$-N$	NV dNaN	NV dNaN
$-\infty$	NV dNaN	NV dNaN
SNaN	NV QSNaN2	NV QSNaN2
QNaN	— op2	— op2

Exception		Target instruction	Condition
<i>illegal_instruction</i>		All	A <i>reserved</i> field is not 0. (iw<18:14> $\neq$ 0)
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.urs1 $\neq$ 0 • XAR.urs2<1> $\neq$ 0 • XAR.urs3 $\neq$ 0 • XAR.urd<1> $\neq$ 0 • XAR.simd = 1 and XAR.urs2<2> $\neq$ 0 • XAR.simd = 1 and XAR.urd<2> $\neq$ 0
<i>fp_exception_ieee_754</i>	NV, DZ	FRCPAs, FRCPAd, FRSQRTs, FRSQRTAd	Conforms to IEEE754.
	UF, NX	FRCPAs, FRCPAd	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		All	Refer to Chapter 8.

7.45. Move Selected Floating-Point Register on Floating-Point Register's Condition

Instruction	var	size	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
FSELMOVD	11 ₂	00 ₂	Select and Move Double	✓	✓	fselmovd	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rs3}</i> , <i>freg_{rd}</i>
FSELMOVS	11 ₂	11 ₂	Select and Move Single.	✓	✓	fselmovs	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rs3}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0111 ₂	rs1	rs3	var	size	rs2
31 30 29	25 24	19 18	14 13	9 8	7 6	5 4	0

Description FSELMOV{s|d} selects F[rs1] or F[rs2] according to the most-significant bit (MSB) of the floating-point register specified by F[rs3] and stores the value of the selected register in F[rd].

For FSELMOVD, if Fd[rs3]<63> is 1, Fd[rs1] is selected; if Fd[rs3]<63> is 0, Fd[rs2] is selected. For FSELMOVS, if Fs[rs3]<31> is 1, Fs[rs1] is selected; if Fs[rs3]<31> is 0, Fs[rs2] is selected.

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<1> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urs3<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.46. Floating-Point Square Root

Instruction	opf	Operation	HPC-ACE Regs.	SIMD	Assembly Language Syntax
FSQRTs	0 0010 1001 ₂	Square Root Single	rd is basic only.		fsqrts <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSQRTd	0 0010 1010 ₂	Square Root Double	rd is basic only.		fsqrt d <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSQRTq	0 0010 1011 ₂	Square Root Quad	rd is basic only.		fsqrt q <i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.64 in UA2011.

Note Rounding is performed as specified by FSR.rd or GSR.irnd.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FSQRTs, FSQRTd	A <i>reserved</i> field is not 0.
	FSQRTq	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<2:1> ≠ 0
<i>fp_exception_ieee_754</i> NV, NX	All	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	FSQRTq	When either of the following is true • rs2<1> ≠ 0 • rd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)	FSQRTs, FSQRTd	Refer to Chapter 8.

7.47. Floating-Point Trigonometric Functions

Instruction	op3	opf	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FTRIMADDd	11 0111 ₂	—	Trigonometric Multiply-Add Double	✓	✓	ftrimaddd <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>index</i> , <i>freg_{rd}</i>
FTRISMULD	11 0110 ₂	1 0111 1010 ₂	Calculate starting value for FTRIMADDd	✓	✓	ftrismuld <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FTRISSELD	11 0110 ₂	1 0111 1000 ₂	Select coefficient for final calculation in Taylor series approximation FTRIMADDd	✓	✓	ftrisseld <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0111 ₂	rs1	index	var = 10 ₂	size = 00 ₂	rs2								
10 ₂	rd	op3 = 11 0110 ₂	rs1	opf			rs2								
31	30	29	25	24	19	18	14	13	9	8	7	6	5	4	

Instruction	Operation
FTRIMADDd	$Fd[rd] \leftarrow Fd[rs1] \times \text{abs}(Fd[rs2]) + T[Fd[rs2] < 63][index]$
FTRISMULD	$Fd[rd] \leftarrow (Fd[rs2] < 0 > << 63) \wedge (Fd[rs1] \times Fd[rs1])$
FTRISSELD	$Fd[rd] \leftarrow (Fd[rs2] < 1 > << 63) \wedge (Fd[rs2] < 0 > ? 1.0 : Fd[rs1])$

Description These instructions accelerate the calculation of the Taylor series approximation of the sine function $\sin(x)$. FTRIMADDd operates on the result of FTRISMULD, and the intermediate result is multiplied by the result of FTRISSELD. All three instructions are defined as double-precision instructions only. FTRIMADDd calculates series terms for either $\sin(x)$ or $\cos(x)$, where the argument is adjusted to be in the range $-\pi/4 < x \leq \pi/4$. These series terms are used to perform the supporting operations shown in Figure 7-1. See the example at the end of this section for description of how to calculate $\sin(x)$ using these support operations.

$$\begin{aligned}
\sin x &\cong x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \frac{1}{9!}x^9 - \frac{1}{11!}x^{11} + \frac{1}{13!}x^{13} - \frac{1}{15!}x^{15} \\
&= x \left(1 - \frac{1}{3!}x^2 + \frac{1}{5!}x^4 - \frac{1}{7!}x^6 + \frac{1}{9!}x^8 - \frac{1}{11!}x^{10} + \frac{1}{13!}x^{12} - \frac{1}{15!}x^{14} \right) \\
&= x \cdot \left(\left(\left(\left(\left(\left(\left(\left(\left(0 \cdot x^2 - \frac{1}{15!} \right) x^2 + \frac{1}{13!} \right) x^2 - \frac{1}{11!} \right) x^2 + \frac{1}{9!} \right) x^2 - \frac{1}{7!} \right) x^2 + \frac{1}{5!} \right) x^2 - \frac{1}{3!} \right) x^2 + 1 \right) \right) \right) \right) \right) \right) \right) \right)
\end{aligned}$$

$$\begin{aligned}
\cos x &\cong 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \frac{1}{8!}x^8 - \frac{1}{10!}x^{10} + \frac{1}{12!}x^{12} - \frac{1}{14!}x^{14} \\
&= 1 \cdot \left(\left(\left(\left(\left(\left(\left(\left(\left(0 \cdot x^2 - \frac{1}{14!} \right) x^2 + \frac{1}{12!} \right) x^2 - \frac{1}{10!} \right) x^2 + \frac{1}{8!} \right) x^2 - \frac{1}{6!} \right) x^2 + \frac{1}{4!} \right) x^2 - \frac{1}{2!} \right) x^2 + 1 \right) \right) \right) \right) \right) \right) \right) \right)
\end{aligned}$$

Figure 7-1 Trigonometric functions assistance operation

FTRIMADDd multiplies Fd[rs1] and the absolute value of Fd[rs2] and adds the product to a double-precision number obtained from a table. This double-precision number is specified by the index field. The result is stored in the double-precision register specified by Fd[rd]. FTRIMADDd is used to calculate series terms in the Taylor series of sin(x) or cos(x), where $-\pi/4 < x \leq \pi/4$.

FTRISMULD squares the value in the double-precision register specified by Fd[rs1]. The sign of the squared value is selected according to bit 0 of the double-precision register specified by Fd[rs2]. The result is written to the double-precision register specified by Fd[rd].

FTRISMULD is used to calculate the starting value of FTRIMADDd.

FTRISSELD checks bit 0 of the double-precision register specified by Fd[rs2]. Based on this bit, either the double-precision register specified by Fd[rs1] or the value 1.0 is selected. Bit 1 of Fd[rs2] indicates the sign; the exclusive OR of this bit and the selected value is written to the double-precision register specified by Fd[rd]. FTRISSELD is used to select the coefficient for calculating the last step in the Taylor series approximation.

To calculate the series terms of sin(x) and cos(x), the initial source operands of FTRIMADDd are zero for Fd[rs1] and x^2 for Fd[rs2], where $-\pi/4 < x \leq \pi/4$. FTRIMADDd is executed 8 times; this calculates the sum of 8 series terms, which gives the resulting number sufficient precision for a double-precision floating-point number. As shown in Figure 7-1, the coefficients of the series terms are different for sin(x) and cos(x). FTRIMADDd uses the sign of Fd[rs2] to determine which set of coefficients to use.

- When Fd[rs2]<63> = 0, the coefficient table for sin(x) is used.
- When Fd[rs2]<63> = 1, the coefficient table of cos(x) is used.

The expected usage for FTRIMADDd is shown in the example below. Coefficients are chosen to minimize the loss of precision; these differ slightly from the exact mathematical values. Table 7-18 and Table 7-19 show the coefficient tables for FTRIMADDd.

Table 7-18 Coefficient Table for $\sin(x)$ ($Fd[rs2] <63> = 0$)

Index	Coefficient used for the operation		Exact value of the coefficient
	Hexadecimal representation	Decimal representation	
0	3ff0 0000 0000 0000 ₁₆	1.0	= 1/1!
1	bfc5 5555 5555 5543 ₁₆	−0.1666666666666661	> −1/3!
2	3f81 1111 1110 f30c ₁₆	0.8333333333320002e−02	< 1/5!
3	bf2a 01a0 19b9 2fc6 ₁₆	−0.1984126982840213e−03	> −1/7!
4	3ec7 1de3 51f3 d22b ₁₆	0.2755731329901505e−05	< 1/9!
5	be5a e5e2 b60f 7b91 ₁₆	−0.2505070584637887e−07	> −1/11!
6	3de5 d840 8868 552f ₁₆	0.1589413637195215e−09	< 1/13!
7	0000 0000 0000 0000 ₁₆	0	> −1/15!

Table 7-19 Coefficient Table for $\cos(x)$ ($Fd[rs2] <63> = 1$)

Index	Coefficient used for the operation		Exact value of the coefficient
	Hexadecimal representation	Decimal representation	
0	3ff0 0000 0000 0000 ₁₆	1.0	= 1/0!
1	bfe0 0000 0000 0000 ₁₆	−0.5000000000000000	= −1/2!
2	3fa5 5555 5555 5536 ₁₆	0.4166666666666645e−01	< 1/4!
3	bf56 c16c 16c1 3a0b ₁₆	−0.1388888888886111e−02	> −1/6!
4	3efa 01a0 19b1 e8d8 ₁₆	0.2480158728388683e−04	< 1/8!
5	be92 7e4f 7282 f468 ₁₆	−0.2755731309913950e−06	> −1/10!
6	3e21 ee96 d264 1b13 ₁₆	0.2087558253975872e−08	< 1/12!
7	bda8 f763 80fb b401 ₁₆	−0.1135338700720054e−10	> −1/14!

The initial value in $Fd[rs2]$ for FTRIMADDd is calculated using FTRISMULD, which is executed with $Fd[rs1]$ set to x , where $-\pi/4 < x \leq \pi/4$ and $Fd[rs2]$ set to Q , as defined in Figure 7-2. FTRISMULD returns x^2 as the result, where the sign bit specifies which set of coefficients to use to calculate the series terms. Q is an integer, not a floating-point number. Bits $Fd[rs2]<63:1>$ are not used. An exception is not detected if $Fd[rs2]$ is NaN.

The final step in the calculation of the Taylor series is the multiplication of the FTRIMADDd result and the coefficient selected by FTRISSELD. This coefficient is selected by executing FTRISSELD with $Fd[rs1]$ set to x , where $-\pi/4 < x \leq \pi/4$ and $Fd[rs2]$ set to Q , as defined in Figure 7-2. Either x or 1.0 is selected, and the appropriate sign is affixed to the result. Q is an integer, not a floating-point number. Bits $Fd[rs2]<63:2>$ are not used. An exception is not detected if $Fd[rs2]$ is NaN.

$$q : (2q-1) \cdot \frac{\pi}{4} < x \leq (2q+1) \cdot \frac{\pi}{4}$$

$$Q : q \bmod 4$$

$$R : x - q \cdot \frac{\pi}{2} \left(-\frac{\pi}{4} < R \leq \frac{\pi}{4} \right)$$

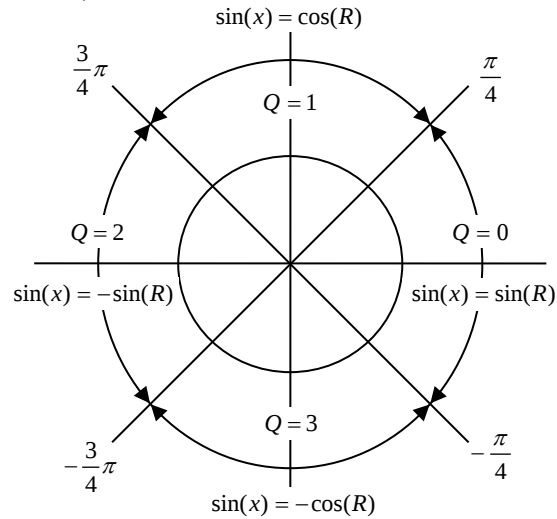


Figure 7-2 Relationships for calculating sin(x)

Example: calculating sin(x)

```

/*
 * Input value: x
 * q: where (2q-1)*pi/4 < x <= (2q+1)*pi/4
 * Q: q%4
 * R: x - q * pi/2
 */

ftrismuld R, Q, M
ftrisseld R, Q, N

/*
 * M ← R2[63]=table_type, R2[62:0]=R2
 * Because R2 is always positive, the sign bit (bit<63>) is always 0.
 * This sign bit indicates the table_type of ftrimaddd. )
 * N ← coefficient used in final step; the value is (1.0 or R)* sign
 * S ← 0
 */

ftrimaddd S, M, 7, S
ftrimaddd S, M, 6, S
ftrimaddd S, M, 5, S
ftrimaddd S, M, 4, S
ftrimaddd S, M, 3, S
ftrimaddd S, M, 2, S
ftrimaddd S, M, 1, S
ftrimaddd S, M, 0, S
fmuld S, N, S

/*
 * S ← result
 */

```

Exception		Target instruction	Condition
<i>illegal_instruction</i>		FTRIMADDd	index > 7
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>fp_exception_ieee754</i>	NV	FTRIMADDd, FTRISMULd	Conforms to IEEE754. FTRISMULd : only rs1
	NX	FTRIMADDd, FTRISMULd	Conforms to IEEE754.
	OF	FTRIMADDd, FTRISMULd	Conforms to IEEE754.
	UF	FTRIMADDd, FTRISMULd	Conforms to IEEE754.
<i>fp_exception_other</i> (FSR.ftt = <i>unfinished_FPop</i>)		FTRIMADDd, FTRISMULd	

7.48. Illegal Instruction Trap

Refer to Section 7.69 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	Always

7.49. Integer Logical Operation

Refer to Sections 7.7, 7.98, and 7.144 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0. (i = 0 and iw<12:5> $\neq$ 0)
<i>illegal_action</i>	All	XAR.v = 1

7.51. Jump and Link

Refer to Section 7.71 in UA2011.

Note SPARC64™ X / SPARC64™ X+ clear the most significant 32 bits of the PC value stored in R[rd] when PSTATE.am = 1. The updated value in R[rd] is visible to the delay slot instruction immediately.

Note If either of the two lowest bits of the jump address is nonzero, a *mem_address_not_aligned* exception occurs.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	XAR.v = 1
<i>mem_address_not_aligned</i>	When either of the two lowest bits of the target address is not 0

7.52. Load Integer

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
LDSB	00 1001 ₂	Load Signed Byte	✓		ldsb [address], reg _{rd}
LDSH	00 1010 ₂	Load Signed Halfword	✓		ldsh [address], reg _{rd}
LDSW	00 1000 ₂	Load Signed Word	✓		ldsw [address], reg _{rd}
LDUB	00 0001 ₂	Load Unsigned Byte	✓		ldub [address], reg _{rd}
LDUH	00 0010 ₂	Load Unsigned Halfword	✓		lduh [address], reg _{rd}
LDUW	00 0000 ₂	Load Unsigned Word	✓		lduw [address], reg _{rd}
					ld [address], reg _{rd}
LDX	00 1011 ₂	Load Extended Word	✓		ldx [address], reg _{rd}

Refer to Section 7.72 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>mem_address_not_aligned</i>	LDUH, LDSH, LDUW, LDSW, LDX	Refer to UA2011.
<i>VA_watchpoint</i>	All	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

Related LDTW (page 135)

7.53. Load Integer from Alternate Space

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
LDSBA ^{PASI}	01 1001 ₂	Load Signed Byte from Alternate Space	✓		ldsba [address] <i>imm_asi</i> , reg _{rd}	
					ldsba [address] %asi, reg _{rd}	
LDSHA ^{PASI}	01 1010 ₂	Load Signed Halfword from Alternate Space	✓		ldsha [address] <i>imm_asi</i> , reg _{rd}	
					ldsha [address] %asi, reg _{rd}	
LDSWA ^{PASI}	01 1000 ₂	Load Signed Word from Alternate Space	✓		ldswa [address] <i>imm_asi</i> , reg _{rd}	
					ldswa [address] %asi, reg _{rd}	
LDUBA ^{PASI}	01 0001 ₂	Load Unsigned Byte from Alternate Space	✓		lduba [address] <i>imm_asi</i> , reg _{rd}	
					lduba [address] %asi, reg _{rd}	
LDUHA ^{PASI}	01 0010 ₂	Load Unsigned Halfword from Alternate Space	✓		lduha [address] <i>imm_asi</i> , reg _{rd}	
					lduha [address] %asi, reg _{rd}	
LDUWA ^{PASI}	01 0000 ₂	Load Unsigned Word from Alternate Space	✓		lduwa [address] <i>imm_asi</i> , reg _{rd}	
					lduwa [address] %asi, reg _{rd}	
					lda [address] <i>imm_asi</i> , reg _{rd}	
					lda [address] %asi, reg _{rd}	
LDXA ^{PASI}	01 1011 ₂	Load Extended Word from Alternate Space	✓		ldxa [address] <i>imm_asi</i> , reg _{rd}	
					ldxa [address] %asi, reg _{rd}	

Refer to Section 7.73 in UA2011.

Exception	Target instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>mem_address_not_aligned</i>	LDUHA, LDSHA, LDUWA, LDSWA, LDXA	Refer to the UA2011.
<i>privileged_action</i>	All	PSTATE.priv = 0 and ASI 00 ₁₆ – 7F ₁₆ is specified
<i>VA_watchpoint</i>	All	Refer to 12.5.1.62
<i>DAE_invalid_asi</i>	All	Refer to UA2011 and 12.5.1.5
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

Related

LDTWA (136Page)

7.54. Block Load

Instruction	ASI	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
LDBLOCKF	F0 ₁₆	64-byte block load from primary address space	✓		ldda [regaddr] ASI_BLK_P, freg _{rd} ldda [reg_plus_imm] %asi, freg _{rd}
LDBLOCKF	F1 ₁₆	64-byte block load from secondary address space	✓		ldda [regaddr] ASI_BLK_S, freg _{rd} ldda [reg_plus_imm] %asi, freg _{rd}
LDBLOCKF	F8 ₁₆	64-byte block load from primary address space, little-endian	✓		ldda [regaddr] ASI_BLK_PL, freg _{rd} ldda [reg_plus_imm] %asi, freg _{rd}
LDBLOCKF	F9 ₁₆	64-byte block load from secondary address space, little-endian	✓		ldda [regaddr] ASI_BLK_SL, freg _{rd} ldda [reg_plus_imm] %asi, freg _{rd}

Refer to Section 7.74 in UA2011.

The LDBLOCKF can only be used to access cacheable addresses, unlike a normal load. LDBLOCKF ASIs do not allow LDBLOCKF to access the non-cacheable space.

The effective address is “R[rs1] + R[rs2]” if *i* = 0, or “R[rs1] + **sign_ext**(simm13)” if *i* = 1.

When an exception is generated for a block load, register values may have been updated by the block load.

LDBLOCKF on SPARC64™ X / SPARC64™ X+ follow TSO. That is, the ordering between the preceding and following load/store/atomic instructions and the 8-byte loads comprising the block loads conforms to TSO.

LDBLOCKF on SPARC64™ X / SPARC64™ X+ preserves the order of register accesses in the same manner as any other load instruction. The cache behavior of LDBLOCKF is the same as for a normal load. A block load reads data from the L1D cache; if the data is not in the L1D cache, the L1D cache is updated with data from memory before being read.

A *VA_watchpoint* exception is detected only for the first eight bytes accessed by an LDBLOCKF instruction.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	Register number specified for rd is not a multiple of 8.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd<1> $\neq$ 0
<i>mem_address_not_aligned</i>	All	Address is not aligned on a 64-byte boundary
<i>VA_watchpoint</i>	All	On access to lowest 8 bytes only Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	ASI F0 ₁₆ , F1 ₁₆ , F8 ₁₆ , and F9 ₁₆	PSTATE.priv = 0 and TTE.p = 1 Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	Access to non-cacheable space Refer to 12.5.1.6
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.55. Load Floating-Point

Instruction	op3	rd ^{xi}	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
LDF	10 0000 ₂	0 – 31	Load to Single Floating-Point Register (XAR.v = 0)			ld	[address], freg _{rd}
LDF	10 0000 ₂	0 – 126, 256 – 382	Load to Double Floating-Point Register (XAR.v = 1)	✓	✓	ld	[address], freg _{rd}
LDDF	10 0011 ₂	0 – 126, 256 – 382	Load to Double Floating-Point Register	✓	✓	ldd	[address], freg _{rd}
LDQF	10 0010 ₂	0 – 126, 256 – 382	Load to Quad Floating-Point Register	✓		ldq	[address], freg _{rd}

Non-SIMD execution

Refer to Section 7.75 in UA2011.

LDF copies a word from memory into the 32-bit floating-point destination register F[rd]. If XAR.v = 0, LDF copies the word into a single-precision floating point register. If XAR.v = 1, LDF copies the word into the upper 32-bits of a double-precision floating point register.

SIMD execution

On SPARC64™ X / SPARC64™ X+, LDF and LDDF can be executed as SIMD instructions. A SIMD LDF and SIMD LDDF simultaneously execute basic and extended loads from the effective address for single-precision and double-precision data, respectively. Refer to Section 5.5.14 (page 35) for details on how to specify the registers.

A SIMD LDF loads 2 single-precision data aligned on a 4-byte boundary. Data from the lower 4-bytes of the address is loaded into the upper 4-bytes of Fd[rd], and data from the upper 4-byte of the address is loaded into the upper 4-bytes of Fd[rd+256]. Misaligned accesses cause a *mem_address_not_aligned* exception.

A SIMD LDDF instruction loads 2 double-precision data aligned on an 8-byte boundary. Data from the lower 8-bytes of the address is loaded into Fd[rd], and data from the upper 8-bytes of the address is loaded into Fd[rd+256]. Misaligned accesses cause a *mem_address_not_aligned* exception.

Note A double-precision SIMD load that accesses data aligned on a 4-byte boundary but not an 8-byte boundary does not cause an *LDDF_mem_address_not_aligned* exception.

SIMD LDF and SIMD LDDF can only be used to access cacheable address spaces. An attempt to access a noncacheable address space using a SIMD LDF or SIMD LDDF causes a *DAE_nc_page* exception.

Like non-SIMD load instructions, memory access semantics for SIMD load instructions adhere to TSO. A SIMD load simultaneously executes basic and extended loads; however, the ordering between the basic and extended loads conforms to TSO.

^{xi} Encoding defined in 5.3.1 "Floating-Point Register Number Encoding" (page 26).

For a SIMD load instruction, endian conversion is done separately for the basic and extended loads. When the basic and extended data are located on different pages with different endianness, conversion is only done for one of the loads. A watchpoint can be detected in both the basic and extended loads.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	LDF, LDDF	A <i>reserved</i> field is not 0.
	LDQF	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	LDF, LDDF	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDQF	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	LDQF	rd<1> ≠ 0
<i>LDDF_mem_address_not_aligned</i>	LDDF	XAR.v = 0 or XAR.simd = 0, and address is 4-byte aligned but not 8-byte aligned
<i>mem_address_not_aligned</i>	LDF, LDQF	Address not 4-byte aligned
	LDDF	When either of the following is true • XAR.v = 0 or XAR.simd = 0, and address not 4-byte aligned • XAR.v = 1, XAR.simd = 1, and address not 8-byte aligned
<i>VA_watchpoint</i>	All	Refer to the description and 12.5.1.62
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	XAR.v = 1, XAR.simd = 1, and access to non-cacheable space
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.56. Load Floating-Point from Alternate Space

Instruction	op3	rd ^{xii}	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
L DFA ^{PASI}	11 0000 ₂	0 – 31	Load to Single Floating-Point register from Alternate Space (XAR.v = 0)			lda [address] imm_asi , freg _{rd}	lda [address] %asi, freg _{rd}
L DFA ^{PASI}	11 0000 ₂	0 – 126, 256 – 382	Load to Double Floating-Point register from Alternate Space (XAR.v = 1)	✓	✓	lda [address] imm_asi , freg _{rd}	lda [address] %asi, freg _{rd}
L DDFA ^{PASI}	11 0011 ₂	0 – 126, 256 – 382	Load to Double Floating-Point Register from Alternate Space	✓	✓	ldda [address] imm_asi , freg _{rd}	ldda [address] %asi, freg _{rd}
L DQFA ^{PASI}	11 0010 ₂	0 – 126, 256 – 382	Load to Quad Floating-Point Register from Alternate Space	✓		ldqa [address] imm_asi , freg _{rd}	ldqa [address] %asi, freg _{rd}

Non-SIMD execution

Refer to Section 7.76 in UA2011.

L DFA copies a word from Alternate Space into the 32-bit floating-point destination register F[rd]. If XAR.v = 0, LDF copies the word into a single precision floating-point register. If XAR.v = 1, LDF copies the word into the upper 32 bits of a double-precision floating point register.

SIMD execution

On SPARC64™ X / SPARC64™ X+, L DFA and L DDFA can be executed as SIMD instructions. A SIMD L DFA and SIMD L DDFA simultaneously execute basic and extended loads from the effective address for single-precision and double-precision data, respectively. Refer to Section 5.5.14 (page 35) for details on how to specify the registers.

A SIMD L DFA loads 2 single-precision data aligned on a 4-byte boundary. Data from the lower 4 bytes of the address is loaded into the upper 4-bytes of Fd[rd], and data from the upper 4 bytes of the address is loaded into the upper 4 bytes of Fd[rd+256]. Misaligned accesses cause a *mem_address_not_aligned* exception.

A SIMD L DDFA loads 2 double-precision data aligned on an 8-byte boundary. Data from the lower 8 bytes of the address is loaded into Fd[rd], and data from the upper 8 bytes of the address is loaded into Fd[rd+256]. Misaligned accesses cause a *mem_address_not_aligned* exception.

Note A double-precision SIMD load that accesses data aligned on a 4-byte boundary but not an 8-byte boundary does not cause an *LDDF_mem_address_not_aligned* exception.

^{xii} Encoding defined in 5.3.1 "Floating-Point Register Number Encoding" (page 26).

SIMD LDFA and SIMD LDDFA can only be used to access cacheable address spaces. An attempt to access a non-cacheable address space using a SIMD LDFA or SIMD LDDFA causes a *DAE_nc_page* exception.

Like non-SIMD load instructions, memory access semantics for SIMD load instructions adhere to TSO. SIMD LDFA and SIMD LDDFA simultaneously execute basic and extended loads; however, the ordering between the basic and extended loads conforms to TSO.

For SIMD LDFA and SIMD LDDFA, endian conversion is done separately for the basic and extended loads. When the basic and extended data are located on different pages with different endianness, conversion is only done for one of the loads. A watchpoint can be detected in both the basic and extended loads.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	LDQFA	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	LDFA, LDDFA	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDQFA	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	LDQFA	rd<1> ≠ 0
<i>LDDF_mem_address_not_aligned</i>	LDDFA	XAR.v = 0 or XAR.simd = 0, and address 4-byte aligned but not 8-byte aligned
<i>mem_address_not_aligned</i>	LDFA, LDQFA	Address not 4-byte aligned
	LDDFA	When either of the following is true • XAR.v = 0 or XAR.simd = 0, and address not 4-byte aligned • XAR.v = 1, XAR.simd = 1, and address not 8-byte aligned
<i>privileged_action</i>	All	Refer to 12.5.1.49
<i>VA_watchpoint</i>	All	Refer to the description and 12.5.1.62
<i>DAE_invalid_asl</i>	All	Refer to the UA2011 and 12.5.1.5
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	XAR.v = 1, XAR.simd = 1, and access to non-cacheable space
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7
<i>DAE_side_effect_page</i>	All	Refer to 12.5.1.9

7.57. Short Floating-Point Load

Refer to Section 7.78 in UA2011.

LDSHORTF is equivalent to LDDFA using ASIs D0₁₆ – D3₁₆ and D8₁₆ – DB₁₆. No other ASIs can be used with LDSHORTF.

An ASI is specified by the `imm_asi` instruction field when `i = 0`, or the contents of the ASI register when `i = 1`. If `i = 0`, the effective address for these instructions is “R[rs1] + R[rs2]” and if `i = 1`, the effective address is “R[rs1] + **sign_ext**(simm13)”.

Programming Note LDSHORTF is typically used with the FALIGNDATA instruction.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1
<i>mem_address_not_aligned</i>	Refer to UA2011
<i>VA_watchpoint</i>	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	Refer to 12.5.1.7

7.58. Load-Store Unsigned Byte

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
LDSTUB	00 1101 ₂	Load-Store Unsigned Byte	✓		ldstub [<i>address</i>], <i>reg_{rd}</i>

Refer to Section 7.79 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>VA_watchpoint</i>	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	Refer to 12.5.1.8
<i>DAE_nc_page</i>	Refer to 12.5.1.6
<i>DAE_nfo_page</i>	Refer to 12.5.1.7

7.59. Load-Store Unsigned Byte to Alternate Space

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
LDSTUBA ^{P_{ASI}}	01 1101 ₂	Load-Store Unsigned Byte into Alternate Space	✓		ldstuba [address] <i>imm_asi</i> , reg _{rd} ldsba [address] %asi, reg _{rd}

Refer to Section 7.80 in UA2011.

The effective address is “R[rs1] + R[rs2]” if *i* = 0, or “R[rs1] + **sign_ext**(simm13)” if *i* = 1.

The coherence and atomicity of memory operations between processors and I/O DMA memory accesses are not described in this specification. Refer to the system specification.

ASIs valid for LDSTUBA.	
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

Exception	Condition
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>privileged_action</i>	PSTATE.priv = 0 and ASI 00 ₁₆ - 7F ₁₆ is satisfied
<i>VA_watchpoint</i>	Refer to 12.5.1.62
<i>DAE_invalid_asi</i>	Refer to UA2011 and 12.5.1.5
<i>DAE_privilege_violation</i>	Refer to 12.5.1.8
<i>DAE_nc_page</i>	Refer to 12.5.1.6
<i>DAE_nfo_page</i>	Refer to 12.5.1.7

7.60. Load Integer Twin Word

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
LDTW ^D	00 0011 ₂	Load integer twin word	✓		ldtw [address], reg _{rd}

Refer to Section 7.81 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	When either of the following is true • $i = 0$ and $iw<12:5> \neq 0$ • LDTW refers to an odd-numbered destination register (rd).
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq 0$ • XAR.urs2 $\neq 0$ • XAR.urs3<2> $\neq 0$ • XAR.urd $\neq 0$
<i>mem_address_not_aligned</i>	Refer to UA2011
<i>VA_watchpoint</i>	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	Refer to 12.5.1.7

Related LDX (page 122)
 STTW (page 179)

7.61. Load Integer Twin Word from Alternate Space

Instruction	op3	Operation	HPC-ACE	Assembly Language Syntax	
			Regs SIMD		
LDTWA ^{D,P_{ASI}}	01 0011 ₂	Load Twin word from Alternate Space	✓	ldtwa [address] imm_asi, reg _{rd}	
				ldtwa [address] %asi, reg _{rd}	

11 ₂	rd	op3	rs1	i = 0	imm_asi	rs2
11 ₂	rd	op3	rs1	i = 1	simm13	
31 30 29	25 24	19 18	14 13 12	5 4	0	

Refer to Section 7.82 in UA2011.

Note For instructions that specify ASI_TWIX* for LDTWA, refer to 7.62.

ASIs valid for LDTWA	
ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE
ASI_PRIMARY_NO_FAULT	ASI_PRIMARY_NO_FAULT_LITTLE
ASI_SECONDARY_NO_FAULT	ASI_SECONDARY_NO_FAULT_LITTLE

Exception	Condition
<i>illegal_instruction</i>	rd is an odd-numbered register.
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd $\neq$ 0
<i>mem_address_not_aligned</i>	Refer to UA2011
<i>privileged_action</i>	PSTATE.priv = 0 and ASI of 00 ₁₆ - 7F ₁₆ .is specified
<i>VA_watchpoint</i>	Refer to 12.5.1.62
<i>DAE_invalid_asl</i>	Refer to 12.5.1.5
<i>DAE_privilege_violation</i>	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	Refer to 12.5.1.7

Related

LDXA (page 123)
STTWA (page 180)

7.62. Load Integer Twin Extended Word from Alternate Space

Instruction	ASI	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
LDTXA ^N	E2 ₁₆	Load Integer Twin Extended Word from Alternate Space	✓		ldtxa	[<i>regaddr</i>]#ASI_TWIX_P, <i>reg_{rd}</i>
	E3 ₁₆	Load Integer Twin Extended Word from Alternate Space	✓		ldtxa	[<i>regaddr</i>]#ASI_TWIX_S, <i>reg_{rd}</i>
	EA ₁₆	Load Integer Twin Extended Word from Alternate Space	✓		ldtxa	[<i>regaddr</i>]#ASI_TWIX_PL, <i>reg_{rd}</i>
	EB ₁₆	Load Integer Twin Extended Word from Alternate Space	✓		ldtxa	[<i>regaddr</i>]#ASI_TWIX_SL, <i>reg_{rd}</i>

11 ₂	rd	op3	rs1	i = 0	imm_asi	rs2
11 ₂	rd	op3	rs1	i = 1	simm13	
31 30 29		25 24	19 18	14 13 12	5 4	0

Refer to Section 7.83 in UA2011.

If *i* = 0, the LDTXA instruction contains the address space identifier (ASI) to be used for the load in its *imm_asi* field and the effective address for the instruction is “R[rs1] + R[rs2]”. If *i* = 1, the ASI to be used is contained in the ASI register and the effective address for the instruction is “R[rs1] + **sign_ext**(simm13)”.

A LDTXA instruction that performs a little-endian access behaves as if it comprises two 32-bit loads, each of which is byte-swapped independently before being written into its respective destination register.

A successful LDTXA instruction operates atomically.

Programming Note LDTXA can be used to read one entry of TSB TTE atomically.

ASIs E2₁₆, E3₁₆, EA₁₆ and EB₁₆ are used with LDTXA. An attempt to use other ASIs with LDTXA has the same result as LDTWA with those ASIs. ASIs E2₁₆, E3₁₆, EA₁₆ and EB₁₆ perform an access using a VA.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	rd is an odd-numbered register
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true. • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd $\neq$ 0
<i>mem_address_not_aligned</i>	All	Address not 16-byte aligned
<i>VA_watchpoint</i>	ASI E2 ₁₆ , E3 ₁₆ , EA ₁₆ , and EB ₁₆	Only detected for first 8 bytes Refer to 12.5.1.62.
<i>DAE_privilege_violation</i>	ASI E2 ₁₆ , E3 ₁₆ , E3 ₁₆ , and EB ₁₆	When PSTATE.priv = 0 and access to page with TTE.p = 1
<i>DAE_nc_page</i>	All	Refer to 12.5.1.6
<i>DAE_nfo</i>	All	Refer to 12.5.1.7

7.63. Load Floating-Point State Register

Instruction	op3	rd	Operation	HPC-ACE		Assembly Language	
				Regs	SIMD	Syntax	
LDFSR ^D	10 0001 ₂	0	Load Floating-Point State Register (Lower)	✓		ld	[address], %fsr
LDXFSR	10 0001 ₂	1	Load Floating-Point State Register	✓		ldx	[address], %fsr
—	10 0001 ₂	2 - 31	reserved				

Refer to Section 7.77 and Section 7.84 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	LDFSR, LDXFSR	$i = 0$ and $iw<12:5> \neq 0$.
	—	$rd = 2 - 31$
<i>fp_disabled</i>	All	$PSTATE.pef = 0$ or $FPRS.fef = 0$
<i>illegal_action</i>	All	$XAR.v = 1$ and any of the following are true $XAR.simd = 1$ $XAR.urs1 \neq 0$ $XAR.urs2 \neq 0$ $XAR.urs3<2> \neq 0$ $XAR.urd \neq 0$
<i>mem_address_not_aligned</i>	LDFSR	Address not 4-byte aligned.
	LDXFSR	Address not 8-byte aligned.
<i>VA_watchpoint</i>	All	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.64. Memory Barrier

Refer to Section 7.87 in UA2011.

Note `mmask<3>` has no effect on SPARC64™ X / SPARC64™ X+ because all stores are performed in program order.

Note `mmask<1>` has no effect on SPARC64™ X / SPARC64™ X+ because all stores are performed in program order and the ordering between a load and a store is guaranteed.

Note `mmask<0>` has no effect on SPARC64™ X / SPARC64™ X+ because all loads are performed in program order.

Note `#StoreStore` is equivalent to the deprecated `STBAR` instruction on SPARC64™ X / SPARC64™ X+.

Note `#MemIssue` is equivalent to `#Sync` on SPARC64™ X / SPARC64™ X+

Note `#Lookaside` is equivalent to `#Sync` on SPARC64™ X / SPARC64™ X+.

Exception	Condition
<i>illegal_instruction</i>	<code>iw<12:7> ≠ 0</code>
<i>illegal_action</i>	<code>XAR.v = 1</code>

7.65. Move Integer Register on Condition (MOV_{cc})

Refer to Section 7.90 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	$i = 0$ and $iw<10:5> \neq 0$
	—	Either of the following is true $cc2::cc1::cc0 = 101_2$ $cc2::cc1::cc0 = 111_2$
<i>fp_disabled</i>	MOV _F *	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1

7.66. Move Integer Register on Register Condition (MOVr)

Refer to Section 7.91 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0.
	—	When either of the following is true • $rcond = 000_2$ • $rcond = 100_2$
<i>illegal_action</i>	All	$XAR.v = 1$

7.67. Multiply Step

Instruction	op3	Operation	HPC-ACE	Assembly Language Syntax	
			Regs SIMD		
MULSCC ^D	10 0100 ₂	Multiply Step and modify CC		mulSCC	reg _{rs1} , reg_or_imm, reg _{rd}

10 ₂	rd	op3 = 10 0100 ₂	rs1	i = 0	—	rs2
31 30 29	25 24	19 18	14 13 12	5 4	0	
10 ₂	rd	op3 = 10 0100 ₂	rs1	i = 1	simm13	

Description MULSCC is a deprecated instruction that assists in performing a multiplication operation. MULSCC treats the less significant 32 bits of both R[rs1] and the Y register as a single 64-bit, right-shiftable doubleword register. The least significant bit of R[rs1] is treated as if it were adjacent to bit 31 of the Y register. The MULSCC instruction adds, based on the least significant bit of Y.

Multiplication assumes that the Y register initially contains the multiplier, R[rs1] contains the most significant bits of the product, and R[rs2] contains the multiplicand. Upon completion of the multiplication, the Y register contains the least significant bits of the product.

Note A standard MULSCC instruction has rs1 = rd.

1. The multiplicand is R[rs2] if i = 0, or **sign_ext**(simm13) if i = 1.
2. A 32-bit value is computed by shifting R[rs1] right by one bit with “CCR.icc.n xor CCR.icc.v” replacing bit 31 of R[rs1]. (This is the proper sign for the previous partial product.)
3. If the least significant bit of Y = 1, the shifted value from step (2) and the multiplicand are added. If the least significant bit of the Y = 0, then 0 is added to the shifted value from step (2).
4. The following values are set to the register.

Register field	Value set by MULSCC
CCR.icc	Updated according to the addition performed in step 3.
R[rd]<63:33>	0
R[rd]<32>	CCR.icc.c
R[rd]<31:0>	The lower 32 bits of R[rd] of step 3.
CCR.xcc.n	0
CCR.xcc.v	0
CCR.xcc.c	0
CCR.xcc.z	Set to 1 if R[rd]=0. Otherwise, set to 0.

Compatibility Note In SPARC V9 and JPS1, the upper 32 bits of R[rd] and CCR.xcc were undefined.

5. The Y register is shifted right by one bit, with the least significant bit of the unshifted R[rs1] replacing bit 31 of Y.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	XAR.v = 1

7.68. Multiply and Divide (64-bit)

Refer to Section 7.95 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	All	XAR.v = 1
<i>division_by_zero</i>	SDIVX, UDIVX	Divisor is 0.

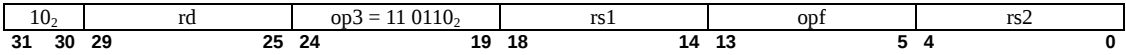
7.69. No Operation

Refer to Section 7.96 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	$iw<29:25, 21:0> \neq 0$
<i>illegal_action</i>	All	$XAR.v = 1$

7.72. Partitioned Add

Instruction	opf	Operation	HPC-ACE	Assembly Language Syntax
			Regs. SIMD	
PADD32	0 1000 1001 ₂	Two 32-bit Add Two 32 bit addition		padd32 <i>reg_{rs1}</i> , <i>reg_{rs2}</i> , <i>reg_{rd}</i>



Description PADD32 performs two 32-bit partitioned adds between the corresponding fixed-point values contained in the source operands (the 64-bit integer registers specified by rs1 and rs2). The result is placed in the 64-bit destination register specified by rd.

Exception	Condition
<i>illegal_action</i>	XAR.v = 1

7.73. Pixel Component Distance (with Accumulation)

Refer to Section 7.101 in UA2011.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

7.74. Population Count

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
POPC	10 1110 ₂	Population Count	✓		popc	<i>reg_or_imm, reg_{rd}</i>

Refer to Section 7.103 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	• $iw<18:14> \neq 0$. • $i = 0$ and $iw<12:5> \neq 0$.
<i>illegal_action</i>	XAR.v = 1

7.75. Prefetch

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
PREFETCH	10 1101 ₂	Prefetch Data	✓		prefetch [address], prefetch_fcn
PREFETCHA ^{PASI}	11	Prefetch Data from	✓		prefetch [regaddr], imm_asi , prefetch_fcn
	1101 ₂	Alternate Space			prefetch [reg_plus_imm] %asi, prefetch_fcn

Refer to Section 7.104 in UA2011.

An arbitrary address can be specified for the address specified by the instruction. One cache line (128 bytes) or two cache lines (256 bytes), as specified by the instruction, are copied. The *mem_address_not_aligned* exception is never generated.

The PREFETCH{A} instruction becomes a NOP when the specified address is non-cacheable or in an undefined cacheable space.

ASIs that can be specified by the PREFETCHA instruction are shown in Table 7-20. When an ASI other than those listed below is specified, the PREFETCHA instruction becomes a NOP.

Table 7-20 **ASIs valid for PREFETCHA**

ASI_PRIMARY	ASI_PRIMARY_LITTLE
ASI_SECONDARY	ASI_SECONDARY_LITTLE

The prefetch instruction has no side effects other than bringing a data block into cache.

The prefetch instruction might not be executed due to a lack of hardware resources (prefetch lost). Whether a prefetch instruction has been executed or lost cannot be known.

7.75.1. Prefetch Variants

Table 7-21 shows available fcns on SPARC64™ X / SPARC64™ X+ and describes their operation.

Table 7-21 fcns for PREFETCH and PREFETCHA

fcn	JPS1 and UA2011 Definition	Operation on SPARC64™ X / SPARC64™ X+
0	Frequently used data is prefetched for reading.	128-byte data is copied into the L1 data cache.
1	Infrequently used data is prefetched for reading.	128-byte data is copied into the U2 cache.
2	Frequently used data is prefetched for writing.	128-byte data is copied into the L1 data cache with exclusive ownership.
3	Infrequently used data is prefetched for writing.	128-byte data is copied into the U2 cache with exclusive ownership.
4	Page mapping performed by privileged software.	NOP
5 - 15 (05 ₁₆ - 0F ₁₆)	An <i>illegal_instruction</i> exception is detected.	An <i>illegal_instruction</i> exception is detected.
16 - 19 (10 ₁₆ - 13 ₁₆)	Implementation dependent	NOP
20 (14 ₁₆)	Frequently used data is prefetched for reading. Strong prefetch.	128-byte data is copied into the L1 data cache. Strong prefetch.
21 (15 ₁₆)	Infrequently used data is prefetched for reading. Strong prefetch.	128-byte data is copied into the U2 cache. Strong prefetch.
22 (16 ₁₆)	Frequently used data is prefetched for writing. Strong prefetch.	128-byte data is copied into the L1 data cache with exclusive ownership. Strong prefetch.
23 (17 ₁₆)	Infrequently used data is prefetched for writing. Strong prefetch.	128-byte data is copied into the U2 cache with exclusive ownership. Strong prefetch.
24 - 28 (18 ₁₆ - 1C ₁₆)	Implementation dependent	NOP
29 (1D ₁₆)		256 byte data aligned on 256-byte boundary is copied into the U2 cache. Strong prefetch.
30 (1E ₁₆)		NOP
31 (1F ₁₆)		256-byte data aligned on 256-byte boundary is copied into the U2 cache with exclusive ownership. Strong prefetch.

7.75.2. Weak versus Strong Prefetches

Programming Note Strong prefetches might block subsequent load or store instructions. Therefore, strong prefetches should be used only when prefetched data is guaranteed to be accessed.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	When either of the following is true • A <i>reserved</i> field is not 0. • fcn = 5 - 15
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd $\neq$ 0

7.76. Read Ancillary State Register (RDASR)

Instruction	rs1	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
RDY ^D	0	Read Y Register; deprecated (see A.71.9 in JPS1 Commonality)			rd %y, <i>reg_{rd}</i>
RDCCR	2	Read Condition Codes Register			rd %ccr, <i>reg_{rd}</i>
RDASI	3	Read ASI Register			rd %asi, <i>reg_{rd}</i>
RD ^P TICK ^{NPT}	4	Read Tick Register			rd %tick, <i>reg_{rd}</i>
RDPC	5	Read Program Counter			rd %pc, <i>reg_{rd}</i>
RDFPRS	6	Read Floating-Point Registers Status Register			rd %fprs, <i>reg_{rd}</i>
MEMBAR	15	MEMBAR (page 141).			
RD ^P PCR ^{PCR}	16	Read Performance Control Registers (PCR)			rd %pcr, <i>reg_{rd}</i>
RD ^P PIC ^{PCR}	17	Read Performance Instrumentation Counters (PIC)			rd %pic, <i>reg_{rd}</i>
RDGSR	19	Read Graphic Status Register (GSR)			rd %gsr, <i>reg_{rd}</i>
RD ^P STICK ^{NPT}	24	Read System TICK Register			rd %stick, <i>reg_{rd}</i>
RDXASR	30	Read XASR			rd %xasr, <i>reg_{rd}</i>

RDASR copies the contents of an Ancillary State Register to R[rd]. For descriptions of Ancillary State Registers, see Section 5.5 (page 31). Though MEMBAR corresponds to rs1 = 15, MEMBAR is described on page 141 and not covered in this section.

- RDY reads the Y register into R[rd]. Instructions that reference the Y register should be avoided. (deprecated)
- RDFPRS waits for all pending FPods to complete before reading the FPRS register.
- When PCR.priv = 1, an attempt to execute RDPCR or RDPIC in non-privileged mode causes a *privileged_action* exception.

For exceptions when rd = 15, refer to MEMBAR.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	When any of the following are true • rs1 = 1, 7 - 14, 18, 20 - 21, 26 - 29 • i = 1 • iw<12:0> ≠ 0
<i>fp_disabled</i>	RDGSR	PSTATE.PEF = 0 or FPRS.FEF = 0
<i>illegal_action</i>	All	XAR.v = 1
<i>privileged_action</i>	RDTICK	PSTATE.priv = 0 and TICK.npt = 0
	RDPCR, RDPIC	PSTATE.priv = 0 and PCR.priv = 1
	RDSTICK	PSTATE.priv = 0 and STICK.npt = 1

7.79. Return

Refer to Section 7.110 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw_{\langle 29:25, 12:5 \rangle} \neq 0$ $i = 1$ and $iw_{\langle 29:25 \rangle} \neq 0$
<i>illegal_action</i>	$XAR.v = 1$
<i>fill_n_normal</i>	
<i>fill_n_other</i>	
<i>mem_address_not_aligned</i>	Effective address is not 4-byte aligned
<i>control_transfer_instruction</i>	$PSTATE.tct = 1$

7.80. SAVE and RESTORE

Refer to Section 7.111 and Section 7.107 in UA2011.

<SAVE>

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$
<i>spill_n_normal</i>	
<i>spill_n_other</i>	
<i>clean_window</i>	

<RESTORE>

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$
<i>fill_n_normal</i>	
<i>fill_n_other</i>	

7.82. Signed Divide (64-bit $\div$ 32-bit)

Refer to Section 7.113 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$
<i>division_by_zero</i>	Divisor is zero

7.83. SETHI

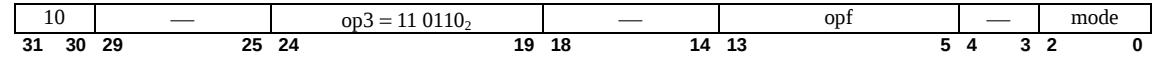
Refer to Section 7.114 in UA2011.

Exception	Condition
<i>illegal_action</i>	XAR.v = 1

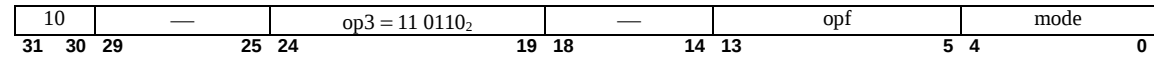
7.85. Set Interval Arithmetic Mode

Instru ction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
SIAM	0 1000 0001 ₂	Set the interval arithmetic mode fields in the GSR .			siam <i>siam_mode</i>
SDIAM	0 1000 0101 ₂	Set the decimal interval arithmetic mode fields in the GSR			sdiam <i>siam_mode</i>

SIAM



SDIAM



Refer to Section 7.116 in UA2011.

Description The SIAM instruction sets the GSR.im and GSR.irnd fields as follows:

GSR.im ← mode<2>
GSR.irnd ← mode<1:0>

The SDIAM instruction sets the GSR.dim and GSR.dirnd fields as follows:

GSR.dim ← mode<4>
GSR.dirnd ← mode<2:0>

Exception	Target instruction	Condition
<i>illegal_instruction</i>	SIAM	A <i>reserved</i> field is not 0.
	SDIAM	When either of the following is true - A <i>reserved</i> field is not 0 - mode<3> ≠ 0
<i>fp_disabled</i>	All	PSTATE.PEF = 0 or FPRS.FEF = 0

7.87. Shift

Instruction	op3	X	r	Operation	HPC-ACE		Assembly Language Syntax	
					Regs	SIMD		
SLL	10 0101 ₂	0	0	Shift left logical – 32 bits			sll	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>
SRL	10 0110 ₂	0	0	Shift right logical – 32 bits			srl	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>
SRA	10 0111 ₂	0	0	Shift right arithmetic – 32 bits			sra	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>
SLLX	10 0101 ₂	1	0	Shift left logical – 64 bits			sllx	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>
SRLX	10 0110 ₂	1	0	Shift right logical – 64 bits			srlx	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>
SRAX	10 0111 ₂	1	0	Shift right arithmetic – 64 bits			srax	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>
ROLX	10 0101 ₂	1	1	Rotate left – 64 bits			rolx	<i>reg_{rs1}</i> , <i>reg_or_shcnt</i> , <i>reg_{rd}</i>

10 ₂	rd	op3	rs1	i = 0	x	r	—	rs2
10 ₂	rd	op3	rs1	i = 1	x = 0	r = 0	—	shcnt32
10 ₂	rd	op3	rs1	i = 1	x = 1	r	—	shcnt64
31 30 29 25 24 19 18 14 13 12 11 10 6 5 4 0								

Refer to Section 7.117 in UA2011.

ROLX rotates all 64 bits of the value in R[rs1] left(towards the higher-order bit positions) by the number of bits specified by the shift count. Unlike shift instructions, the rotate instruction replaces the vacated positions on the right (the lower-order bit positions) with the overflow bits from the left. The rotated result is written to R[rd].

Compatibility Note ROLX is a new instruction defined for SPARC64™ X / SPARC64™ X+. Bit 11 of the instruction word is *reserved* in SPARCV9.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0.
	SLL, SLLX, ROLX	x = 0 and r = 1
	SRL, SRA, SRLX, SRAX	r = 1
<i>illegal_action</i>	All	XAR.v = 1

7.88. Signed Multiply (32-bit)

Refer to Section 7.118 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$

7.89. Sleep

Instruction	Opf	Operation	HPC-ACE	Assembly Language
			Regs. SIMD	Syntax
SLEEP	0 1000 0011 ₂	VCPU is stopped during the fixed time.		sleep

10 ₂	—	op3 = 11 0110 ₂	—	Opf	—
31 30 29		25 24	19 18	14 13	5 4 0

The SLEEP instruction stops the VCPU for a fixed period of time, unless there is a pending interrupt.

The stopped VCPU restarts execution when either of the following conditions is true.

- A fixed period of time, which is implementation dependent, has passed.
- An interrupt is pending or has occurred.

Programming Note Software should not expect the SLEEP instruction to always stop a VCPU for a fixed amount of time.

Compatibility Note On SPARC64 VIIIfx, and earlier processors, execution was restarted when the interrupt occurs. In SPARC64™ X / SPARC64™ X+, execution is restarted when an interrupt is pending (for example, when the processor cannot accept interrupts). That is, execution may restart when an interrupt has not occurred.

Exception	Condition
<i>illegal_instruction</i>	A <i>reserved</i> field is not 0.
<i>illegal_action</i>	XAR.v = 1

7.91. Store Barrier

Instruction	op3	Operation	HPC-ACE	Assembly Language
			Regs. SIMD	Syntax
STBAR	10 1000 ₂	nop		stbar

10 ₂	0 0000 ₂	op3 = 10 1000 ₂	01111 ₂	i = 0	—
31 30 29	25 24	19 18	14 13 12	0	

On SPARC64™ X / SPARC64™ X+, Store Barrier (STBAR) behaves as a NOP since the hardware memory model always enforces the semantics of this instruction for all memory accesses.

Exception	Condition
<i>illegal_instruction</i>	iw<12:0> ≠ 0
<i>illegal_action</i>	XAR.v = 1

7.92. Store Integer

Refer to Section 7.119 in UA2011.

Exception	Target Instruction	Condition
<i>illegal_instruction</i>	All	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq 0$ • XAR.urs2 $\neq 0$ • XAR.urs3<2> $\neq 0$ • XAR.urd $\neq 0$
<i>mem_address_not_aligned</i>	STH	Effective address is not 2-byte aligned
	STW	Effective address is not 4-byte aligned
	STX	Effective address is not 8-byte aligned
<i>VA_watchpoint</i>	All	
<i>DAE_privilege_violation</i>	All	
<i>DAE_nfo_page</i>	All	

7.93. Store Integer into Alternate Space

Refer to Section 7.120 in UA2011.

Exception	Target Instruction	Condition
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd $\neq$ 0
<i>mem_address_not_aligned</i>	STHA	Effective address is not 2-byte aligned
	STWA	Effective address is not 4-byte aligned
	STXA	Effective address is not 8-byte aligned
<i>privileged_action</i>	All	
<i>VA_watchpoint</i>	All	
<i>DAE_invalid_asl</i>	All	
<i>DAE_privilege_violation</i>	All	
<i>DAE_nfo_page</i>	All	

7.94. Block Initializing Store

UA2011 defines `ASI_STBI_*`. On SPARC64™ X / SPARC64™ X+, if `ASI_STBI_*` is specified for the STBA, STHA, STWA, STXA, and STTWA instructions, these stores behave as normal store instructions. For example, if `ASI_STBI_P` is specified for STBA, STBA behaves as if `ASI_P` was specified.

The behavior of Block Initializing Stores is shown below.

ASI number	ASI name	Integer store (STBA, STHA, STWA, STXA, STTWA) operation
E2 ₁₆	ASI_STBI_P	ASI_P
E3 ₁₆	ASI_STBI_S	ASI_S
EA ₁₆	ASI_STBI_PL	ASI_PL
EB ₁₆	ASI_STBI_SL	ASI_SL

Only *DAE_invalid_ASI* and *mem_address_not_aligned* exceptions are generated. *DAE_** exceptions, except for *DAE_invalid_ASI*, do not occur.

7.95. Block Store

Instruction	ASI	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
STBLOCKF	F0 ₁₆	64 bytes block store is executed to primary address space.	✓		stda <i>freg_{rd}</i> , [<i>regaddr</i>] ASI_BLK_P stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
STBLOCKF	F1 ₁₆	64 bytes block store is executed to secondary address space.	✓		stda <i>freg_{rd}</i> , [<i>regaddr</i>] ASI_BLK_S stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
STBLOCKF	F8 ₁₆	64 bytes block store is executed to primary address space. Little endian.	✓		stda <i>freg_{rd}</i> , [<i>regaddr</i>] ASI_BLK_PL stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
STBLOCKF	F9 ₁₆	64 bytes block store is executed to secondary address space. Little endian.	✓		stda <i>freg_{rd}</i> , [<i>regaddr</i>] ASI_BLK_SL stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
STBLOCKF	E0 ₁₆	64 bytes block committing store is executed to primary address space.	✓		stda <i>freg_{rd}</i> , [<i>regaddr</i>] ASI_BLK_COMMIT_P stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
STBLOCKF	E1 ₁₆	64 bytes block committing store is executed to secondary address space.	✓		stda <i>freg_{rd}</i> , [<i>regaddr</i>] ASI_BLK_COMMIT_S stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi

Refer to Section 7.121 in UA2011.

The effective address is "R[rs1] + R[rs2]" if i = 0, or "R[rs1] + **sign_ext**(simm13)" if i = 1".

On SPARC64™ X / SPARC64™ X+, block store instruction and block committing store instruction behave exactly the same.

STBLOCKF on SPARC64™ X / SPARC64™ X+ follow TSO. That is, the ordering between the preceding and following load/store/atomic instructions and the 8-bytes stores comprising the block store conforms to TSO.

STBLOCKF on SPARC64™ X / SPARC64™ X+ preserves the order of register accesses in the same manner as any other store instruction.

The cache behavior of STBLOCKF is the same as for a normal store. If there is data in L1D cache, then the block store writes to the L1D cache. If there is no data in the L1D cache, the data is loaded into the L1D cache and then written.

A non-cacheable address can be specified for STBLOCKF.

A *VA_watchpoint* exception is detected only for the first eight bytes accessed by a STBLOCKF instruction.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	Register number specified by rd is not a multiple of 8
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0
<i>mem_address_not_aligned</i>	All	Address not aligned on a 64-byte boundary.
<i>VA_watchpoint</i>	All	On access to eight lowest bytes only. Refer to 12.5.1.62.
<i>DAE_privilege_violation</i>	ASI E0 ₁₆ , E1 ₁₆ , F0 ₁₆ , F1 ₁₆ , F8 ₁₆ , and F9 ₁₆ .	PSTATE.priv = 0 and TTE.p = 1 Refer to 12.5.1.8
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.96. Store Floating-Point

Instruction	op3	rd ^{xiii}	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
STF	10 0100 ₂	0 – 31	Store single floating point register (XAR.v = 0)			st	<i>freg_{rd}</i> , [<i>address</i>]
STF	10 0100 ₂	0 – 126, 256 – 382	Store double floating point register (XAR.v = 1)	✓	✓	st	<i>freg_{rd}</i> , [<i>address</i>]
STDF	10 0111 ₂	0 – 126, 256 – 382	Store double floating point register	✓	✓	std	<i>freg_{rd}</i> , [<i>address</i>]
STQF	10 0110 ₂	0 – 126, 256 – 382	Store quad floating point register to memory	✓		stq	<i>freg_{rd}</i> , [<i>address</i>]

non-SIMD execution

Refer to Section 7.122 in UA2011.

STF copies 4 bytes in F[rd] to an address aligned on a 4-byte boundary. When XAR.v = 0, STF copies the contents of a single-precision floating-point register. When XAR.v = 1, STF copies the upper 4 bytes of a double-precision floating-point register.

The STQF instruction is defined by SPARC V9 but it is not implemented on SPARC64™ X / SPARC64™ X+. If STQF is executed, an *illegal_instruction* exception occurs.

SIMD execution On SPARC64™ X / SPARC64™ X+, STF and STDF can be executed as SIMD instructions. A SIMD STF and SIMD STDF simultaneously execute basic and extended stores for single-precision and double-precision data, respectively. Refer to Section 5.5.14 (page 35) for details on how to specify the registers.

A SIMD STF copies the upper 4 bytes of Fd[rd] to the lower 4 bytes of the address and copies the upper 4 bytes of Fd[rd + 256] to the upper 4 bytes of the address. The address must be aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

A SIMD STDF copies the 8 bytes of Fd[rd] to the lower 8 bytes of the address and copies the 8 bytes of Fd[rd + 256] to the upper 8 bytes of the address. The address must be aligned on a 16-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

Note A SIMD STDF that accesses data aligned on a 4-byte boundary but not an 8-byte boundary does not cause an *STDF_mem_address_not_aligned* exception.

SIMD STF and SIMD STDF can only write to cacheable address. An attempt to access a non-cacheable space causes a *DAE_nc_page* exception.

Like non-SIMD store instructions, memory access semantics adhere to TSO. SIMD STF and SIMD STDF simultaneously execute basic and extended stores; however, the ordering between the basic and extended stores conforms to TSO.

A *VA_watchpoint* exception can be detected in either the basic or extended operation of a SIMD STF or SIMD STDF instruction.

^{xiii} Encoding defined in 5.3.1 "Floating-Point Register Number Encoding" (page 26).

Exception	Target instruction	Condition
<i>illegal_instruction</i>	STF, STDF	$i = 0$ and a <i>reserved</i> is not 0.
	STQF	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	STF, STDF	XAR.v = 1 and any of the following are true • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd<1> $\neq$ 0 • XAR.simd = 1 and XAR.urd<2> $\neq$ 0
	STQF	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd<1> $\neq$ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	STQF	rd<1> $\neq$ 0
<i>STDF_mem_address_not_aligned</i>	STDF	Address aligned on 4-byte boundary but not 8-byte boundary when XAR.v = 0 or XAR.simd = 0.
<i>mem_address_not_aligned</i>	STF	When either of the following is true • Address not aligned on 4-byte boundary when XAR.v = 0 or XAR.simd = 0 • Address not aligned on 8-byte boundary when XAR.v = 1 and XAR.simd = 1
	STDF	When either of the following is true • Address not aligned on 4-byte boundary when XAR.v = 0 or XAR.simd = 0 • Address not aligned on 16-byte boundary when XAR.v = 1 and XAR.simd = 1
	STQF	Address not aligned on 4-byte boundary.
<i>VA_watchpoint</i>	All	Refer to the description and 12.5.1.62.
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	Access to non-cacheable space when XAR.v = 1 and XAR.simd = 1
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.97. Store Floating-Point into Alternate Space

Instruction	op3	rd ^{xiv}	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
STFA ^{PASI}	11 0100 ₂	0 – 31	Store single floating point register into alternate space (xar.v = 0)			sta <i>freg_{rd}</i> , [<i>address</i>] imm_asi	sta <i>freg_{rd}</i> , [<i>address</i>] %asi
STFA ^{PASI}	11 0100 ₂	0 – 126, 256 – 382	Store double floating point register into alternate space (xar.v = 1)	✓	✓	sta <i>freg_{rd}</i> , [<i>address</i>] imm_asi	sta <i>freg_{rd}</i> , [<i>address</i>] %asi
STDFA ^{PASI}	11 0111 ₂	0 – 126, 256 – 382	Store double floating point register into alternate space	✓	✓	stda <i>freg_{rd}</i> , [<i>address</i>] imm_asi	stda <i>freg_{rd}</i> , [<i>address</i>] %asi
STQFA ^{PASI}	11 0110 ₂	0 – 126, 256 – 382	Store quad floating point register into alternate space	✓		stqa <i>freg_{rd}</i> , [<i>address</i>] imm_asi	stqa <i>freg_{rd}</i> , [<i>address</i>] %asi

non-SIMD execution

Refer to Section 7.123 in UA2011.

STFA copies the 4-bytes in F[rd] to the 4-byte aligned address in the specified alternate space. When xar.v = 0, STFA copies the content of a single-precision floating-point register. When xar.v = 1, STFA copies the upper 4 bytes of a double-precision floating-point register.

The STQFA instruction is defined by SPARC V9 but is not implemented on SPARC64™ X / SPARC64™ X+. If STQFA is executed, an *illegal_instruction* exception occurs.

SIMD execution On SPARC64™ X / SPARC64™ X+, STFA and STDFA can be executed as SIMD instructions. SIMD STFA and SIMD STDFA simultaneously execute basic and extended stores for single-precision and double-precision data. Refer to Section 5.5.14 (page 35) for details on how to specify the registers.

SIMD STFA copies the upper 4 bytes of Fd[rd] to the lower 4 bytes of the address and copies the upper 4 bytes of Fd[rd + 256] to the upper 4 bytes of the address. The address must be aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

SIMD STDFA copies the 8bytes of Fd[rd] to the lower 8 bytes of the address and copies the 8 bytes of Fd[rd + 256] to the upper 8 bytes of the address. The address must be aligned on a 16-byte boundary. Misaligned addresses cause a *mem_address_not_aligned* exception.

Note A SIMD STDFA that access data aligned on a 4-byte boundary but not an 8-byte boundary does not cause an *STDF_mem_address_not_aligned* exception. Unlike SIMD LDDFA, a SIMD STDFA that accesses data aligned on an 8-byte boundary but not a 16-byte boundary causes a *mem_address_not_aligned* exception.

^{xiv} Encoding defined in 5.3.1 "Floating-Point Register Number Encoding" (page 26).

SIMD STFA and SIMD STDFA can only write to cacheable addresses. An attempt to access a non-cacheable space causes a *DAE_nc_page* exception. If a nontranslating ASI is specified for a SIMD STFA or SIMD STDFA, a *DAE_invalid_ASI* exception will occur.

Like non-SIMD store instructions, memory access semantics adhere to TSO. SIMD STFA and SIMD STDFA simultaneously execute basic and extended stores; however, the ordering between the basic and extended stores conforms to TSO.

A *VA_watchpoint* exception can be detected in either the basic or extended operation of a SIMD STFA or SIMD STDFA.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	STQFA	Always For this instruction, exceptions with priority lower than <i>illegal_instruction</i> are intended for emulation.
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	STFA, STDFA	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	STQFA	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0
<i>fp_exception_other</i> (FSR.ftt = <i>invalid_fp_register</i>)	STQFA	rd<1> ≠ 0
<i>STDF_mem_address_not_aligned</i>	STDFA	Address aligned on 4-byte boundary but not 8-byte boundary when XAR.v = 0 or XAR.simd = 0
<i>mem_address_not_aligned</i>	STFA	When either of the following is true • Address not aligned on 4-byte boundary when XAR.v = 0 or XAR.simd = 0 • Address not aligned on 8-byte boundary when XAR.v = 1 and XAR.simd = 1
	STDFA	When either of the following is true • Address not aligned on 4-byte boundary when XAR.v = 0 or XAR.simd = 0 • Address not aligned on 16-byte boundary when XAR.v = 1 and XAR.simd = 1
	STQFA	
<i>privileged_action</i>	All	Refer to 12.5.1.49
<i>VA_watchpoint</i>	All	Refer to the description
<i>DAE_invalid_asi</i>	All	Refer to the description and 12.5.1.5
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	Access to non-cacheable space when XAR.v = 1 and XAR.simd = 1
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.98. Store Floating-Point Register on Register Condition (for SPARC64™ X)

Compatibility Note For the specification of this instruction on SPARC64™ X+, refer to page 248.

Instruction	op3	rs2, rd	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
STFR	10 1100 ₂	0 – 31	Store single floating point register on condition (xar.v = 0)			stfr	<i>freg_{rd}</i> , <i>freg_{rs2}</i> , [<i>reg_{rs1}</i>]
STFR	10 1100 ₂	0 – 126, 256 – 382 ^{xv}	Store single floating point register on condition (xar.v = 1)	✓	✓	stfr	<i>freg_{rd}</i> , <i>freg_{rs2}</i> , [<i>reg_{rs1}</i>]
STDFR	10 1111 ₂	0 – 126, 256 – 382 ^{xv}	Store double floating point register on condition	✓	✓	stdfr	<i>freg_{rd}</i> , <i>freg_{rs2}</i> , [<i>reg_{rs1}</i>]

11 ₂	rd		op3		rs1		i = 1	—		rs2	
31 30 29	25 24		19 18		14 13 12			5 4		0	

Programming Note This instruction does not execute a store operation if the MSB of the corresponding register F[rs2] or F[rs2+256] is 0. However, certain exception can still occur.

Non-SIMD execution

When XAR.v = 0 and the MSB (bit 31) of Fs[rs2] is 1, STFR copies the 4bytes of the single-precision register Fs[rd] to the specified address, which should be aligned on a 4-byte boundary. When XAR.v = 1, XAR.simd = 0, and MSB (bit 63) of Fd[rs2] is 1, STFR copies the upper 4bytes of the double-precision register Fd[rd] to the specified address, which should be aligned on a 4-byte boundary.

When the MSB (bit 63) of Fd[rs2] is 1, STDFR copies the 8 bytes of the double-precision register Fd[rd] to the specified address, which should be aligned on a 4-byte boundary.

These floating-point store instructions use implicit ASIs (refer to Section 6.3.1.3 in UA2011) to access memory. The effective write address is "R[rs1]".

STFR and STDFR cause a *mem_address_not_aligned* exception when writing to an address that is not aligned on a word boundary.

When executing a non-SIMD STDFR, the address needs to be aligned on a word boundary. However, if the address is aligned on a word boundary but is not aligned on a doubleword boundary, a *STDF_mem_address_not_aligned* exception will occur. The trap handler must emulate the STDFR instruction when this exception occurs.

Regardless of whether the store operation is actually executed, a *VA_watchpoint* exception is detected for STFR and STDFR if the address matches.

SIMD execution STFR and STDFR support SIMD execution on SPARC64™ X. SIMD STFR and SIMD STDFR simultaneously execute basic and extended stores for single-precision and double-precision data, respectively. Refer to Section 5.5.14 (page 35) for details on how to specify the registers.

^{xv} Encoding defined in 5.3.1 "Floating-Point Register Number Encoding" (page 26)

A SIMD STFR copies the upper 4 bytes of $Fd[rd]$ to the lower 4 bytes of the address when $XAR.v = 1$, $XAR.simd = 1$, and the MSB (bit 63) of $Fd[rs2]$ is 1, and copies the upper 4 bytes of $Fd[rd + 256]$ to the upper 4 bytes of the address when $XAR.v = 1$, $XAR.simd = 1$, and the MSB (bit 63) of $Fd[rs2+256]$ is 1. The address must be aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

A SIMD STDFR copies $Fd[rd]$ to the lower 8 bytes of the address when $XAR.v = 1$, $XAR.simd = 1$, and the MSB (bit 63) of $Fd[rs2]$ is 1, and copies $Fd[rd + 256]$ to the upper 8 bytes of the address when $XAR.v = 1$, $XAR.simd = 1$, and the MSB (bit 63) of $Fd[rs2+256]$ is 1. The address must be aligned on a 16-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

Note A SIMD STDFR does not cause a *STDF_mem_address_not_aligned* exception when writing to an address that is aligned on a 4-byte boundary but not an 8-byte boundary.

SIMD STFR and SIMD STDFR can only write to cacheable address spaces. An attempt to write a non-cacheable space causes a *DAE_nc_page* exception.

Like non-SIMD store instructions, memory access semantics adhere to TSO. SIMD STFR and SIMD STDFR simultaneously execute basic and extended stores; however, the ordering between the basic and extended stores conforms to TSO.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	$i = 0$ or a <i>reserved</i> field is not 0.
<i>fp_disabled</i>	All	$PSTATE.pef = 0$ or $FPRS.fef = 0$
<i>illegal_action</i>	All	$XAR.v = 1$ and any of the following are true • $XAR.urs1 \neq 0$ • $XAR.urs2<1> \neq 0$ • $XAR.urs3<2> \neq 0$ • $XAR.urd<1> \neq 0$ • $XAR.simd = 1$ and $XAR.urs2<2> \neq 0$ • $XAR.simd = 1$ and $XAR.urd<2> \neq 0$
<i>STDF_mem_address_not_aligned</i>	STDFR	Store to address aligned on 4-byte boundary but not an 8-byte boundary when $XAR.v = 0$ or $XAR.v = 1$ and $XAR.simd = 0$
<i>mem_address_not_aligned</i>	STFR	When either of the following is true • Store to address not aligned on 4-byte boundary when $XAR.v = 1$ and $XAR.simd = 0$, or $XAR.v = 0$ • Store to address not aligned on 8-byte boundary when $XAR.v = 1$ and $XAR.simd = 1$
	STDFR	When either of the following is true • Store to address not aligned on 4-byte boundary when $XAR.v = 1$ and $XAR.simd = 0$, or $XAR.v = 0$. • Store to address not aligned on 16-byte boundary when $XAR.v = 1$ and $XAR.simd = 1$
<i>VA_watchpoint</i>	All	Refer to the description and 12.5.1.62
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nc_page</i>	All	Access to non-cacheable space when $XAR.v = 1$ and $XAR.simd = 1$
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.99. Store Partial Floating-Point

Refer to Section 7.125 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 1$
<i>fp_disabled</i>	$PSTATE.pef = 0$ or $FPRS.fef = 0$
<i>illegal_action</i>	$XAR.v = 1$
<i>STDF_mem_address_not_aligned</i>	Effective address is 4-byte aligned but not 8-byte aligned
<i>mem_address_not_aligned</i>	Effective address is 4-byte aligned
<i>VA_watchpoint</i>	
<i>DAE_privilege_violation</i>	
<i>DAE_nfo_page</i>	

7.100. Store Short Floating-Point

Refer to Section 7.126 in UA2011.

Exception	Target Instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1
<i>mem_address_not_aligned</i>	ASI 0xD2, 0xD3, 0xDA, 0xDB	Effective address is not 2-byte aligned
<i>VA_watchpoint</i>	All	
<i>DAE_privilege_violation</i>	All	
<i>DAE_nfo_page</i>	All	

7.101. Store Integer Twin Word

Refer to Section 7.127 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	• rd is an odd numbered register • i = 0 and iw<12:5> ≠ 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>mem_address_not_aligned</i>	Effective address is not 8-byte aligned
<i>VA_watchpoint</i>	
<i>DAE_privilege_violation</i>	
<i>DAE_nfo_page</i>	

7.102. Store Integer Twin Word into Alternate Space

Refer to Section 7.128 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	rd is an odd numbered register
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2 $\neq$ 0 • XAR.urs3<2> $\neq$ 0 • XAR.urd $\neq$ 0
<i>mem_address_not_aligned</i>	Effective address is not 8-byte aligned
<i>privileged_action</i>	
<i>VA_watchpoint</i>	
<i>DAE_invalid_as</i>	
<i>DAE_privilege_violation</i>	
<i>DAE_nfo_page</i>	

7.103. Store Floating-Point State Register

Instruction	op3	rd	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
STFSR ^D	10 0101 ₂	0	Store FSR (Only lower 32 bits)	✓		st %fsr, [address]
STXFSR	10 0101 ₂	1	Store FSR	✓		stx %fsr, [address]
—	10 0101 ₂	2 - 31	reserved			

Refer to Section 7.124 and Section 7.129 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	STFSR, STXFSR	i = 0 and the <i>reserved</i> field is not 0.
	—	rd = 2–31
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
<i>mem_address_not_aligned</i>	STFSR	Address not aligned on 4-byte boundary
	STXFSR	Address not aligned on 8-byte boundary.
<i>VA_watchpoint</i>	All	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.104. Subtract

Refer to Section 7.130 in UA2011.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	A <i>reserved</i> field is not 0. (i = 0 and iw<12:5> = 0)
<i>illegal_action</i>	All	XAR.v = 1

7.105. Swap Register with Memory

Refer to Section 7.131 and Section 7.132 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq 0$ • XAR.urs2 $\neq 0$ • XAR.urs3<2> $\neq 0$ • XAR.urd $\neq 0$
<i>mem_address_not_aligned</i>	Effective address is not 4-byte aligned
<i>VA_watchpoint</i>	
<i>DAE_privilege_violation</i>	
<i>DAE_nfo_page</i>	

7.106. Set XAR (SXAR)

Instruction	op2	cmb	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
SXAR1	111 ₂	0	Set XAR for the following instruction			sxar1
SXAR2	111 ₂	1	Set XAR for the following two instructions			sxar2

00 ₂	cmb	f_simd	f_urd	op2 = 111 ₂	f_urs1	f_urs2	f_urs3	s_simd	s_urd	s_urs1	s_urs2	s_urs3
31 30 29	28	27	25 24	22 21 19 18	16 15 13	12	11 9 8	6 5	3 2	0		

Description SXAR updates the XAR register. XAR holds values for up to two instructions. SXAR1 sets values for one instruction, and SXAR2 sets values for two instructions. Fields that start with *f_* are used by the first instruction executed after SXAR, and fields that start with *s_* are used by the second instruction executed after SXAR.

The fields of SXAR1 that starts with *s_* must be 0. An *illegal_instruction* exception will occur if a value other than 0 is specified.

Compatibility Note In SPARC64 VIIIfx, values other than 0 in fields starting with *s_* were ignored for SXAR1.

SXAR modifies the one or two instructions that are immediately executed after SXAR. The SXAR instruction is used to specify the HPC-ACE floating-point registers on SPARC64™ X / SPARC64™ X+, HPC-ACE SIMD operations with floating-point registers, and disabling hardware prefetch for memory access instructions. Even if the instruction fields, **_simd*, **_urs1*, **_urs2*, **_urs3* and **_urd* are all specified as 0, XAR.f_v is set to 1 after SXAR1 is executed; both XAR.f_v and XAR.s_v are set to 1 after SXAR2 is executed.

Performance may suffer if the SXAR instruction and the instructions that it modifies are not contiguous in memory. For example, if SXAR is placed in the delay slot of a branch instruction or a TCC instruction is inserted after the SXAR.

SXAR itself is not XAR eligible. If XAR.v = 1 when executing SXAR, an *illegal_action* exception will occur.

Compatibility Note op = 00₂ and op2 = 111₂ are reserved in SPARC V9, but SPARC V8 defines FBCC in these opcodes. When running a SPARC V8 application on SPARC64™ X / SPARC64™ X+, there is the possibility of different behavior.

Programming Note The SXAR instruction word contains the value to be set in the XAR register but this value is not shown in the assembly syntax of SXAR. Instead, HPC-ACE features are indicated by mnemonic suffixes appended to the instruction(s) that SXAR modifies, and the assembler generates the appropriate value for the SXAR instruction word.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	SXAR1	<i>s_*</i> ≠ 0
<i>illegal_action</i>	All	XAR.v = 1

7.107. Tagged Add and Subtract

Refer to Sections 7.133, 7.134, 7.136 and 7.137 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$

7.108. Trap on Integer Condition Code (Tcc)

Refer to Section 7.135 in UA2011.

The state of the XAR register does not affect the operation of the TCC instruction. Even if XAR.v = 1, no *illegal_action* exception is detected.

When the condition is not true, no trap is generated, but settings in the XAR register for one instruction are cleared. That is, if XAR.f_v = 1, then XAR.f_* are set to 0. If XAR.f_v = 0 and XAR.s_v = 1, then XAR.s_* are set to 0.

Programming Note XAR is ignored so that the TCC instruction can be inserted at any location in a sequence of instructions. This behavior is useful for implementing breakpoints for a debugger.

Whether the trap is generated or not depends on the SWTN. Table 7-22 shows this relationship.

Table 7-22 Trap generated given SWTN

Privilege level	SWTN	
	0 - 127	128 - 255
Non-privileged mode PSTATE.priv = 0	<i>trap_instruction</i>	— (Effective SWTN is only seven bits)

Note The Trap on Control Transfer feature is implemented on SPARC64™ X / SPARC64™ X+.

The values saved in TPC[TL] and TNPC[TL] are affected by the settings of PSTATE.am when the trap occurs.

Refer to pages 309 for more information about trap processing.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	All	When either of the following is true - A <i>reserved</i> field is not 0. - cc0 = 1
<i>control_transfer_instruction</i>	Except TN	Condition is true and PSTATE.tct = 1 Condition always true for TA
<i>trap_instruction</i>	Except TN	Refer to the description

7.109. Unsigned Divide (64-bit ÷ 32-bit)

Refer to Section 7.138 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$
<i>division_by_zero</i>	Divisor is zero

7.110. Unsigned Multiply (32-bit)

Refer to Section 7.139 in UA2011.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$

7.111. Write Ancillary State Register (WRASR)

Instruction	rd	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
WRY ^D	0	Write Y register. (deprecated.)			<code>wr reg_{rs1}, reg_or_imm, %y</code>
WRCCR	2	Write CCR register			<code>wr reg_{rs1}, reg_or_imm, %CCR</code>
WRASI	3	Write ASI register			<code>wr reg_{rs1}, reg_or_imm, %asi</code>
WRFPRS	6	Write FPRS register			<code>wr reg_{rs1}, reg_or_imm, %fprs</code>
WRPCR ^{PCR}	16	Write PCR register			<code>wr reg_{rs1}, reg_or_imm, %pcr</code>
WRPIC ^{PCR}	17	Write PIC register			<code>wr reg_{rs1}, reg_or_imm, %pic</code>
WRGSR	19	Write GSR register			<code>wr reg_{rs1}, reg_or_imm, %gsr</code>
WRPAUSE	27	Write PAUSE register			<code>wr reg_{rs1}, reg_or_imm, %pause</code>
WRXAR	29	Write XAR register			<code>wr reg_{rs1}, reg_or_imm, %xar</code>
WRXASR	30	Write XASR register			<code>wr reg_{rs1}, reg_or_imm, %xasr</code>

Refer to Section 7.141 in UA2011.

The result of a WRASR instruction takes affect immediately. The new setting is visible to downstream instructions.

- An attempt to set values other than 0 to reserved fields of XAR using the WRXAR instruction generates an *illegal_instruction* exception. Note that, in this case, the priority of an *illegal_action* exception is higher than the *illegal_instruction* exception.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	—	rd = 1, 4 – 5, 7 – 14, 18, 26 – 28
	All	i = 0 and iw<12:5> ≠ 0
<i>fp_disabled</i>	WRGSR	PSTATE.PEF = 0 or FPRS.FEF_ <u>=_0</u>
<i>illegal_action</i>	All	XAR.v = 1
<i>privileged_action</i>	WRPCR	PSTATE.priv = 0 and one of the following is true • PCR.priv = 1 • PCR.priv = 0 and PCR.priv is set to 1
	WRPIC	PSTATE.priv = 0 and PCR.priv = 1

7.114. Cache Line Fill with Undetermined Values

Compatibility Note This instruction (and corresponding ASIs) is left to ensure compatibility with SPARC64 VIIIfx

Instruction	ASI	op3	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
XFILL ^N	ASI_XFILL_P	01 1110 ₂	nop	✓		stxa <i>reg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
	ASI_XFILL_S	01 0111 ₂				stxa <i>reg_{rd}</i> , [<i>regaddr</i>] imm_asi
		11 0111 ₂				sttwa <i>reg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
						sttwa <i>reg_{rd}</i> , [<i>regaddr</i>] imm_asi
						stda <i>freg_{rd}</i> , [<i>reg_plus_imm</i>] %asi
					stda <i>freg_{rd}</i> , [<i>regaddr</i>] imm_asi	

11 ₂	rd	op3	rs1	i = 0	imm_asi	rs2
11 ₂	rd	op3	rs1	i = 1	simm13	
31 30 29	25 24	19 18	14 13 12	5 4	0	

Description This instruction is left for compatibility with SPARC64 VIIIfx. On SPARC64 VIIIfx, this instruction updated the entire cache line with an undefined value. On SPARC64™ X / SPARC64™ X+, it does not perform any memory or cache operations. However, exceptions related to memory accesses are detected.

XFILL for noncacheable space does not cause a *DAE_nc_page* exception.

Exception	Target instruction	Condition
<i>illegal_instruction</i>	op3 = 01 0111 ₂ (STTWA)	Odd-numbered destination register (rd)
<i>fp_disabled</i>	op3 = 11 0111 ₂ (STDFA)	PSTATE.PEF = 0 or FPRS.FEF = 0
<i>illegal_action</i>	op3 = 01 1110 ₂ (STXA) op3 = 01 0111 ₂ (STTWA)	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd ≠ 0
	op3 = 11 0111 ₂ (STDFA)	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urd<1> ≠ 0
<i>STDF_mem_address_not_aligned</i>	op3 = 11 0111 ₂ (STDFA)	<i>regaddr</i> aligned on 4-byte boundary but not 8-byte boundary
<i>mem_address_not_aligned</i>	op3 = 11 0111 ₂ (STDFA)	<i>regaddr</i> not aligned on 4-byte boundary
<i>VA_watchpoint</i>	All	When the watchpoint address matches any address in the cache line Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	All	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	All	Refer to 12.5.1.7

7.115. DES support instructions

Compatibility Note Future compatibility of DES support instructions is not guaranteed. These instructions should only be used in libraries for the SPARC64™ X or SPARC64™ X+ platform.

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FDESENCX	0 1001 1000 ₂	DES operation	✓	✓	fdesencx	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FDESPC1X	0 1001 1001 ₂	DES Permuted Choice 1	✓	✓	fdespc1x	<i>freg_{rs1}</i> , <i>freg_{rd}</i>
FDESIPX	0 1001 1010 ₂	DES Initial Permutation	✓	✓	fdesipx	<i>freg_{rs1}</i> , <i>freg_{rd}</i>
FDESIIPX	0 1001 1011 ₂	DES Inverse Initial Permutation	✓	✓	fdesiipx	<i>freg_{rs1}</i> , <i>freg_{rd}</i>
FDESKEYX	0 1001 1100 ₂	DES Key Calculation	✓	✓	fdeskeyx	<i>freg_{rs1}</i> , <i>index</i> , <i>freg_{rd}</i>

FDESENCX

10 ₂	rd	op3 = 11 0110 ₂	rs1	Opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

FDESPC1X, FDESIPX, FDESIIPX

10 ₂	rd	op3 = 11 0110 ₂	rs1	Opf	0 0000 ₂
31 30 29	25 24	19 18	14 13	5 4	0

FDESKEYX

10 ₂	rd	op3 = 11 0110 ₂	rs1	Opf	index
31 30 29	25 24	19 18	14 13	5 4	0

Description FDESENCX processes one round of the 16 rounds in the DES encoding and decoding algorithms. Given Fd[rs1] as the permuted input, we refer to the upper 32 bits as *L* and the lower 32 bits as *R*, Fd[rs2] is the key data, and the result is written in Fd[rd]. FDESENCX preforms the operation shown in the following expression.

$$\begin{aligned} \text{Fd[rd]} &= \text{Fd[rs1]} < 31:0 > :: (\text{Fd[rs1]} < 63:32 > \wedge f(\text{Fd[rs1]} < 31:0 >, \text{PC2}(\text{Fd[rs2]} < 55:0 >))) \\ &= R :: (L \wedge f(R, \text{PC2}(\text{KEY}))) \end{aligned}$$

Here, *f*0 is the encoding or decoding function, which outputs 32-bit data. PC20 is the Permuted Choice2 function, which outputs 48-bit data. Both functions are defined in the DES specification.

Note FDESENCX applies PC20 to Fd[rs2]. That is, the key data specified for Fd[rs2] is not the result *Kn* of the key schedule defined in the DES specification. Instead, Fd[rs2] is the input for PC20. This key data is calculated by the FDESKEYX instruction.

FDESPC1X operates on Fd[rs1] and writes the result in Fd[rd].

FDESPC1X executes PC10, which is the Permuted Choice 1 function defined by the DES specification. PC10 chooses 56 bits from Fd[rs1] and writes the result in the lower 56 bits of Fd[rd]. Parity bits for Fd[rs1] are generated and written in the upper 8 bits of Fd[rd].

$$\begin{aligned} \text{Fd[rd]} < 63 > &: \sim \text{Fd[rs1]} < 63:56 > \\ \text{Fd[rd]} < 62 > &: \sim \text{Fd[rs1]} < 55:48 > \\ \text{Fd[rd]} < 61 > &: \sim \text{Fd[rs1]} < 47:40 > \\ \text{Fd[rd]} < 60 > &: \sim \text{Fd[rs1]} < 39:32 > \\ \text{Fd[rd]} < 59 > &: \sim \text{Fd[rs1]} < 31:24 > \\ \text{Fd[rd]} < 58 > &: \sim \text{Fd[rs1]} < 23:16 > \end{aligned}$$

$Fd[rd]<57>: \sim Fd[rs1]<15:8>$

$Fd[rd]<56>: \sim Fd[rs1]<7:0>$

FDESIPX operates on $Fd[rs1]$ and writes the result in $Fd[rd]$.

FDESIPX performs the initial permutation, IP^0 defined by the DES specification. Table 7-23 shows how IP^0 permutes the input bits. The position of a bit in the output data $Fd[rd]<63:0>$ is a binary number $000000_2 - 111111_2$. We fix the upper or lower three bits of this binary number and let the other bits vary. That is, each row represents the positions in the output with the same upper three bits, and each column represents the positions with the same lower three bits. The intersection of a row and column shows the position of the bit in the input $Fd[rs1]<63:0>$ that is written to that position in the output.

For example, the first row ($111xxx_2$) shows the positions in the input data (bits 6, 14, ..., 62) corresponding to positions in the output data (bits 111111_2 , 111110_2 , ..., 111000_2). The next row ($110xxx_2$) shows the positions in the input data (bits 4, 12, ..., 60) corresponding to positions in the output data (bits 110111_2 , 110110_2 , ..., 110000_2). Specifically, the cell in the first row and the first column, whose upper bits are 111_2 and lower bits are 111_2 , has the value 6. That is, input bit $Fd[rs1]<6>$ is written to output bit $Fd[rd]<63>$.

Table 7-23 FDESIPX bit permutation

Bit position in $Fd[rd]$								
Upper three bits	Lower three bits							
	$xxx111_2$	$xxx110_2$	$xxx101_2$	$xxx100_2$	$xxx011_2$	$xxx010_2$	$xxx001_2$	$xxx000_2$
$111xxx_2$	6	14	22	30	38	46	54	62
$110xxx_2$	4	12	20	28	36	44	52	60
$101xxx_2$	2	10	18	26	34	42	50	58
$100xxx_2$	0	8	16	24	32	40	48	56
$011xxx_2$	7	15	23	31	39	47	55	63
$010xxx_2$	5	13	21	29	37	45	53	61
$001xxx_2$	3	11	19	27	35	43	51	59
$000xxx_2$	1	9	17	25	33	41	49	57

FDESIIPX operates on $Fd[rs1]$ and writes the result in $Fd[rd]$.

FDESIIPX executes IP^{-10} , which is the inverse function of the initial permutation, as defined by the DES specification. Table 7-24 shows how IP^{-10} permutes the input bits. The position of a bit in the output data $Fd[rd]<63:0>$ is a binary number $000000_2 - 111111_2$. We fix the upper or lower three bits and let the other bits vary. That is, each row represents the positions in the output with the same upper three bits, and each column represents the positions with the same lower three bits. The intersection of a row and column shows the positions of the bit in the input $Fd[rd]<63:0>$ that is written to that position in the output.

For example, the first row ($111xxx_2$) shows the positions in the input data (bits 56, 24, ..., 0) corresponding to the positions in the output data (bits 111111_2 , 111110_2 , ..., 111000_2). The next row ($110xxx_2$) shows the positions in the input data (bits 57, 25, ..., 1) corresponding to the positions in the output data (bits 110111_2 , 110110_2 , ..., 110000_2). Specifically, the cell in the first row and the first column, whose upper 3 bits are 111_2 and lower 3 bits are 111_2 , has the value 56. That is, input bit $Fd[rs1]<56>$ is written to output bit $Fd[rd]<63>$.

Table 7-24 FDESIIPX bit permutation

Bit position in Fd[rd]								
Upper three bits	Lower three bits							
	xxx111 ₂	xxx110 ₂	xxx101 ₂	xxx100 ₂	xxx011 ₂	xxx010 ₂	xxx001 ₂	xxx000 ₂
111xxx ₂	56	24	48	16	40	8	32	0
110xxx ₂	57	25	49	17	41	9	33	1
101xxx ₂	58	26	50	18	42	10	34	2
100xxx ₂	59	27	51	19	43	11	35	3
011xxx ₂	60	28	52	20	44	12	36	4
010xxx ₂	61	29	53	21	45	13	37	5
001xxx ₂	62	30	54	22	46	14	38	6
000xxx ₂	63	31	55	23	47	15	39	7

Note Table 7-24 differs from the table in the DES specification for the inverse function of the initial permutation because FDESIIPX is a composite operation that exchanges the upper and lower 32 bits in the pre-output and then applies $IP^{-1}()$.

FDESKEYX performs the operation specified by the index field on the input Fd[rs1] and writes the result in Fd[rd].

index	Operation
0	Fd[rs1]<63:56> :: ROTL(Fd[rs1]<55:28>, 1) :: ROTL(Fd[rs1]<27:0>, 1)
1	Fd[rs1]<63:56> :: ROTL(Fd[rs1]<55:28>, 2) :: ROTL(Fd[rs1]<27:0>, 2)
2 – 1F ₁₆	reserved

In the above table, ROTL(x, y) is a function that rotates x left by y bits.

The DES key schedule takes 56 bits from the 64-bit key, permutes these bits by PC10, and divides the result into two 28-bit data, *C* and *D*. Depending on the round, *C* and *D* are each rotated left by one or two bits and then merged. Merged data is permuted by PC20. FDESKEYX does part of this processing. The FDESKEYX instruction assumes *C* and *D* as inputs, rotates each data based on the index, and outputs the merged result. *C* and *D* are generated by the FDESPC1X instruction.

Usage example

```

/*
 * DES encryption
 *
 * Input
 *   %f0: Plaintext data
 *   %f2: Key data
 * Output
 *   %f0: Ciphertext data
 */

fdespc1x    %f2, %f2          ! IP(key)
fdeskeyx    %f2, 0, %f4       ! K1
fdeskeyx    %f4, 0, %f6       ! K2
fdeskeyx    %f6, 1, %f8       ! K3
fdeskeyx    %f8, 1, %f10      ! K4
fdeskeyx    %f10, 1, %f12     ! K5
fdeskeyx    %f12, 1, %f14     ! K6

```

```

fdeskeyx    %f14, 1, %f16    ! K7
fdeskeyx    %f16, 1, %f18    ! K8
fdeskeyx    %f18, 0, %f20    ! K9
fdeskeyx    %f20, 1, %f22    ! K10
fdeskeyx    %f22, 1, %f24    ! K11
fdeskeyx    %f24, 1, %f26    ! K12
fdeskeyx    %f26, 1, %f28    ! K13
fdeskeyx    %f28, 1, %f30    ! K14
fdeskeyx    %f30, 1, %f32    ! K15
fdeskeyx    %f32, 0, %f34    ! K16

fdesipx     %f0, %f0        ! IP(data)
fdesencx    %f0, %f4, %f0    ! round 1
fdesencx    %f0, %f6, %f0    ! round 2
fdesencx    %f0, %f8, %f0    ! round 3
fdesencx    %f0, %f10, %f0   ! round 4
fdesencx    %f0, %f12, %f0   ! round 5
fdesencx    %f0, %f14, %f0   ! round 6
fdesencx    %f0, %f16, %f0   ! round 7
fdesencx    %f0, %f18, %f0   ! round 8
fdesencx    %f0, %f20, %f0   ! round 9
fdesencx    %f0, %f22, %f0   ! round 10
fdesencx    %f0, %f24, %f0   ! round 11
fdesencx    %f0, %f26, %f0   ! round 12
fdesencx    %f0, %f28, %f0   ! round 13
fdesencx    %f0, %f30, %f0   ! round 14
fdesencx    %f0, %f32, %f0   ! round 15
fdesencx    %f0, %f34, %f0   ! round 16
fdesiipx    %f0, %f0        ! IP-1(data)

/*
 * DES decryption
 *
 * Input
 *   %f0: Ciphertext data
 *   %f2: Key data
 * Output
 *   %f0: Plaintext data
 */

fdespc1x    %f2, %f2        ! IP(key)
fdeskeyx    %f2, 0, %f4     ! K1
fdeskeyx    %f4, 0, %f6     ! K2
fdeskeyx    %f6, 1, %f8     ! K3
fdeskeyx    %f8, 1, %f10    ! K4
fdeskeyx    %f10, 1, %f12   ! K5
fdeskeyx    %f12, 1, %f14   ! K6
fdeskeyx    %f14, 1, %f16   ! K7
fdeskeyx    %f16, 1, %f18   ! K8
fdeskeyx    %f18, 0, %f20   ! K9
fdeskeyx    %f20, 1, %f22   ! K10
fdeskeyx    %f22, 1, %f24   ! K11
fdeskeyx    %f24, 1, %f26   ! K12
fdeskeyx    %f26, 1, %f28   ! K13
fdeskeyx    %f28, 1, %f30   ! K14
fdeskeyx    %f30, 1, %f32   ! K15
fdeskeyx    %f32, 0, %f34   ! K16

fdesipx     %f0, %f0        ! IP(data)
fdesencx    %f0, %f34, %f0   ! round 16
fdesencx    %f0, %f32, %f0   ! round 15
fdesencx    %f0, %f30, %f0   ! round 14
fdesencx    %f0, %f28, %f0   ! round 13
fdesencx    %f0, %f26, %f0   ! round 12
fdesencx    %f0, %f24, %f0   ! round 11
fdesencx    %f0, %f22, %f0   ! round 10
fdesencx    %f0, %f20, %f0   ! round 9

```

fdesencx	%f0, %f18, %f0	! round 8
fdesencx	%f0, %f16, %f0	! round 7
fdesencx	%f0, %f14, %f0	! round 6
fdesencx	%f0, %f12, %f0	! round 5
fdesencx	%f0, %f10, %f0	! round 4
fdesencx	%f0, %f8, %f0	! round 3
fdesencx	%f0, %f6, %f0	! round 2
fdesencx	%f0, %f4, %f0	! round 1
fdesiipx	%f0, %f0	! IP ⁻¹ (data)

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FDESPC1X, FDESIPX, FDESII PX	iw<4:0> ≠ 0
	FDESKEYX	index = 02 ₁₆ – 1F ₁₆
<i>fp_disabled</i>	All	PSTATE.PEF = 0 or FPRS.FEF = 0
<i>illegal_action</i>	FDESENCX	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	FDESPC1X, FDESIPX, FDESII PX, FDESKEYX	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.116. AES support instructions

Compatibility Note Future compatibility of AES support instructions is not guaranteed. These instructions should only be used in libraries for the SPARC64™ X or SPARC64™ X+ platform.

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FAESEN CX	0 1001 0000 ₂	AES encryption operation	✓	✓	faesencx	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FAESDECX	0 1001 0001 ₂	AES decryption operation	✓	✓	faesdecx	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FAESENCLX	0 1001 0010 ₂	AES final round of encryption	✓	✓	faesenclx	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FAESDECLX	0 1001 0011 ₂	AES the final round of decryption	✓	✓	faesdeclx	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FAESKEYX	0 1001 0100 ₂	AES key generation	✓	✓	faeskeyx	<i>freg_{rs1}</i> , <i>index</i> , <i>freg_{rd}</i>

FAESEN CX, FAESDECX, FAESENCLX, FAESDECLX

10 ₂	rd	op3 = 11 0110 ₂	rs1	Opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

FAESKEYX

10 ₂	rd	op3 = 11 0110 ₂	rs1	Opf	index
31 30 29	25 24	19 18	14 13	5 4	0

Description FAESEN CX executes the four functions defined for each round of the AES encryption algorithm.

$$Fd[rd] = \{MixColumns(ShiftRows(SubBytes(\{Fd[rd], Fd[rs1]\}))) <127:64> \wedge Fd[rs2]\}$$

Fd[rd] and Fd[rs1] are the 16-byte input data. Fd[rd] is the upper eight bytes, and Fd[rs1] is the lower eight bytes. Fd[rs2] is the upper eight bytes of the round key. First, SubBytes() operates on the input data. Second, ShiftRows() operates on the result. Third, MixColumns() operates on the result, and a 16-byte result is generated. Fourth, the upper eight bytes of this result are XORed with the round key in Fd[rs2]. This fourth step is called AddRoundKey() in the AES specification. This final 8-byte result is written in Fd[rd]. Note that Fd[rd] is used both as an input and as the output.

The operations performed by FAESEN CX are symmetric for the upper and lower eight bytes of the input data. By exchanging the input, FAESEN CX outputs a result which is the lower eight bytes for the round. That is, specify the lower eight bytes in Fd[rd] and the upper eight bytes in Fd[rs1]. Fd[rs2] is the lower eight bytes of the round key.

When using FAESEN CX as a SIMD instruction, any floating-point register Fd[0] – Fd[126], Fd[256] – Fd[382] can be specified for Fd[rs1]. However, only basic registers Fd[0] – Fd[126] can be specified for Fd[rs2] and Fd[rd]. When an extended register Fd[256] – Fd[382] is specified for Fd[rs1], the corresponding basic register in Fd[0] – Fd[126] is used for the extended SIMD operation.

FAESENCLX executes the three functions defined for the final round of the AES encryption algorithm.

$$Fd[rd] = \{ShiftRows(SubBytes(\{Fd[rd], Fd[rs1]\})) <127:64> \wedge Fd[rs2]\}$$

Fd[rd] and Fd[rs1] are the 16-byte input data. Fd[rd] is the upper eight bytes, and Fd[rs1] is the lower eight bytes. Fd[rs2] is the upper eight bytes of the round key. First, SubBytes() operates on the input data. Second, ShiftRows() operates on the result, and a 16-byte result is generated. Third, the upper eight bytes of the result are XORed with the round key in

Fd[rs2]. This third step is called AddRoundKey() in the AES specification. This final result is written in Fd[rd]. Note that Fd[rd] is used both as an input and as the output.

The operations performed by FAESENCLX are symmetric for the upper and lower eight bytes of the input data. By exchanging the input, FAESENCLX outputs a result which is the lower eight bytes for the final round. That is, specify the lower eight bytes in Fd[rd] and the upper eight bytes in Fd[rs1]. Fd[rs2] is the lower eight bytes of the round key.

When using FAESENCLX as a SIMD instruction, any floating-point register Fd[0] – Fd[126], Fd[256] – Fd[382] can be specified for Fd[rs1]. However, only basic registers Fd[0] – Fd[126] can be specified for Fd[rs2] and Fd[rd]. When an extended register Fd[256] – Fd[382] is specified for Fd[rs1], the corresponding basic register in Fd[0] – Fd[126] is used for the extended SIMD operation.

FAESDECX executes the four functions defined for each round of the AES decryption algorithm.

$$\text{Fd[rd]} = \{\text{InvMixColumns}((\text{InvSubBytes}(\text{InvShiftRows}(\{\text{Fd[rd]}, \text{Fd[rs1]}\}))) <127:64> \wedge \text{Fd[rs2]})\}$$

Fd[rd] and Fd[rs1] are the 16-byte input data. Fd[rd] is the upper eight bytes and Fd[rs1] is the lower eight bytes. Fd[rs2] is the upper eight bytes of the round key. First, InvShiftRows() operates on the input data. Second, InvSubBytes() operates on the result, and a 16-byte result is generated. Third, the upper eight bytes of this result are XORed with the round key in Fd[rs2]. This third step is called AddRoundKey() in the AES specification. Fourth, InvMixColumns() operates on the result. The final result is written in Fd[rd]. Note that Fd[rd] is used both as an input and as the output.

The operations performed by FAESDECX are symmetric for the upper and lower eight bytes of the input data. By exchanging the input, FAESDECX outputs a result which is the lower eight bytes for the round. That is, specify the lower eight bytes in Fd[rd] and the upper eight bytes in Fd[rs1]. Fd[rs2] is the lower eight bytes of the round key.

When using FAESDECX as a SIMD instruction, any floating-point register Fd[0] – Fd[126], Fd[256] – Fd[382] can be specified for Fd[rs1]. However, only basic registers Fd[0] – Fd[126] can be specified for Fd[rs2] and Fd[rd]. When an extended register Fd[256] – Fd[382] is specified for Fd[rs1], the corresponding basic register in Fd[0] – Fd[126] is used for the extended SIMD operation.

FAESDECLX executes the three functions defined for the final round of the AES decryption algorithm.

$$\text{Fd[rd]} = \{(\text{InvSubBytes}(\text{InvShiftRows}(\{\text{Fd[rd]}, \text{Fd[rs1]}\}))) <127:64> \wedge \text{Fd[rs2]}\}$$

Fd[rd] and Fd[rs1] are the 16-byte input data. Fd[rd] is the upper eight bytes, and Fd[rs1] is the lower eight bytes. Fd[rs2] is the upper eight bytes of the round key. First, InvShiftRows() operates on the input data. Second, InvSubBytes() operates on the results and a 16-byte result is generated. Third, the upper eight bytes of the result are XORed with the round key in Fd[rs2]. This third step is called AddRoundKey() in the AES specification. This final result is written in Fd[rd]. Note that Fd[rd] is used both as an input and an output.

The operations performed by FAESDECLX are symmetric for the upper and lower eight bytes of the input data. By exchanging the input, FAESDECLX outputs a result which is the lower eight bytes for the final round. That is, specify the lower eight bytes in Fd[rd] and the upper eight bytes in Fd[rs1]. Fd[rs2] is the lower eight bytes of the round key.

When using FAESDECLX as a SIMD instruction, any floating-point register Fd[0] – Fd[126], Fd[256] – Fd[382] can be specified for Fd[rs1]. However, only basic registers Fd[0] – Fd[126] can be specified for Fd[rs2] and Fd[rd]. When an extended register Fd[256] – Fd[382] is specified for Fd[rs1], the corresponding basic register in Fd[0] – Fd[126] is used for the extended SIMD operation.

FAESKEYX generates the round key. To calculate all 4-byte data $W[i]$, which are the round keys used in each round, two 4-byte inputs $W[i-1]$ and $W[i-N_k]$ are needed from the previous rounds where N_k is the number of 4-byte words comprising the key data. FAESKEYX calculates both $W[i]$ and $W[i+1]$ at the same time. Specify $W[i-2]$ in the upper four bytes of $Fd[rs1]$ and $W[i-1]$ in the lower four bytes. Specify $W[i-N_k]$ in the upper four bytes of $Fd[rd]$ and $W[i-N_k+1]$ in the lower 4 bytes.. FAESKEYX performs the operation specified by the index field and the 32-bit result is output to temp.

index	Operation
00 ₁₆	temp = $Fd[rd]<63:32> \wedge Fd[rs1]<31:0>$
01 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(Fd[rs1]<31:0>)$
10 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 0100\ 0000_{16}$
11 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 0200\ 0000_{16}$
12 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 0400\ 0000_{16}$
13 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 0800\ 0000_{16}$
14 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 1000\ 0000_{16}$
15 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 2000\ 0000_{16}$
16 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 4000\ 0000_{16}$
17 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 8000\ 0000_{16}$
18 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 1B00\ 0000_{16}$
19 ₁₆	temp = $Fd[rd]<63:32> \wedge \text{SubWord}(\text{RotWord}(Fd[rs1]<31:0>)) \wedge 3600\ 0000_{16}$

After this calculation, temp is equivalent to $W[i]$. $W[i+1]$ is calculated as "temp **xor** $W[i-N_k+1]$ ", which is equivalent to "temp **xor** $Fd[rd]<31:0>$ ". $W[i]$ is written in the upper four bytes of $Fd[rd]$ and $W[i+1]$ is written in the lower four bytes.

Usage example, non SIMD

```

/*
 * AES-128 key generation
 *
 * Input:
 *   %f0, %f2                : Key data
 * Output:
 *   %f0, %f2 ... %f40, %f42: Round keys
 */

fmovd    %f0, %f40          ! W[0],W[1]
fmovd    %f2, %f42          ! W[2],W[3]
faeskeyx %f42, 0x10, %f40
fmovd    %f40, %f4          ! W[4],W[5]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f6          ! W[6],W[7]
faeskeyx %f42, 0x11, %f40
fmovd    %f40, %f8          ! W[8],W[9]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f10         ! W[10],W[11]
faeskeyx %f42, 0x12, %f40
fmovd    %f40, %f12         ! W[12],W[13]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f14         ! W[14],W[15]
faeskeyx %f42, 0x13, %f40
fmovd    %f40, %f16         ! W[16],W[17]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f18         ! W[18],W[19]
faeskeyx %f42, 0x14, %f40
fmovd    %f40, %f20         ! W[20],W[21]
faeskeyx %f40, 0x0, %f42

```

```

fmovd    %f42, %f22      ! W[22],W[23]
faeskeyx %f42, 0x15, %f40
fmovd    %f40, %f24      ! W[24],W[25]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f26      ! W[26],W[27]
faeskeyx %f42, 0x16, %f40
fmovd    %f40, %f28      ! W[28],W[29]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f30      ! W[30],W[31]
faeskeyx %f42, 0x17, %f40
fmovd    %f40, %f32      ! W[32],W[33]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f34      ! W[34],W[35]
faeskeyx %f42, 0x18, %f40
fmovd    %f40, %f36      ! W[36],W[37]
faeskeyx %f40, 0x0, %f42
fmovd    %f42, %f38      ! W[38],W[39]
faeskeyx %f42, 0x19, %f40 ! W[40],W[41]
faeskeyx %f40, 0x0, %f42 ! W[42],W[43]

/*
 * AES-128 ECB mode encryption
 *
 * Input:
 *   %f0, %f2 ... %f40, %f42: Round keys
 *   %f50, %f52             : Plaintext data
 * Output:
 *   %f60, %f62             : Ciphertext data
 */

fxor     %f50, %f0, %f60 ! Round 0
fxor     %f52, %f2, %f62

fmovd    %f60, %f58      ! Round 1
faesencx %f62, %f4, %f60
faesencx %f58, %f6, %f62
fmovd    %f60, %f58      ! Round 2
faesencx %f62, %f8, %f60
faesencx %f58, %f10, %f62
fmovd    %f60, %f58      ! Round 3
faesencx %f62, %f12, %f60
faesencx %f58, %f14, %f62
fmovd    %f60, %f58      ! Round 4
faesencx %f62, %f16, %f60
faesencx %f58, %f18, %f62
fmovd    %f60, %f58      ! Round 5
faesencx %f62, %f20, %f60
faesencx %f58, %f22, %f62
fmovd    %f60, %f58      ! Round 6
faesencx %f62, %f24, %f60
faesencx %f58, %f26, %f62
fmovd    %f60, %f58      ! Round 7
faesencx %f62, %f28, %f60
faesencx %f58, %f30, %f62
fmovd    %f60, %f58      ! Round 8
faesencx %f62, %f32, %f60
faesencx %f58, %f34, %f62
fmovd    %f60, %f58      ! Round 9
faesencx %f62, %f36, %f60
faesencx %f58, %f38, %f62

fmovd    %f60, %f58      ! Round 10 final round
faesenclx %f62, %f40, %f60
faesenclx %f58, %f42, %f62

/*

```

```

* AES-128 ECB mode decryption
*
* Input:
*   %f0, %f2 ... %f40, %f42: Round keys
*   %f50, %f52             : Ciphertext data
* Output:
*   %f60, %f62             : Plaintext data
*/

fxor    %f50, %f40, %f60    ! Round 0
fxor    %f52, %f42, %f62

fmovd   %f60, %f58         ! Round 1
faesdecx %f62, %f36, %f60
faesdecx %f58, %f38, %f62
fmovd   %f60, %f58         ! Round 2
faesdecx %f62, %f32, %f60
faesdecx %f58, %f34, %f62
fmovd   %f60, %f58         ! Round 3
faesdecx %f62, %f28, %f60
faesdecx %f58, %f30, %f62
fmovd   %f60, %f58         ! Round 4
faesdecx %f62, %f24, %f60
faesdecx %f58, %f26, %f62
fmovd   %f60, %f58         ! Round 5
faesdecx %f62, %f20, %f60
faesdecx %f58, %f22, %f62
fmovd   %f60, %f58         ! Round 6
faesdecx %f62, %f16, %f60
faesdecx %f58, %f18, %f62
fmovd   %f60, %f58         ! Round 7
faesdecx %f62, %f12, %f60
faesdecx %f58, %f14, %f62
fmovd   %f60, %f58         ! Round 8
faesdecx %f62, %f8, %f60
faesdecx %f58, %f10, %f62
fmovd   %f60, %f58         ! Round 9
faesdecx %f62, %f4, %f60
faesdecx %f58, %f6, %f62

fmovd   %f60, %f58         ! Round 10 final round
faesdec1x %f62, %f0, %f60
faesdec1x %f58, %f2, %f62

```

Usage example SIMD

```

/*
* AES-128 key generation
* Input:
*   %f2, %f258             : Key data
* Output:
*   %f2, %f258, %f4, %f260 ... %f22, %f278 : Round keys
*/

fmovd   %f2, %f32          ! W[0],W[1]
sxar1
fmovd   %f258, %f34         ! W[2],W[3]
faeskeyx %f34, 0x10, %f32
faeskeyx %f32, 0x0, %f34
fmovd   %f32, %f4          ! W[4],W[5]
sxar1
fmovd   %f34, %f260         ! W[6],W[7]
faeskeyx %f34, 0x11, %f32
faeskeyx %f32, 0x0, %f34
fmovd   %f32, %f6          ! W[8],W[9]

```

```

sxar1
fmovd    %f34, %f262      ! W[10],W[11]
faeskeyx %f34, 0x12, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f8        ! W[12],W[13]
sxar1
fmovd    %f34, %f264      ! W[14],W[15]
faeskeyx %f34, 0x13, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f10       ! W[16],W[17]
sxar1
fmovd    %f34, %f266      ! W[18],W[19]
faeskeyx %f34, 0x14, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f12       ! W[20],W[21]
sxar1
fmovd    %f34, %f268      ! W[22],W[23]
faeskeyx %f34, 0x15, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f14       ! W[24],W[25]
sxar1
fmovd    %f34, %f270      ! W[26],W[27]
faeskeyx %f34, 0x16, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f16       ! W[28],W[29]
sxar1
fmovd    %f34, %f272      ! W[30],W[31]
faeskeyx %f34, 0x17, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f18       ! W[32],W[33]
sxar1
fmovd    %f34, %f274      ! W[34],W[35]
faeskeyx %f34, 0x18, %f32
faeskeyx %f32, 0x0, %f34
fmovd    %f32, %f20       ! W[36],W[37]
sxar1
fmovd    %f34, %f276      ! W[38],W[39]
faeskeyx %f34, 0x19, %f22 ! W[40],W[41]
sxar1
faeskeyx %f22, 0x0, %f278 ! W[42],W[43]

/*
 * AES-128 ECB mode encryption (SIMD use)
 *
 * Input:
 *   %f2, %f258, %f4, %f260 ... %f22, %f278 : Round keys
 *   %f0, %f256                               : Plaintext data
 * Output:
 *   %f0, %f256                               : Ciphertext data
 */

sxar2
fxor,s    %f0, %f2, %f0    ! Round 0
faesencx,s %f256, %f4, %f0 ! Round 1
sxar2
faesencx,s %f256, %f6, %f0 ! Round 2
faesencx,s %f256, %f8, %f0 ! Round 3
sxar2
faesencx,s %f256, %f10, %f0 ! Round 4
faesencx,s %f256, %f12, %f0 ! Round 5
sxar2
faesencx,s %f256, %f14, %f0 ! Round 6
faesencx,s %f256, %f16, %f0 ! Round 7
sxar2
faesencx,s %f256, %f18, %f0 ! Round 8
faesencx,s %f256, %f20, %f0 ! Round 9

```

```

sxa1
faesenc1x,s  %f256, %f22, %f0    ! Round 10 final round

/*
 * AES-128 ECB mode decryption (SIMD use)
 *
 * Input:
 *   %f2, %f258, %f4, %f260 ... %f22, %f278    : Round keys
 *   %f0, %f256                                : Ciphertext data
 * Output:
 *   %f0, %f256                                : Plaintext data
 */

sxa2
fxor,s      %f0, %f22, %f0        ! Round 0
faesdecx,s  %f256, %f20, %f0      ! Round 1
sxa2
faesdecx,s  %f256, %f18, %f0      ! Round 2
faesdecx,s  %f256, %f16, %f0      ! Round 3
sxa2
faesdecx,s  %f256, %f14, %f0      ! Round 4
faesdecx,s  %f256, %f12, %f0      ! Round 5
sxa2
faesdecx,s  %f256, %f10, %f0      ! Round 6
faesdecx,s  %f256, %f8, %f0       ! Round 7
sxa2
faesdecx,s  %f256, %f6, %f0       ! Round 8
faesdecx,s  %f256, %f4, %f0       ! Round 9
sxa1
faesdec1x,s %f256, %f2, %f0       ! Round 10 final round

```

Exception	Target instruction	Condition
<i>illegal_instruction</i>	FAESKEYX	index = 02 ₁₆ – 0F ₁₆ , 1A ₁₆ – 1F ₁₆
<i>fp_disabled</i>	All	PSTATE.PEF = 0 or FPRS.FEF = 0
<i>illegal_action</i>	FAESKEYX	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	FAEENCX, FAESDECX, FAEENCLX, FAESDECLX	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.117. Decimal Floating-Point Operations

Instruction	Opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FADDtd	0 1010 0000 ₂	Add decimal floating point	rd : basic only		faddtd <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSUBtd	0 1010 0001 ₂	Subtract decimal floating point	rd : basic only		fsubtd <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMULtd	0 1010 0010 ₂	Multiply decimal floating point	rd : basic only		fmultd <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FDIVtd	0 1010 0011 ₂	Divide decimal floating point	rd : basic only		fdivtd <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FQUAtd	0 1010 0110 ₂	Significant digit conversion of decimal floating point	rd : basic only		fquatl <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd			op3 = 11 0110 ₂	rs1			opf			rs2		
31 30 29	25 24			19 18	14 13			5 4			0		

Description FADDtd adds two double precision decimal floating point numbers, Fd[rs1] and Fd[rs2]. The result is written in Fd[rd]. When the operation result is exact, the preferred exponent is the smaller exponent of Fd[rs1] and Fd[rs2]. When the result is inexact, the preferred exponent is the closest value to the smaller exponent of Fd[rs1] and Fd[rs2] within the range and the precision of the output format. When the result is exactly zero, the sign of the result is negative if the rounding mode is 3 (towards $-\infty$). Otherwise, the sign is positive.

Table 7-25 FADDtd

		Fd[rs2]							
		−∞	−N	−0	+0	+N	+∞	QNaN	SNaN
Fd[rs1]	−∞	— −∞					NV dNaN	— QNaN2	NV QNaN2
	−N	— Fd[rs1] + Fd[rs2] ^{xvi}		— Fd[rs1]		— Fd[rs1] + Fd[rs2] ^{xvi}			
	−0	— Fd[rs2]	— −0	— +0 ^{xvii}	— Fd[rs2]				
	+0		— +0 ^{xvii}	— +0					
	+N	— Fd[rs1] + Fd[rs2] ^{xvi}		— Fd[rs1]		— Fd[rs1] + Fd[rs2] ^{xvi}			
	+∞	NV dNaN	— +∞						
	QNaN	— QNaN1							
	SNaN	NV QNaN1							

FSUBtd subtracts the double-precision decimal floating-point number in Fd[rs2] from the double-precision decimal floating-point number in Fd[rs1]. The result is written in Fd[rd]. When the result is exact, the preferred exponent is the smaller exponent of Fd[rs1] and Fd[rs2]. When the result is inexact, the preferred exponent is the closest value to the smaller exponent of Fd[rs1] and Fd[rs2] within the range and the precision of the output

^{xvi} When the result is 0, footnote (xvii) applies.

^{xvii} When the rounding mode is towards $-\infty$, the result is -0 .

format. When the operation result is exactly zero, the sign of the result is negative if the rounding mode is 3 (towards $-\infty$). Otherwise, the sign is positive.

Table 7-26 FSUBtd

		Fd[rs2]									
		$-\infty$	$-N$	-0	$+0$	$+N$	$+\infty$	QNaN	SNaN		
Fd[rs1]	$-\infty$	NV dNaN						$-\infty$	QNaN2	NV QNaN2	
	$-N$		--- Fd[rs1] – Fd[rs2] ^{xviii}	--- Fd[rs1]		--- Fd[rs1] – Fd[rs2] ^{xviii}					
	-0		--- –Fd[rs2]	--- $+0^{\text{xix}}$	--- -0	--- –Fd[rs2]					
	$+0$			--- $+0$	--- $+0^{\text{xix}}$						
	$+N$		--- Fd[rs1] – Fd[rs2] ^{xviii}	--- Fd[rs1]		--- Fd[rs1] – Fd[rs2] ^{xviii}					
	$+\infty$	--- $+\infty$						NV dNaN			
	QNaN	--- QNaN1									
	SNaN	NV QNaN1									

FMULTd multiplies two double-precision decimal floating-point numbers Fd[rs1] and Fd[rs2]. The result is written in Fd[rd]. When the result is exact, the preferred exponent is the sum of the exponents of Fd[rs1] and Fd[rs2]. When the operation result is inexact, the preferred exponent is the closest value to the sum of the exponents of Fd[rs1] and Fd[rs2] within the range and the precision of the output format.

Table 7-27 FMULTd

		Fd[rs2]							QNaN	SNaN
		$-\infty$	$-N$	-0	$+0$	$+N$	$+\infty$			
Fd[rs1]	$-\infty$	--- $+\infty$			NV dNaN			--- $-\infty$	--- QNaN2	NV QNaN2
	$-N$	--- Fd[rs1] $\times$ Fd[rs2]				--- Fd[rs1] $\times$ Fd[rs2]				
	-0	NV dNaN	--- $+0$		--- -0			NV dNaN		
	$+0$		--- -0		--- $+0$					
	$+N$		--- Fd[rs1] $\times$ Fd[rs2]			--- Fd[rs1] $\times$ Fd[rs2]				
	$+\infty$	--- $-\infty$			NV dNaN			--- $+\infty$		
	QNaN	--- QNaN1								
	SNaN	NV QNaN1								

^{xviii} When the result is 0, the footnote (xix) applies..

^{xix} When the rounding mode is towards $-\infty$, the result is -0 .

FDIVtd divides the double-precision decimal floating-point number Fd[rs1] by the double-precision decimal floating-point number Fd[rs2]. The result is written in Fd[rd]. When the result is exact, the preferred exponent is the difference of the exponents of Fd[rs1] and Fd[rs2]. When the result is inexact, the preferred exponent is the closest value to the difference of the exponents of Fd[rs1] and Fd[rs2] within the range and the precision of the output format. If the result of the operation is not within the range of the preferred exponent, the quotient is calculated until the remainder is zero or the maximum number of significant figures is reached. The exponent is then adjusted to be consistent with the quotient.

When the result is 0, the exponent is the preferred exponent and the significand is all zeros. However, if the result is 0 and the divisor is ∞ , the exponent is all zeros as well.

Table 7-28 FDIVtd

		Fd[rs2]							
		$-\infty$	$-N$	-0	$+0$	$+N$	$+\infty$	QNaN	SNaN
Fd[rs1]	$-\infty$	NV dNaN	--- $+\infty$		--- $-\infty$		NV dNaN	--- QNaN2	NV QNaN2
	$-N$	--- $+0^{\text{xx}}$	--- Fd[rs1] / Fd[rs2]	DZ $+\infty$	DZ $-\infty$	--- Fd[rs1] / Fd[rs2]	--- -0^{xx}		
	-0		--- $+0^{\text{xxi}}$	NV dNaN		--- -0^{xxi}			
	$+0$	--- -0^{xxi}				--- $+0^{\text{xxi}}$			
	$+N$	--- -0^{xx}	--- Fd[rs1] / Fd[rs2]	DZ $-\infty$	DZ $+\infty$	--- Fd[rs1] / Fd[rs2]	--- $+0^{\text{xx}}$		
	$+\infty$	NV dNaN	--- $-\infty$		--- $+\infty$		NV dNaN		
	QNaN	--- QNaN1							
	SNaN	NV QNaN1							

FQUAtd converts the double-precision decimal floating-point number in Fd[rs1] into a cohort member whose exponent is the same as the double-precision decimal floating-point number in Fd[rs2]. The result is written in Fd[rd]. The preferred exponent is the exponent of Fd[rd], regardless of whether the result is exact or inexact.

On overflow, the output is QNaN with an NV exception. That is, an NV exception is generated when the result cannot be expressed using the specified exponent.

When the significand is 0 due to underflow or rounding, the result remains 0 to be consistent with the preferred exponent.

^{xx} Significand is 0. Exponent after bias is 0.

^{xxi} Exponent is the preferred exponent.

Table 7-29 FQUAtd

		Fd[rs2]							
		−∞	−Fn	−0	+0	+Fn	+∞	QNaN	SNaN
Fd[rs1]	−∞	— −∞	NV dNaN				— −∞	— QNaN2	NV QNaN2
	−Fn	NV dNaN	— Q(Fd[rs1] : Fd[rs2])				NV dNaN		
	−0		— −0 ^{xxii}						
	+0		— +0 ^{xxii}						
	+Fn		— Q(Fd[rs1] : Fd[rs2])						
	+∞	— +∞	NV dNaN				— +∞		
	QNaN	— QNaN1							
	SNaN	NV QNaN1							

Instructions that operate on single-precision and quadruple-precision data are not defined.

The result is rounded as specified by GSR.dirnd when GSR.dim = 1, or FSR.drd when GSR.dim = 0.

The following table summarizes the preferred exponents for the decimal floating-point instructions.

Table 7-30 Preferred exponents

Instruction	Result is exact	Result is inexact
FADDtd	Smaller of Fd[rs1] and Fd[rs2] exponents	Closest value to preferred exponent for an exact result
FSUBtd	Smaller of Fd[rs1] and Fd[rs2] exponents	Closest value to preferred exponent for an exact result
FMULtd	Sum of Fd[rs1] and Fd[rs2] exponents	Closest value to preferred exponent for an exact result
FDIVtd	Difference of Fd[rs1] and Fd[rs2] exponents	Closest value to preferred exponent for an exact result If result is not within the range of the preferred exponent, quotient is calculated until remainder is zero or maximum significant figures reached. Exponent adjusted to be consistent with quotient.
FQUAtd	Exponent of Fd[rs2]	Exponent of Fd[rs2]

^{xxii} Significand is 0. Exponent is exponent of rs2.

Exception		Target instruction	Condition
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<2:1> ≠ 0
<i>fp_exception_ieee_754</i>	OF, UF	FADDtd, FSUBtd, FMULtd, FDIVtd	Conforms to IEEE754-2008
	NX, NV	All	Conforms to IEEE754-2008
	DZ	FDIVtd	Refer to table in description

7.118. Oracle Floating-Point Operations

Compatibility Note Future compatibility of Oracle floating-point operations is not guaranteed. These instructions should only be used in libraries for the SPARC64™ X / SPARC64™ X+ platforms.

Instruction	Opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FADDod	0 1011 0000 ₂	Add Oracle floating point	rd : basic only		faddod	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FSUBod	0 1011 0001 ₂	Subtract Oracle floating point	rd : basic only		fsubod	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FMULod	0 1011 0010 ₂	Multiply Oracle floating point	rd : basic only		fmulod	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FDIVod	0 1011 0011 ₂	Divide Oracle floating point	rd : basic only		fdivod	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FXADDodL O	0 1011 1000 ₂	Exact add Oracle floating point (lower)	rd : basic only		fxaddodlo	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FXADDodH I	0 1011 1001 ₂	Exact add Oracle floating point (upper)	rd : basic only		fxaddodhi	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FXMULodL O	0 1011 1010 ₂	Exact multiply Oracle floating point (lower)	rd : basic only		fxmulodlo	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FQUAod	0 1011 0110 ₂	Significant digit conversion of Oracle floating point	rd : basic only		fquaod	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FRQUAod	0 1011 0111 ₂	Extended exponent conversion of Oracle floating point	rd : basic only		Frquaod	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0110 ₂	rs1	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description FADDod adds the Oracle floating-point numbers Fd[rs1] and Fd[rs2] and writes the result in Fd[rd]. The result is normalized.

Table 7-31 FADDod

		Fd[rs2]				
		$-\infty$	$-N$	0	$+N$	$+\infty$
Fd[rs1]	$-\infty$	$-\infty$				NV dNaN
	$-N$		$\frac{\text{Fd[rs1]} + \text{Fd[rs2]}^{xxiii}}{\text{Fd[rs1]}}$	$\frac{\text{Fd[rs1]}}{\text{Fd[rs1]} + \text{Fd[rs2]}^{xxiii}}$		
	0		$\frac{\text{Fd[rs2]}}{0^{xxiv}}$	$\frac{\text{Fd[rs2]}}{\text{Fd[rs2]}}$		
	$+N$		$\frac{\text{Fd[rs1]} + \text{Fd[rs2]}^{xxiii}}{\text{Fd[rs1]}}$	$\frac{\text{Fd[rs1]}}{\text{Fd[rs1]} + \text{Fd[rs2]}^{xxiii}}$		
	$+\infty$	NV dNaN				$-\infty$ $+\infty$

^{xxiii} When the result is 0, footnote (xxiv) applies.

^{xxiv} When the rounding mode is towards −∞, the result is −0.

FSUBod subtracts the Oracle floating-point number Fd[rs2] from the Oracle floating-point number Fd[rs1] and writes the result in Fd[rd]. The result is normalized.

Table 7-32 FSUBod

		Fd[rs2]				
		$-\infty$	$-N$	0	$+N$	$+\infty$
Fd[rs1]	$-\infty$	NV dNaN				$-\infty$
	$-N$		--- Fd[rs1] – Fd[rs2] ^{xxv}	--- Fd[rs1]	--- Fd[rs1] – Fd[rs2] ^{xxv}	
	0		--- – Fd[rs2]	--- 0 ^{xxvi}	--- – Fd[rs2]	
	$+N$		--- Fd[rs1] – Fd[rs2] ^{xxv}	--- Fd[rs1]	--- Fd[rs1] – Fd[rs2] ^{xxv}	
	$+\infty$		--- $+\infty$	NV dNaN		

FMULod multiplies the Oracle floating-point numbers Fd[rs1] and Fd[rs2] and writes the result in Fd[rd]. The result is normalized.

Table 7-33 FMULod

		Fd[rs2]				
		$-\infty$	$-N$	0	$+N$	$+\infty$
Fd[rs1]	$-\infty$	--- $+\infty$		NV dNaN		--- $-\infty$
	$-N$		--- $\text{Fd[rs1]} \times \text{Fd[rs2]}$		--- $\text{Fd[rs1]} \times \text{Fd[rs2]}$	
	0	NV dNaN		--- 0		NV dNaN
	$+N$		--- $\text{Fd[rs1]} \times \text{Fd[rs2]}$		--- $\text{Fd[rs1]} \times \text{Fd[rs2]}$	
	$+\infty$	--- $-\infty$		NV dNaN		--- $+\infty$

FDIVod divides the Oracle floating-point number Fd[rs1] by the Oracle floating-point number Fd[rs2] and writes the result in Fd[rd]. The result is normalized.

FDIVod generates a DZ exception if the divisor Fd[rs2] is 0. However, even if the divisor is 0, a DZ exception is not generated if the dividend Fd[rs1] is $+\infty$, $-\infty$ or 0. If the divisor is 0 and the dividend is $+\infty$ or $-\infty$, the result is $+\infty$ and $-\infty$, respectively. If the divisor is 0 and the dividend is 0, the result is dNaN with an NV exception.

^{xxv} When the result is 0, footnote (xxvi) applies.

^{xxvi} When the rounding mode is towards $-\infty$, the result is -0 .

Table 7-34 FDIvod

		Fd[rs2]				
		$-\infty$	$-N$	0	$+N$	$+\infty$
Fd[rs1]	$-\infty$	NV dNaN	--- $+\infty$	--- $-\infty$	--- $-\infty$	NV dNaN
	$-N$	0	--- Fd[rs1] / Fd[rs2]	DZ $-\infty$	--- Fd[rs1] / Fd[rs2]	0
	0		--- Fd[rs1] / Fd[rs2]	NV dNaN		
	$+N$		--- Fd[rs1] / Fd[rs2]	DZ $+\infty$	--- Fd[rs1] / Fd[rs2]	
	$+\infty$		NV dNaN	--- $-\infty$	--- $+\infty$	

FXADDodLO and FXADDodHI are used to calculate the exact sum of two Oracle floating-point numbers. Note that the FADDod instruction outputs a rounded result and generates an NX exception when the sum cannot be expressed precisely as an Oracle floating-point number. FXADDodHI and FXADDodLO, however, output the exact sum of two Oracle floating-point numbers. These instructions can be used to add and subtract with arbitrary precision.

FXADDodLO behaves differently depending on the difference of the exponents of the two Oracle floating-point numbers Fd[rs1] and Fd[rs2]. If the difference is less than or equal to seven, Fd[rs1] and Fd[rs2] are added and the lower digits of the result are written in Fd[rd]. If the difference is greater than seven, the number with the smaller exponent is selected and written in Fd[rd]. If either Fd[rs1] or Fd[rs2] is the special value 0, the special value 0 is written in Fd[rd], regardless of the difference of the exponents.

FXADDodHI behaves differently depending on the difference of the exponents of the two Oracle floating-point numbers Fd[rs2] and Fd[rs1]. If the difference is less than or equal to seven, Fd[rs1] and Fd[rs2] are added and the upper digits of the result are written in Fd[rd]. If the difference is greater than seven, the number with the larger exponent is selected and written in Fd[rd]. If either Fd[rs1] or Fd[rs2] is the special value 0, the other number which is not 0 is written in Fd[rd], regardless of the difference of the exponents. If both Fd[rs1] and Fd[rs2] are the special value 0, the special value 0 is written in Fd[rd].

The results of FXADDodLO and FXADDodHI are normalized.

For the FXADDodLO and FXADDodHI instructions, the rounding mode is set to 1 (towards 0). Results are rounded regardless of the settings in FSR and GSR.

Note To precisely express the sum of two values with opposite signs, the number of digits required is equal to the difference of the exponents.

Example: $1 \times 10^{10} + (-1 \times 10^0) = 9999999999 \times 10^0$

When adding two numbers of opposite signs where the difference of their exponents is large, the result exceeds the number of digits that can be saved in two registers. FXADDodLO and FXADDodHI preserve precision by saving the inputs when the difference of the exponents is greater than seven.

Table 7-35 FXADDodLO

		Fd[rs2]				
		$-\infty$	$-N$	0	$+N$	$+\infty$
Fd[rs1]	$-\infty$	$-\infty$				NV dNaN
	$-N$		$\text{ADDLO}(\text{Fd}[\text{rs1}], \text{Fd}[\text{rs2}])$		$\text{ADDLO}(\text{Fd}[\text{rs1}], \text{Fd}[\text{rs2}])$	
	0		0			
	$+N$		$\text{ADDLO}(\text{Fd}[\text{rs1}], \text{Fd}[\text{rs2}])$		$\text{ADDLO}(\text{Fd}[\text{rs1}], \text{Fd}[\text{rs2}])$	
	$+\infty$	NV dNaN				

Table 7-36 FXADDodHI

		Fd[rs2]				
		−∞	−N	0	+N	+∞
Fd[rs1]	−∞	— −∞				NV dNaN
	−N		— ADDHI(Fd[rs1], Fd[rs2])	— Fd[rs1]	— ADDHI(Fd[rs1], Fd[rs2])	
	0		— Fd[rs2]	— 0	— Fd[rs2]	
	+N		— ADDHI(Fd[rs1], Fd[rs2])	— Fd[rs1]	— ADDHI(Fd[rs1], Fd[rs2])	
	+∞	NV dNaN				

FXMULodLO is used with FMULod to calculate the exact product of two Oracle floating-point numbers. Note that the FMULod instruction outputs a rounded result and generates an NX exception when the product cannot be expressed precisely as an Oracle floating-point number. FXMULodLO outputs the lower part of the product that is rounded by FMULod. The FXMULodLO instruction can be used to multiply with arbitrary precision.

FXMULodLO multiplies two Oracle floating-point numbers Fd[rs2] and Fd[rs1] and writes the lower part of the result in Fd[rd].

For the FXMULodLO instruction, the rounding mode is set to 1 (towards 0). Results are rounded regardless of the settings in FSR and GSR.

Programming Note Add the result of FMULod with rounding mode set to 1 (towards 0) and the result of FXMULodLO to calculate the exact product.

Table 7-37 FXMULodLO

		Fd[rs2]				
		$-\infty$	$-N$	0	$+N$	$+\infty$
Fd[rs1]	$-\infty$	$---$ $+\infty$			NV dNaN	$---$ $-\infty$
	$-N$	$---$ LO(Fd[rs1] $\times$ Fd[rs2])			$---$ LO(Fd[rs1] $\times$ Fd[rs2])	
	0	NV dNaN	$---$ 0			NV dNaN
	$+N$	$---$ LO(Fd[rs1] $\times$ Fd[rs2])			$---$ LO(Fd[rs1] $\times$ Fd[rs2])	
	$+\infty$	$---$ $-\infty$			NV dNaN	$---$ $+\infty$

Table 7-38 Multiplying rs1 = 99.99999999 $\times$ 100¹⁰ and rs2 = 99.99999999 $\times$ 100¹⁰

FMULod	FXMULodLO
99.99999998 $\times$ 100 ²¹	00.0001 $\times$ 100 ¹⁴

Table 7-39 Output of FXMULodLO if lower bits of result is 0

	Sign	Exponent part	Significand
Either of the inputs is 0	1	0	0x0000000000000000
Neither of the inputs is 0	Same as multiplication result	Exponent of multiplication result minus 7 (Exponent ≤ -59 : exponent $-7+128$) (Exponent $\geq +70$: exponent $-7-128$)	Sign is positive: 0x0101010101010101 Sign is negative 0x6565656565656565

FQUAod converts the Oracle floating-point number in Fd[rs1] into a cohort member whose exponent is the same as the Oracle floating-point number in Fd[rs2]. The result is written in Fd[rd]. The result is not normalized.

On overflow, the output is dNaN with an NV exception. That is, an NV exception is generated when the result cannot be expressed using the specified exponent.

When the significand is 0 due to underflow or rounding, the result remains 0. This 0 is not the special value 0. The sign is the same as Fd[rs1], the exponent is the same as Fd[rs2], and 0 corresponding to the sign is written in the significand.

The result of FQUAod is rounded as specified by GSR.dimrd when GSR.dim = 1, and FSR.drd when GSR.dim = 0.

Table 7-40 FQUAod

		Fd[rs2]				
		$-\infty$	$-Fn$	0	$+Fn$	$+\infty$
Fd[rs1]	$-\infty$	--- $-\infty$	NV dNaN			--- $-\infty$
	$-Fn$	NV dNaN	--- Q(Fd[rs1] : Fd[rs2])			NV dNaN
	0		--- 0			
	$+Fn$		--- Q(Fd[rs1] : Fd[rs2])			
	$+\infty$	--- $+\infty$	NV dNaN			--- $+\infty$

FRQUAod converts the Oracle floating-point number in Fd[rs1] into a cohort member with the extended exponent (page 21) is specified by Fd[rs2]. The rounded result is converted into an Oracle floating-point number and written in Fd[rd]. An extended exponent can be used to round an arbitrary digit of the decimal number. The result is not normalized.

When the exponent of Fd[rs1] more than double the extended exponent specified by Fd[rs2], Fd[rs1] is written in Fd[rd].

When the significand is 0 due to underflow or rounding, the result remains 0. This 0 is not the special value 0. The sign is the same as Fd[rs1], the exponent is the extended exponent specified by Fd[rs2], and 0 corresponding to the sign is written in the significand.

The result is rounded as specified by GSR.dirnd when GSR.dim = 1, and FSR.drd when GSR.dim = 0.

The rounded result, which is expressed in terms of the extended exponent, is converted to an Oracle floating-point number and then written in Fd[rd].

- When the extended exponent is even, the converted Oracle floating-point number consists of the sign of the rounded result, an exponent equivalent to the extended exponent of the rounded result, and the significand of the rounded result.
- When the extended exponent is odd, the converted Oracle floating-point number consists of the sign of the rounded result, an exponent equivalent to the extended exponent of the rounded result minus one, and a significand which is the rounded result's significand shifted left by one digit.

Table 7-41 FRQUAod

		Fd[rs2]	
		$\exp 10 \leq \exp(Fd[rs1]) \times 2$	$\exp 10 > \exp(Fd[rs1]) \times 2$
Fd[rs1]	$-\infty$	NV dNaN	
	$-Fn$	— Fd[rs1]	—
	0		RQ(Fd[rs1] : Fd[rs2])
	$+Fn$		
	$+\infty$	NV dNaN	

Table 7-42 shows examples of FRQUAod when the exponent of Fd[rs1] is 10, the significand is 11.223344556677, and the rounding mode is 4 (towards nearest, away from 0 if tie) .

Table 7-42 Examples of FRQUAod

Fd[rs2]	Processing	Exp	Extended Exp	Significand (underflow)	Remarks
Extended exponent 21	1) Convert to extended exponent	10.5	21	01.122334455667(7)	Right shift 1 digit
	2) Rounding process	10.5	21	01.122334455668	Round at 10^{21-13}
	3) Exponent correction	10	20	11.223344556680	Left shift 1 digit
Extended exponent 22	1) Convert to extended exponent	11	22	00.112233445566(7)	Right shift 2 digits
	2) Rounding process	11	22	00.112233445567	Round at 10^{22-13}
	3) Exponent correction	11	22	00.112233445567	No shift
Extended exponent 23	1) Convert to extended exponent	11.5	23	00.011223344556(6)	Right shift 3 digits
	2) Rounding process	11.5	23	00.011223344557	Round at 10^{23-13}
	3) Exponent correction	11	22	00.112233445570	Left shift 1 digit
Extended exponent 24	1) Convert to extended exponent	12	24	00.001122334455(6)	Right shift 2 digits
	2) Rounding process	12	24	00.001122334456	Round at 10^{24-13}
	3) Exponent correction	12	24	00.001122334456	No shift unnecessary.
Extended exponent 32	1) Convert to extended exponent	16	32	00.000000000011(2)	Right shift 12 digits
	2) Rounding process	16	32	00.000000000011	Round at 10^{32-13}
	3) Exponent correction	16	32	00.000000000011	No shift
Extended exponent 33	1) Convert to extended exponent	16.5	33	00.000000000001(1)	Right shift 13 digits
	2) Rounding process	16.5	33	00.000000000001	Round at 10^{33-13}
	3) Exponent correction	16	32	00.000000000010	Left shift 1 digit
Extended exponent 34	1) Convert to extended exponent	17	34	00.000000000000(1)	Right shift 14 digits
	2) Rounding process	17	34	00.000000000000	Round at 10^{34-13}
	3) Exponent correction	17	34	00.000000000000	No shift.

The results of FADDod, FSUBod, FMULod, FDIVod, FQUAod, and FRQUAod are rounded as specified by GSR.dirnd when GSR.dim = 1, or FSR.drd when GSR.dim = 0. The results of FXADDodLO, FXADDodHI, and FXMULodLO are rounded with rounding mode 1 (towards 0) regardless of the settings in FSR or GSR.

When the results of FADDod, FSUBod, FMULod, FDIVod, FXADDodLO, FXADDodHI, and FXMULodLO are zero after rounding, the output is the special value 0 (page 20). If the result is smaller than Nmin (page 20), the output is 0 or Nmin depending on the rounding mode. If the result is not a special value, the output is normalized, except for FQUAod and FRQUAod.

When either Fd[rs1] or Fd[rs2] is written to Fd[rd] as a result of FADDod, FSUBod, FMULod, FDIVod, FXADDodLO, FXADDodHI, FXMULodLO, FQUAod, or FRQUAod, the digits outside the range of the significand (page 18) are converted to the number 0.

Exception		Target instruction	Condition
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<2:1> ≠ 0
<i>fp_exception_ieee_754</i>	OF	FADDod, FSUBod, FMULod, FDIVod, FXADDodHI, FXMULodLO, FRQUAod	Result is larger than N_{max} (page 20)
	UF	FADDod, FSUBod, FMULod, FDIVod, FXADDodHI, FXADDodLO, FXMULodLO	Result is smaller than N_{min} (page 20)
	NX	All	Result cannot be expressed as an Oracle floating-point number For FXADDodHI, FXADDodLO and FXMULodLO, only when OF or UF.
	NV	All	Refer to tables in the description
	DZ	FDIVod	Refer to the table in the description

7.119. Decimal Floating-Point Compare

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FCMPtd	0 1010 0100 ₂	Compare decimal floating point numbers	✓.		fcmp _{td}	%fccn, freg _{rs1} , freg _{rs2}
FCMPetd	0 1010 0101 ₂	Compare decimal floating point numbers (exception if non-ordinal).	✓		fcmp _{etd}	%fccn, freg _{rs1} , freg _{rs2}

10 ₂		—		cc1	cc0	op3 = 11 0110 ₂		rs1		opf		rs2					
31	30	29		27	26	25	24	19	18		14	13		5	4		0

Description FCMPtd and FCMPetd compare two double-precision decimal floating-point numbers Fd[rs2] and Fd[rs1]. The result is written in FSR.fccn. Cohorts members are considered equal.

When either number is SNaN, FCMPtd detects an exception.

When either number is SNaN or QNaN, FCMPetd detects an exception.

Table 7-43 FCMPtd and FCMPetd

		rs2								
		−∞	−Fn	−0	+0	+Fn	+∞	QNaN	SNaN	
rs1	−∞	— 0								
	−Fn	— 0, 1, 2	— 1							
	−0			— 0						
	+0									
	+Fn	— 2	— 0, 1, 2							
	+∞						— 0			
	QNaN	NV ^{xxvii} 3								
	SNaN	NV 3								

Instructions that compare single-precision or quadruple-precision decimal floating-point numbers are not defined.

^{xxvii} FCMPetd instruction only

Exception		Target instruction	Condition
<i>illegal_instruction</i>		All	A <i>reserved</i> field is not 0. (iw<29:27> ≠ 0)
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd ≠ 0
<i>fp_exception_ieee_754</i>	NV	FCMPtd	FSR.tem.nv ≠ 0, and either input is SNaN
		FCMPetd	FSR.tem.nv ≠ 0, and FSR.fccn ≠ 3

7.120. Oracle Decimal Floating-Point Compare

Compatibility Note Future compatibility of Oracle decimal floating-point compare is not guaranteed. This instruction should only be used in libraries for the SPARC64™ X or SPARC64 X+ platform.

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FCMPod	0 1011 0100 ₂	Compare Oracle floating point numbers	✓		fcmpod %fccn, freg _{rs1} , freg _{rs2}

10 ₂	—	cc1	cc0	op3 = 11 0110 ₂	rs1	opf	rs2
31 30 29	27 26 25 24	19 18	14 13	5 4	0		

Description FCMPod compares two Oracle format floating-point numbers Fd[rs2] and Fd[rs1]. The result is written in FSR.fccn.

Table 7-44 **Operation result of FCMPod**

		rs2				
		$-\infty$	$-Fn$	0	$+Fn$	$+\infty$
rs1	$-\infty$	$\overline{\hspace{1cm}}$ 0	$\overline{\hspace{1cm}}$			
	$-Fn$	$\overline{\hspace{1cm}}$ 0, 1, 2		$\overline{\hspace{1cm}}$ 1		
	0	$\overline{\hspace{1cm}}$ 0		$\overline{\hspace{1cm}}$		
	$+Fn$	$\overline{\hspace{1cm}}$ 2		$\overline{\hspace{1cm}}$ 0, 1, 2		
	$+\infty$	$\overline{\hspace{1cm}}$				$\overline{\hspace{1cm}}$ 0

Exception	Target instruction	Condition
<i>Illegal_instruction</i>	All	A reserved field is not 0. (iw<29:27> ≠ 0)
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>Illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd ≠ 0

7.121. Decimal Floating-Point Convert

Compatibility Note Future compatibility of Oracle floating-point numbers is not guaranteed. This format should only be used for libraries for the SPARC64™ X or SPARC64 X+ platform.

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FbuxT0td	1 1010 1100 ₂	Convert decimal floating point	rd : basic only		fbuxt0td	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FtdT0bux	1 1010 1101 ₂	Convert decimal floating point	rd : basic only		ftdt0bux	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FbsxT0td	1 1010 1110 ₂	Convert decimal floating point	rd : basic only		fbsxt0td	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FtdT0bsx	1 1010 1111 ₂	Convert decimal floating point	rd : basic only		ftdt0bsx	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FodT0td	1 1011 1110 ₂	Convert decimal floating point	rd : basic only		fodt0td	<i>freg_{rs2}</i> , <i>freg_{rd}</i>
FtdT0od	1 1011 1111 ₂	Convert decimal floating point	rd : basic only		ftdt0od	<i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0110 ₂	—	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description FbuxT0td converts a 64-bit unsigned BCD integer Fd[rs2] into a double-precision decimal floating point number. The result is written in Fd[rd]. The exponent of the decimal-floating point number after the conversion is 0 (bias only).

Because the range of values that can be represented by a decimal floating-point number is larger than the range of an unsigned BCD number, the conversion result is always exact.

When a digit outside the range 0000₂ – 1001₂ is present in the BCD integer, an NV exception is generated.

FbsxT0td converts a 64-bit signed integer BCD Fd[rs2] into a double-precision decimal floating-point number. The result is written in Fd[rd]. The exponent of the decimal floating-point number after the conversion is 0 (bias only).

Because the range of values that can be represented by a decimal floating-point number is larger than the range of a signed BCD number, the conversion result is always exact.

The lowest four bits of the BCD integer encode the sign of the number. Table 4-6 shows how FbsxT0td interprets the sign field.

When a digit outside the range 0000₂ – 1001₂ is present in a BCD integer, excluding the sign field, an NV exception is generated.

FtdT0bux converts the significand of a double-precision decimal floating-point number Fd[rs2] into a 64-bit signed BCD integer. The result is written in Fd[rd]. Because the range that can be represented by an unsigned BCD number is the same as the range of the significand of a decimal floating-point number, the conversion result is always exact.

Programming Note Because FtdT0bux ignores the exponent, first use FQUAtd to make the biased exponent 0.

If a special value (∞ or NaN) is input, the value of the Combination field (exponent, NaNs, and ∞) is ignored, and no exception is generated. When the LMD cannot be determined because the significand is ∞ or NaN, the LMD is treated as 0.

FtdT0bsx converts the significand of a double-precision decimal floating-point number **Fd[rs2]** into a 64-bit signed BCD integer. The result is written in **Fd[rd]**. The exponent part is ignored.

A signed BCD can represent 15 decimal digits and has a smaller range than a decimal floating-point number, which has 16 digits. The number after conversion is the signed lower 15 digits of the decimal floating-point number. When the upper digit is thrown away, no exception is generated.

Note For **F{sdq}T0i** and **F{sdq}T0x**, an exception is generated if **FSR.tem.nvm** = 1. If **FSR.tem.nvm** = 0, the result is the maximum value (if negative, minimum value) that can be shown.

The left part of the significand is not a target of the conversion and is ignored. If a special value (∞ and NaN) is input, the value of the Combination field (exponent, NaNs, and ∞) is ignored, and no exception is generated. The LMD is not converted, so it doesn't matter if the LMD cannot be determined because the significand is ∞ or NaN.

FodT0td converts Oracle floating-point number **Fd[rs2]** into a double-precision decimal floating-point number. The result is written in **Fd[rd]**. No exceptions are generated.

- If a special value (∞ , 0) is input

The special value is converted to the corresponding value in the output format.

When **Fd[rs2]** is 0, the significand is 0 and the exponent is the exponent of **Fd[rs2]**.

FtdT0od converts a double-precision decimal floating-point number **Fd[rs2]** into an Oracle floating-point number. The result is written in **Fd[rd]**.

- About OF and UF

When the exponent of the rounded result of the conversion of **Fd[rs2]** is larger than **E_{max}** of the output format, an OF exception is generated. The result is ∞ or **N_{max}**.

When the exponent of the rounded result of the conversion of **Fd[rs2]** is smaller than **E_{min}** of the output format, a UF exception is generated. The result is 0 or **N_{min}**.

- If a special value (∞ , NaN) is input

The special value is converted to the corresponding value in the output format.

When the input is NaN, the results is dNaN, and an NV exception is generated.

Exception		Target instruction	Condition
<i>Illegal_instruction</i>		All	A reserved field is not 0. (iw<18:14> $\neq$ 0)
<i>fp_disabled</i>		All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>		All	XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1 $\neq$ 0 • XAR.urs2<1> $\neq$ 0 • XAR.urs3 $\neq$ 0 • XAR.urd<2:1> $\neq$ 0
<i>fp_exception_ieee_754</i>	OF	FtdT0od	Refer to the description
	UF	FtdT0od	Refer to the description
	NX	FtdT0od	Refer to the description
	NV	FbuxT0td, FbsxT0td, FtdT0od	Refer to the description

7.122. Shift Mask Or (for SPARC64™ X)

Note For the specification of this instruction on SPARC64™ X+, refer to page 252.

Instruction	var	size	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FSHIFTORX	10 ₂	11 ₂	Concatenate two double-precision floating-point registers	✓	✓	fshiftorx <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rs3}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0111 ₂	rs1	rs3	var = 10 ₂	size = 11 ₂	rs2
31 30 29	25 24	19 18	14 13	9 8	7 6	5 4	0

Non-SIMD execution

FSHIFTORX shifts Fd[rs1] right or left and extracts part of the result. It also shifts Fd[rs2] right or left and extracts part of the result. The two extracted values are bitwise ORed, and this result is written in Fd[rd]. The shift count and shift direction are specified by Fd[rs3].

Table 7-45 Meaning of bits in Fd[rs3]

set_rs2_default	—	rs1_mask_inv	rs1_mask_offset	rs1_mask_length	rs1_shift_amount
63	62	56	55	48	47
	57				40
					39
					32
—	rs2_mask_inv	rs2_mask_offset	rs2_mask_length	rs2_shift_amount	
31	25	24	23	16	15
					8
					7
					0

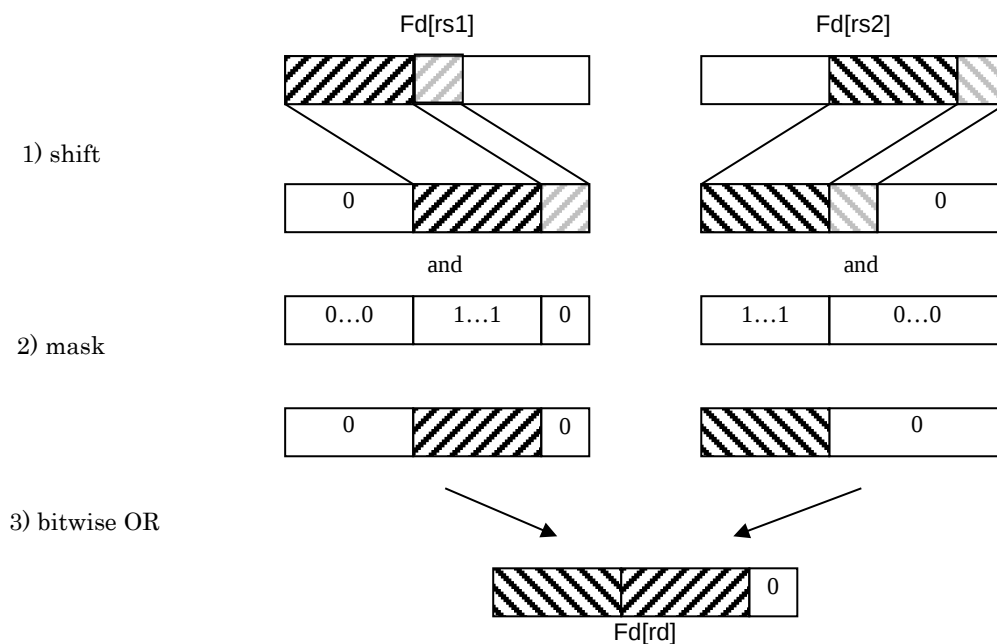
Bit	Field	Description
rs1 rs2		
63 -	set_rs2_default	Specifies set of rs{1 2}_* fields to use for rs2 0: Use the rs2_* fields 1: Use values derived from the rs1_* fields
56 24	rs{1 2}_mask_inv	Specifies whether to invert the mask 0: Do not invert 1: Invert
55:48 23:16	rs{1 2}_mask_offset	Starting bit position in mask (number of bits from MSB), 8-bit signed integer
47:40 15:8	rs{1 2}_mask_length	Mask bit length, 8-bit signed integer
39:32 7:0	rs{1 2}_shift_amount	Shift count (positive: left shift, negative: right shift), 8-bit signed integer

Note Specifying the following values in Fd[rs3] causes an *illegal_instruction* exception.

- 1) '1' in a *reserved* field
- 2) set_rs2_default = 1 and <31:0> ≠ 0
- 3) rs{1|2}_mask_offset, rs{1|2}_mask_length, or rs{1|2}_shift_amount ≠ multiple of 8

Note The FSHIFTORX instruction causes an *illegal_instruction* exception for certain values in the register Fd[rs3]. An *fp_disabled* or *illegal_action* exception may be detected before the register is read. That is, for this instruction, these exceptions may have higher priority than an *illegal_instruction* exception.

The behavior of FSHIFTORX is divided into 1) shift, 2) mask and 3) OR operations.



The shift operation executes a logical shift. If shifting left, the vacated positions on the right are replaced by 0. If shifting right, the vacated positions on the left are replaced by 0. The `rs{1|2}_shift_amount` field specifies both the shift direction and shift count. The shift is leftwards if the value is positive and rightwards if the value is negative. Only 0 and multiples of 8 can be specified for the shift count. If any other value is specified, an *illegal_instruction* exception will occur.

The mask operation extracts the specified bits from the shifted register. Two mask patterns are defined by `rs{1|2}_mask_offset`, `rs{1|2}_mask_length`, and `rs{1|2}_mask_inv`. A mask pattern is generated from one set of these fields and bitwise ANDed with the corresponding shifted register.

A mask pattern is a generated range of contiguous 1s in a 64-bit doubleword, where no bits are 1 outside this range. The inverse of this pattern can also be specified. The `rs{1|2}_mask_offset` field specifies the starting position of the mask (the range of 1s) in the register as the number of bits from the left (MSB). The `rs{1|2}_mask_length` field specifies the length of the range of bits that are set to 1. To invert this mask, specify 1 for `rs{1|2}_mask_inv`.

Only 0 and multiples of 8 can be specified for `rs{1|2}_mask_offset` and `rs{1|2}_mask_length`. If any other value is specified for these fields, an *illegal_instruction* exception will occur.

Note The mask pattern contains bits that are 1 when $0 \leq \text{rs}\{1|2\}_mask_offset < 64$ and $0 < \text{rs}\{1|2\}_mask_length$. That is, the pattern fits inside a doubleword. The specified pattern can exceed the length of a doubleword if $64 < \text{rs}\{1|2\}_mask_offset + \text{rs}\{1|2\}_mask_length$. In this case, the mask pattern is a contiguous range of 1s from `rs{1|2}_mask_offset` to the LSB.

Note The mask pattern may contain bits that are 1 even if the value of `rs{1|2}_mask_offset` is negative. Specifically, when the absolute value of `rs{1|2}_mask_offset` is less than `rs{1|2}_mask_length`, the mask pattern is a contiguous range of 1s from the MSB to `rs{1|2}_mask_length - |rs{1|2}_mask_offset|`.

Note The 64-bit mask pattern is all 0s when `rs{1|2}_mask_offset` and `rs{1|2}_mask_length` satisfy any of the conditions below. The mask pattern is all 1s under the same conditions when `rs{1|2}_mask_inv` is used to invert the mask pattern.

- `rs{1|2}_mask_length ≤ 0`
- `rs{1|2}_mask_offset + rs{1|2}_mask_length ≤ 0`
- `rs{1|2}_mask_offset ≥ 64`

The values obtained from `Fd[rs1]` and `Fd[rs2]` using the shift and mask operations are bitwise ORed. The result is written in `Fd[rd]`. Table 7-46 shows an example of how to specify the bits in `Fd[rs3]`.

Table 7-46 **Example of how to specify bits in `Fd[rs3]`**

Field	Concatenate lower 32 bits of <code>Fd[rs1]</code> and lower 32 bits of <code>Fd[rs2]</code> (<code>Fd[rs1]<31:0>::Fd[rs2]<31:0></code>)
<code>set_rs2_default</code>	0 (<code>rs2_*</code> fields are used.)
<code>rs1_mask_inv</code>	0 (Do not invert the <code>rs1</code> mask)
<code>rs1_mask_offset</code>	0 (Mask shifted <code>rs1</code> register starting from MSB)
<code>rs1_mask_length</code>	32 (Mask length is 32 bits)
<code>rs1_shift_amount</code>	32 (Shift left 32 bits)
<code>rs2_mask_inv</code>	0 (Do not invert the <code>rs2</code> mask)
<code>rs2_mask_offset</code>	32 (Mask shifted <code>rs2</code> register starting from bit<31>)
<code>rs2_mask_length</code>	32 (Mask length is 32 bits)
<code>rs2_shift_amount</code>	0 (No shift)

If 1 is specified for `set_rs2_default`, the shift and mask operations use values for `rs2_mask_inv`, `rs2_mask_offset`, `rs2_mask_length`, and `rs2_shift_amount` that are derived from the corresponding `rs1_*` fields. This behavior is useful when concatenating lower bits of `Fd[rs1]` and upper bits of `Fd[rs2]`, since only the `rs1_*` fields need to be specified. When `set_rs2_default = 1` and values other than 0 are specified in the `rs2_*` fields, an *illegal_instruction* exception will occur.

Table 7-47 shows how the values for `rs2_*` are derived from the corresponding `rs1_*` fields. An example of how to specify the bits in `Fd[rs3]` when `set_rs2_default = 1` is shown in Table 7-48.

Table 7-47 **Derived values for `rs2_*` fields**

Field	Derived value
<code>rs2_mask_inv</code>	<code>rs1_mask_inv</code>
<code>rs2_mask_offset</code>	<code>rs1_mask_offset</code>
<code>rs2_mask_length</code>	<code>rs1_mask_length</code>
<code>rs2_shift_amount</code>	<code>rs1_shift_amount - 64</code>

Table 7-48 Example of how to specify bits in Fd[rs3] when set_rs2_default = 1t.

Field	Concatenate lower 8 bits of Fd[rs1] and upper 24 bits of Fd[rs2] in this order. (Fd[rs1]<7:0>::Fd[rs2]<63:40>)	
set_rs2_default	1 (Use rs1_* fields to derive values for rs2_*.)	
rs1_mask_inv	0 (Do not invert the rs1 mask)	
rs1_mask_offset	0 (Mask shifted rs1 register starting from MSB)	
rs1_mask_length	32 (Mask length is 32 bits)	
rs1_shift_amount	56 (Shift left 56 bits)	
rs2_mask_inv	Set value: 0 (Not used)	Derived value: 0 (Do not invert the derived rs2 mask)
rs2_mask_offset	Set value: 0 (Not used)	Derived value: 0 (Mask shifted rs2 register starting from MSB)
rs2_mask_length	Set value: 0 (Not used)	Derived value: 32 (Mask length is 32 bits)
rs2_shift_amount	Set value: 0 (Not used)	Derived value: -8 (Shift right eight bits)

SIMD execution When FSHIFTORX is used as a SIMD instruction, any floating point register Fd[0] – Fd[126], Fd[256] – Fd[382] can be specified for Fd[rs1]. If an extended register Fd[256] – Fd[382] is specified for Fd[rs1], the extended register Fd[n] is used for the basic operation and the basic register Fd[n – 256] is used for the extended operation. On the other hand, only basic registers Fd[0] – Fd[126] can be specified for Fd[rs2], Fd[rs3], and Fd[rd]. The basic operation uses the basic register Fd[n], and the extended operation uses the extended register Fd[n + 256].

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	When XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<1> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urs3<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>illegal_instruction</i>	All	Refer to the description.

7.123. SIMD Compare (for SPARC64™ X)

Note For the specification of this instruction on SPARC64™ X+, refer to page 257.

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FCMPLE16 X	0 1100 000 0 ₂	Compare four 16 bit signed integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fcmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPLE16 X	0 1100 0001 ₂	Compare four 16 bit unsigned integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fucmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPNE16 X	0 1100 0011 ₂	Compare four 16 bit unsigned integer. If $src1 \neq src2$, then 1.	rd : basic only		<i>fucmpne16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FCMPLE32X	0 1100 0100 ₂	Compare two 32 bit signed integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fcmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPLE32 X	0 1100 0101 ₂	Compare two 32 bit unsigned integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fucmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPNE32 X	0 1100 0111 ₂	Compare two 32-bit unsigned integer. If $src1 \neq src2$, then 1.	rd : basic only		<i>fucmpne16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FCMPGT16X	0 1100 1000 ₂	Compare four 16-bit signed integer. If $src1 > src2$, then 1.	rd : basic only		<i>fcmpgt16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPGT16 X	0 1100 1001 ₂	Compare four 16 bit unsigned integer. If $src1 > src2$, then 1.	rd : basic only		<i>fucmpgt16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPEQ16 X	0 1100 1011 ₂	Compare four 16 bit unsigned integer. If $src1 = src2$, then 1.	rd : basic only		<i>fucmpeq16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FCMPGT32X	0 1100 1100 ₂	Compare two 32-bit signed integer. If $src1 > src2$, then 1.	rd : basic only		<i>fcmpgt32x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPGT32 X	0 1100 1101 ₂	Compare two 32 bit unsigned integer. If $src1 > src2$, then 1.	rd : basic only		<i>fucmpgt32x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPEQ32 X	0 1100 1111 ₂	Compare two 32 bit unsigned integer. If $src1 = src2$, then 1.	rd : basic only		<i>fucmpeq32x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FCMPLE8X	0 1101 0000 ₂	Compare eight 8-bit signed integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fcmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPLE8X	0 1101 0001 ₂	Compare eight 8 bit unsigned integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fucmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FUCMPNE8X	0 1101 0011 ₂	Compare eight 8 bit unsigned integer. If $src1 \neq src2$, then 1.	rd : basic only		<i>fucmpne16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FCMPLEX	0 1101 0100 ₂	Compare 64-bit signed integer. If $src1 \leq src2$, then 1.	rd : basic only		<i>fcmp16x</i> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FUCMPLEX	0 1101 0101 ₂	Compare 64 bit unsigned integer If $src1 \leq src2$, then 1	rd : basic only		fucmplex	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FUCMPNEX	0 1101 0111 ₂	Compare 64 bit unsigned integer. If $src1 \neq src2$, then 1	rd : basic only		fucmpnex	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FCMPGT8X	0 1101 1000 ₂	Compare eight 8- bit signed integer If $src1 > src2$, then 1	rd : basic only		fcmpgt8x	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FUCMPGT8X	0 1101 1001 ₂	Compare eight 8 bit unsigned integer If $src1 > src2$, then 1	rd : basic only		fucmpgt8x	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FUCMPEQ8X	0 1101 1011 ₂	Compare eight 8 bit unsigned integer If $src1 = src2$, then 1	rd : basic only		fucmpeq8x	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FCMPGTX	0 1101 1100 ₂	Compare 64- bit signed integer If $src1 > src2$, then 1	rd : basic only		fcmpgtx	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FUCMPGTX	0 1101 1101 ₂	Compare 64- bit unsigned integer. If $src1 > src2$, then 1	rd : basic only		fucmpgtx	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FUCMPEQX	0 1101 1111 ₂	Compare 64- bit unsigned integer. If $src1 = src2$, then 1	rd : basic only		fucmpeqx	$freg_{rs1}, freg_{rs2}, freg_{rd}$

10 ₂	rd			op3 = 11 0110 ₂	rs1			opf	rs2		
31 30 29	25 24			19 18	14 13			5 4	0		

Description

These instructions compare the elements (partitions) in the two floating-point registers Fd[rs1] and Fd[rs2]. The result is written in the floating-point register Fd[rd]. The comparison results for these elements are written in the most-significant bits of Fd[rd]. 0s are written in the other bits.

The number of elements in a 64-bit input register depends on the data type of the comparison. The number of elements and their bit ranges for each data type are shown in Table 7-49.

Table 7-49 Number of elements (#E) and their bit ranges (E) for each data type

Data Type	#E	E1	E2	E3	E4	E5	E6	E7	E8
8-bit signed integer	8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
8-bit unsigned integer	8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
16-bit signed integer	4	63:48	47:32	31:16	15:0	—	—	—	—
16-bit unsigned integer	4	63:48	47:32	31:16	15:0	—	—	—	—
32-bit signed integer	2	63:32	31:0	—	—	—	—	—	—
32-bit unsigned integer	2	63:32	31:0	—	—	—	—	—	—
64-bit signed integer	1	63:0	—	—	—	—	—	—	—
64-bit unsigned integer	1	63:0	—	—	—	—	—	—	—

Elements of Fd[rs1] and Fd[rs2] that occupy the same bit range are compared. The result is written in the corresponding bit of Fd[rd]. The bit positions of Fd[rd] corresponding to each element are shown in Table 7-50.

Table 7-50 Elements and corresponding bit positions in Fd[rd]

	E1	E2	E3	E4	E5	E6	E7	E8
Fd[rd]	63	62	61	60	59	58	57	56

FCMPLE{8X, 16X, 32X, X} compares the elements of Fd[rs1] and Fd[rs2] as signed integers. If “element of Fd[rs1]” ≤ “element of Fd[rs2]”, the corresponding bit of Fd[rd] is set to 1.

FCMPGT{8X, 16X, 32X, X} compares the elements of Fd[rs1] and Fd[rs2] as signed integers. If “element of Fd[rs1]” > “element of Fd[rs2]”, the corresponding bit of Fd[rd] is set to 1.

FUCMPLE{8X, 16X, 32X, X} compares the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “element of Fd[rs1]” ≤ “element of Fd[rs2]”, the corresponding bit of Fd[rd] is set to 1.

FUCMPNE{8X, 16X, 32X, X} compares the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “element of Fd[rs1]” ≠ “element of Fd[rs2]”, the corresponding bit of Fd[rd] is set to 1.

FUCMPGT{8X, 16X, 32X, X} compares the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “element of Fd[rs1]” > “element of Fd[rs2]”, the corresponding bit of Fd[rd] is set to 1.

FUCMPEQ{8X, 16X, 32X, X} compares the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “element of Fd[rs1]” = “element of Fd[rs2]”, the corresponding bit of Fd[rd] is set to 1.

Note Instructions that compare whether signed integers are equal or not equal are not defined. These comparisons are equivalent to the FUCMPEQ{8X, 16X, 32X, X} and FUCMPNE{8X, 16X, 32X, X} instructions, which respectively compare whether unsigned integers are equal or not equal.

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	When XAR.v = 1 and any of the following are true • XAR.simd = 1 • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<2:1> ≠ 0

7.124. Leading Zero Detect

Instruction	opf	Operation	HPC-ACE	Assembly Language Syntax
			Regs. SIMD	
LZD	0 0001 0111 ₂	Counts number of 0 from left end of R[rs2]		lzd <i>reg_{rs2}, reg_{rd}</i>

Refer to Section 7.85 in UA2011.

Compatibility Note In UA2011, the name of this instruction is LZCNT.

Exception	Condition
<i>illegal_instruction</i>	iw<18:14> ≠ 0
<i>illegal_action</i>	XAR.v = 1

7.125. Fixed-point Partitioned Add (64-bit)

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPADD64	0 0100 0010 ₂	64-bit addition	✓	✓	<code>fpadd64</code> <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.52 in UA2011.

FPADD64 adds the 8-byte integer in Fd[rs1] and the 8-byte integer in Fd[rs2]. The lower 8 bytes of the result is written in Fd[rd].

FPADD64 does not update any fields in FSR.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.126. Fixed-point Partitioned Subtract (64-bit)

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPSUB64	0 0100 0110 ₂	64-bit subtraction	✓	✓	fpsub64 <i>freg_{rs1}, freg_{rs2}, freg_{rd}</i>

Refer to Section 7.58 in UA2011.

FPSUB64 subtracts the 8-byte integer in Fd[rs2] from the 8-byte integer in Fd[rs1]. The lower 8 bytes of the result are stored in Fd[rd].

FPSUB64 does not update any fields in FSR.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.127. SIMD Unsigned Compare

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Refer to Section 7.55 in UA2011.

FPCMPU{LE|NE|GT|EQ}8 do not update any fields in FSR.

Compatibility Note FPCMPUNE8 and FPCMPUEQ8 on SPARC64™ X+ are compatible with FPCMP{ |U}NE8 and FPCMP{ |U}EQ8 in UA2011, respectively.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1

7.128. Floating-Point Lexicographic Compare

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FLCMPS	1 0101 0001 ₂	Single-precision lexicographic compare	✓		flcmps %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FLCMPd	1 0101 0010 ₂	Double-precision lexicographic compare	✓		flcmpd %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>

Refer to Section 7.37 in UA2011.

Table 7-51 FLCMP{s|d}

		F[rs2]							
		$-\infty$	-N	-0	+0	+N	$+\infty$	QNaN2	SNaN2
F[rs1]	$-\infty$							3	
	-N	$\overline{\hspace{1cm}}$ 0, 1		$\overline{\hspace{1cm}}$ 1					
	-0								
	+0								
	+N	$\overline{\hspace{1cm}}$ 0		$\overline{\hspace{1cm}}$ 0, 1					
	$+\infty$								
	QNaN1								
	SNaN1	2							

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>Illegal_instruction</i>	iw<29:27> ≠ 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd ≠ 0 • XAR.simd = 1

7.129. Floating-Point Negative Add

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FNADDs	0 0101 0001 ₂	Floating point negative add single	✓	✓	fnadds	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FNADDd	0 0101 0010 ₂	Floating point negative add double	✓	✓	fnadd	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

Refer to Section 7.47 in UA2011.

Table 7-52 FNADD{s|d}

		F[rs2]							
		$-\infty$	$-N$	-0	$+0$	$+N$	$+\infty$	QNaN2	SNaN2
F[rs1]	$-\infty$	$\frac{\text{---}}{+\infty}$					NV dQNaN	$\frac{\text{---}}{\text{QNaN2}}$	NV QNaN2
	$-N$	$\frac{\text{---}}{-(F[rs1] + F[rs2])}$		$\frac{\text{---}}{-F[rs1]}$		$\frac{\text{---}}{-(F[rs1] + F[rs2])^{xxviii}}$			
	-0	$\frac{\text{---}}{-F[rs2]}$	$\frac{\text{---}}{-0}$	$\frac{\text{---}}{+0^{xxix}}$	$\frac{\text{---}}{-F[rs2]}$				
	$+0$		$\frac{\text{---}}{+0^{xxix}}$	$\frac{\text{---}}{+0}$					
	$+N$	$\frac{\text{---}}{-(F[rs1] + F[rs2])^{xxviii}}$		$\frac{\text{---}}{-F[rs1]}$		$\frac{\text{---}}{-(F[rs1] + F[rs2])}$			
	$+\infty$	NV dQNaN	$\frac{\text{---}}{-\infty}$						
	QNaN1	$\frac{\text{---}}{\text{QNaN1}}$							
	SNaN1	NV QNaN1							

^{xxviii} When the result is 0, footnote (xxix) applies.

^{xxix} When the rounding mode is towards -∞, the result is -0.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true. • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>fp_exception_ieee_754</i>	Same as FADD{s d}
<i>fp_exception_other</i>	Same as FADD{s d}

7.130. Floating-Point Negative Multiply

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FNMULS	0 0101 1001 ₂	Floating-point negative multiply single	✓	✓	fnmuls	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FNMULD	0 0101 1010	Floating-point negative multiply double	✓	✓	fnmuld	$freg_{rs1}, freg_{rs2}, freg_{rd}$
FNSMULD	0 0111 1001	Floating-point negative multiply single to double	✓	✓	fnsmuld	$freg_{rs1}, freg_{rs2}, freg_{rd}$

Refer to Section 7.50 in UA2011.

Table 7-53 FNMUL{s|d}, FNSMULd

		F[rs2]							
		$-\infty$	-N	-0	+0	+N	$+\infty$	QNaN2	SNaN2
F[rs1]	$-\infty$	$\frac{\text{---}}{\text{---}}$		NV dNaN			$\frac{\text{---}}{\text{---}}$	$\frac{\text{---}}{\text{QNaN2}}$	NV QNaN2
	-N		$\frac{\text{---}}{-(F[rs1] \times F[rs2])}$			$\frac{\text{---}}{-(F[rs1] \times F[rs2])}$			
	-0	NV dQNaN		$\frac{\text{---}}{-0}$	$\frac{\text{---}}{+0}$		NV dQNaN		
	+0			$\frac{\text{---}}{+0}$	$\frac{\text{---}}{-0}$				
	+N		$\frac{\text{---}}{-(F[rs1] \times F[rs2])}$		$\frac{\text{---}}{-(F[rs1] \times F[rs2])}$				
	$+\infty$	$\frac{\text{---}}{\text{---}}$		NV dNaN			$\frac{\text{---}}{\text{---}}$		
	QNaN1	$\frac{\text{---}}{\text{QNaN1}}$							
	SNaN1	NV QNaN1							

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>fp_exception_ieee_754</i>	Same as FMUL{s d}, FsmULd
<i>fp_exception_other</i>	Same as FMUL{s d}, FsmULd

7.131. WRPAUSE(PAUSE)

Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
WRPAUSE	11 0000 ₂	Pause VCPU for specified number of cycles.			<i>wr</i> <i>reg_{rs1}, reg_or_imm, %pause</i>
PAUSE	11 0000 ₂	Pause VCPU for specified number of cycles.			<i>pause</i> <i>reg_or_imm</i>

10 ₂	rd = 1 1011 ₂	op3 = 11 0000 ₂	rs1	i = 0	—	rs2
10 ₂	rd = 1 1011 ₂	op3 = 11 0000 ₂	rs1	i = 1	simm13	
31 30 29	25 24	19 18	14 13 12	5 4	0	

Description WRPAUSE and PAUSE stop the VCPU for the specified number of processor cycles.

The WRPAUSE and PAUSE instructions write the number of cycles in the PAUSE register (ASR27). The PAUSE register has the following fields.

63	15	14	3	2	0
—	—	Pause	—	—	—

Bit	Field	Access	Description
63:15	<i>Reserved</i>	WO	Reserved
14:3	Pause	WO	Specifies the number of cycles the VCPU is paused. Can be accessed in nonprivileged mode.
2:0	<i>Reserved</i>	WO	Ignored

When *i* = 0, WRPAUSE writes ($\min(2^{15} - 1, (R[rs1] \mathbf{xor} R[rs2])) \gg 3$) into the pause field (PAUSE<14:3>) of the PAUSE register. When *i* = 1, WRPAUSE writes ($\min(2^{15} - 1, (R[rs1] \mathbf{xor} \mathbf{sign_ext}(simm13))) \gg 3$) into the pause field.

When *i* = 0, PAUSE writes ($\min(2^{15} - 1, R[rs2]) \gg 3$) into the pause field (PAUSE<14:3>) of the PAUSE register. When *i* = 1, PAUSE writes ($\min(2^{15} - 1, \mathbf{sign_ext}(simm13)) \gg 3$) into the pause field.

Programming Note The behavior of PAUSE is the same as WRPAUSE with *rs1* = 0.

The number of cycles the VCPU will be paused is the lower 15 bits (PAUSE<14:0>) of the PAUSE register. However, bits PAUSE<2:0> are ignored. The pause field (PAUSE<14:3>) is decremented by 1 every 8 CPU clock cycles. Therefore, the maximum number of cycles which the VCPU can be paused is 32760 (and 32760 is written to PAUSE<14:0> if the specified value exceeds this number).

The paused VCPU will restart operation when either of the following conditions is true.

- Value of the pause field (PAUSE<14:3>) in the PAUSE register is 0.
- A disrupting exception causes a trap.

Note The VCPU stays paused if the exception is masked and no trap is generated.

When the VCPU restarts operation, the instruction specified by the NPC of the WRPAUSE or PAUSE instruction is executed. If a trap occurs while the VCPU is paused, the WRPAUSE or PAUSE instruction is not treated as the instruction that was disrupted by the trap.

Exception	Condition
<i>illegal_instruction</i>	$i = 0$ and $iw<12:5> \neq 0$
<i>illegal_action</i>	$XAR.v = 1$

7.132. Load Entire Floating-Point State Register

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	op3	rd	Operation	HPC-ACE	Assembly Language Syntax	
				Regs	SIMD	
LDXEFSR	10 0001 ₂	3	Read from memory to FSR	✓	ldx	[<i>address</i>], %efsr

Refer to Section 7.84 in UA2011.

If an LDXEFSR exception generates a precise trap, FSR is not updated.

Exception	Condition
<i>Illegal_instruction</i>	$i = 0$ and $reserved \neq 0$
<i>fp_disabled</i>	$PSTATE.pcf = 0$ or $FPRS.fef = 0$
<i>illegal_action</i>	$XAR.v = 1$ and any of the following are true • $XAR.urs1 \neq 0$ • $XAR.urs2 \neq 0$ • $XAR.urs3<2> \neq 0$ • $XAR.urd \neq 0$ • $XAR.simd = 1$
<i>mem_access_not_aligned</i>	Address not aligned on 8-byte boundary
<i>VA_watchpoint</i>	Refer to 12.5.1.62
<i>DAE_privilege_violation</i>	Refer to 12.5.1.8
<i>DAE_nfo_page</i>	Refer to 12.5.1.7

7.133. Compare and Branch (CBcond)

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Refer to Section 7.17 in UA2011.

Note The Trap on Control Transfer feature is implemented on SPARC64™ X+.

Exception	Condition
<i>illegal_instruction</i>	$c_lo = 000_2$
<i>illegal_action</i>	$XAR.v = 1$
<i>control_transfer_instruction</i>	$PSTATE.tct = 1$ and CBcond causes a transfer of control

7.134. Partitioned Move Selected Floating-Point Register on Floating-Point Register's Condition

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FPSELMOV8X	0 1001 0101 ₂	Select eight 8-bit data from registers	✓	✓	fpselmov8x	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPSELMOV16X	0 1001 0110 ₂	Select four 16-bit data from registers	✓	✓	fpselmov16x	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPSELMOV32X	0 1001 0111 ₂	Select two 32-bit data from registers	✓	✓	fpselmov32x	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd			op3 = 11 0110 ₂	rs1			opf	rs2		
31 30 29	25 24			19 18	14 13			5 4	0		

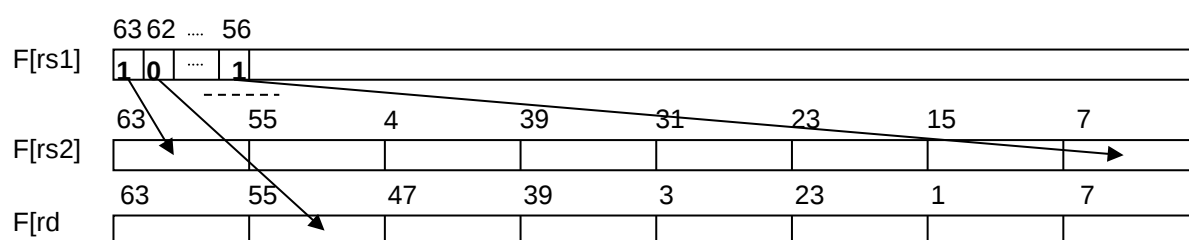
Description The n most significant bits of Fd[rs1] select bit ranges from either Fd[rs2] or Fd[rd]. Selected bit ranges are written in Fd[rd]. If the (63 – n)th bit of Fd[rs1] is 1, the corresponding bit range in Fd[rs2] is selected and written to the same bit range in Fd[rd]. If the bit in Fd[rs1] is 0, the corresponding bit range in Fd[rd] is selected

The bit ranges of .Fd[rs2] and Fd[rd] that are selected by Fd[rs1] are shown below.

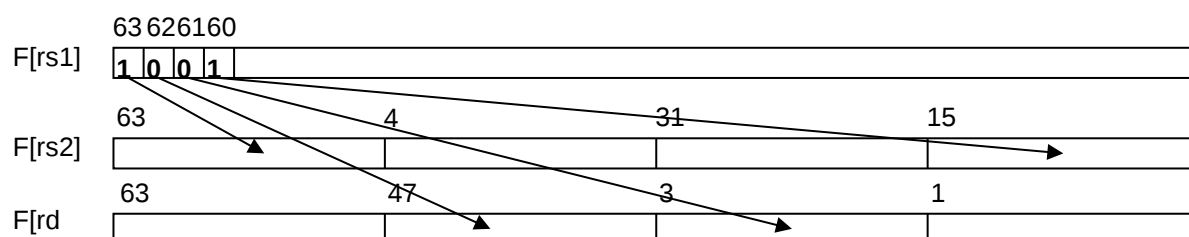
Note Bits Fd[rs1]<55:0> for FPSELMOV8X, Fd[rs1]<59:0> for FPSELMOV16X, and Fd[rs1]<61:0> for FPSELMOV32X are ignored and have no effect.

	F[rs1] bit 63	F[rs1] bit 62	F[rs1] bit 61	F[rs1] bit 60	F[rs1] bit 59	F[rs1] bit 58	F[rs1] bit 57	F[rs1] bit 56
Corresponding bit ranges for FPSELMOV8X	<63:56>	<55:48>	<47:40>	<39:32>	<31:24>	<23:16>	<15:8>	<7:0>
Corresponding bit ranges for FPSELMOV16X	<63:48>	<47:32>	<31:16>	<15:0>				
Corresponding bit ranges for FPSELMOV32X	<63:32>	<31:0>						

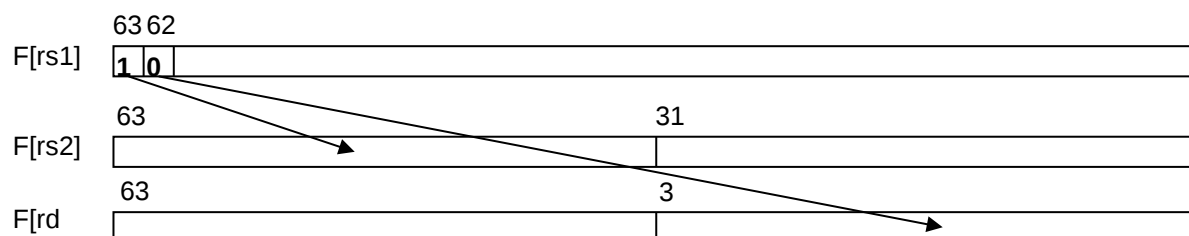
Example of FPSELMOV8X



Example of FPSELMOV16X



Example of FPSELMOV32X



Note The 64-bit FPSELMOV instruction is not defined because its behavior would be the same as FSELMOVd (page 112). However, FSELMOVd updates fields in FSR.

FPSELMOV{8|16|32}X do not update any fields in FSR.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.135. 64-bit Integer Compare on Floaing-Point Register

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur).

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs.	SIMD	
FPCMP64X	1 0000 0100 ₂	Compare signed 64-bit integers	✓		fpcmp64x %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>
FPCMPU64X	1 0000 0101 ₂	Compare unsigned 64-bit integers	✓		fpcmpu64x %fccn, <i>freg_{rs1}</i> , <i>freg_{rs2}</i>

10 ₂		—		cc1	cc0	op3 = 11 0110 ₂				rs1		opf				rs2						
31	30	29		27	26	25	24			19	18			14	13			5	4			0

cc1	cc0	Condition code
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

Description Compare the 64-bit integer values in the floating-point registers Fd[rs1] and Fd[rs2] and stores the result in the floating-point condition code field FSR.fccn specified by the instruction.

Comparison result	Value of %fccn
F[rs1] = F[rs2]	0
F[rs1] < F[rs2]	1
F[rs1] > F[rs2]	2
	3 N/A

Programming Note FPCMP{64|U64}X is not an FPop. FSR.cexc and FSR.aexc are not updated, and *fp_exception_other* exceptions do not occur.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>Illegal_instruction</i>	iw<29:27> ≠ 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd ≠ 0 • XAR.simd = 1

7.136. 64-bit Integer Shift on Floating-Point Register

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur).

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FPSLL64X	1 0000 0110 ₂	Shift left logical 64-bit integer	✓	✓	fpsll64x	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPSRL64X	1 0000 0111 ₂	Shift right logical 64-bit integer	✓	✓	fpsrl64x	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPSRA64X	1 0000 1111 ₂	Shift right arithmetic 64-bit integer	✓	✓	fpsra64x	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0110 ₂	rs1	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description These instructions shift the data in Fd[rs1] right or left and store the result in Fd[rd]. The shift count is specified by the lowest 6 bits of Fd[rs2].

FPSLL64X shifts all 64 bits of Fd[rs1] left, replacing the vacated positions on the right with 0, and stores the result in Fd[rd].

FPSRL64X shifts all 64 bits of Fd[rs1] right, replacing the vacated positions on the left with 0, and stores the result in Fd[rd].

FPSRA64X shifts all 64 bits of Fd[rs1] right, replacing the vacated positions on the left with the MSB of Fd[rs1], and stores the result in Fd[rd].

FP{SLL | SRL | SRA}64X do not update any fields in FSR.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.137. Store Floating-Point Register on Register Condition (Extension of SPARC64™ X+)

Compatibility Note For the specification of this instruction on SPARC64™ X, refer to page 174.

Instruction	op3	rs2, rd	Operation	HPC-ACE		Assembly Language Syntax	
				Regs	SIMD		
STFR	10 1100 ₂	0 – 31	Store single-precision floating-point register on register condition (XAR.v = 0)			stfr	<i>freg_{rd}</i> , <i>freg_{rs2}</i> , [<i>regrs1</i>]
STFR	10 1100 ₂	0 – 126, 256 – 382 ^{xxx}	Store single-precision floating-point register on register condition (XAR.v = 1)	✓	✓	stfr	<i>freg_{rd}</i> , <i>freg_{rs2}</i> , [<i>regrs1</i>]
STDFR	10 1111 ₂	0 – 126, 256 – 382 ^{xxx}	Store double-precision floating-point register on register condition	✓	✓	stdfr	<i>freg_{rd}</i> , <i>freg_{rs2}</i> , [<i>regrs1</i>]

11 ₂	rd		op3		rs1		i = 1	—		rs2	
31 30 29	25 24		19 18		14 13 12			5 4		0	

non-SIMD execution

When XAR.v = 0 and the MSB (bit 31) of Fs[rs2] is 1, STFR writes the 4 bytes of the single-precision register Fs[rd] to the specified address, which should be aligned on a 4-byte boundary. When XAR.v = 1, XAR.simd = 0, and the MSB (bit 63) of F[rs2] is 1, STFR writes the upper 4 bytes of the double-precision register Fd[rd] to the specified address, which should be aligned on a 4-byte boundary.

When the MSB (bit 63) of Fd[rs2] is 1, STDFR writes the 8 bytes of the double-precision register Fd[rd] to the specified address, which should be aligned on a 4-byte boundary.

These floating-point store instructions use implicit ASIs (refer to Section 6.3.1.3 in UA2011) to access memory. The effective address is “R[rs1]”.

STFR and STDFR cause a *mem_address_not_aligned* exception when the address is not aligned on a word boundary.

When executing a non-SIMD STDFR, the address needs to be aligned on a word boundary. However, if the address is aligned on a word boundary but is not aligned on a doubleword boundary, a *STDF_mem_address_not_aligned* exception will occur. The trap handler must emulate the STDFR instruction when this exception occurs.

STFR does not cause any exceptions except *illegal_instruction*, *fp_disabled*, and *illegal_action* when XAR.v = 1, XAR.simd = 0, and the MSB (bit 63) of Fd[rs2] is 0; or when XAR.v = 0 and the MSB (bit 31) of Fs[rs2] is 0. STDFR does not cause any exceptions except *illegal_instruction*, *fp_disabled*, and *illegal_action* when the MSB (bit 63) of Fd[rs2] is 0.

^{xxx} 5.3.1 Encoding which is defined in “Floating-Point Register Number Encoding”(page 26)

Exceptions that are always detected	Exceptions that are detected only when MSB of Fs[rs2] or MSB of Fd[rs2] is 1
<i>illegal_instruction</i> <i>fp_disabled</i> <i>illegal_action</i>	<i>mem_address_not_aligned</i> <i>STDF_mem_address_not_aligned</i> <i>VA_watchpoint</i> <i>DAE_privilege_violation</i> <i>DAE_nfo_page</i>

SIMD execution STFR and STDFR support SIMD execution on SPARC64 X+. SIMD STFR and SIMD STDFR simultaneously execute basic and extended stores for single-precision and double-precision data, respectively. Refer to Section 5.5.15 (page 35) for details on how to specify the registers.

A SIMD STFR writes the upper 4 bytes of Fd[rd] to the lower 4 bytes of the address when XAR.v = 1, XAR.simd = 1, and the MSB (bit 63) of Fd[rs2] is 1, and writes the upper 4 bytes of Fd[rd + 256] to the upper 4 bytes of the address when XAR.v = 1, XAR.simd = 1, and the MSB (bit 63) of Fd[rs2+256] is 1. The address must be aligned on an 8-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

SIMD STDFR writes Fd[rd] to the lower 8 bytes of the address when XAR.v = 1, XAR.simd = 1, and the MSB (bit 63) of Fd[rs2] is 1, and writes Fd[rd + 256] to the upper 8 bytes of the address when XAR.v = 1, XAR.simd = 1, and the MSB (bit 63) of Fd[rs2+256] is 1. The address must be aligned on a 16-byte boundary. Misaligned accesses cause a *mem_address_not_aligned* exception.

These floating-point store instructions use implicit ASIs (refer to Section 6.3.1.3 in UA2011) to access memory.

Note A SIMD STDFR does not cause a *STDF_mem_address_not_aligned* exception when the address is aligned on a word boundary but is not aligned on a doubleword boundary.

SIMD STFR and SIMD STDFR can only be used to access cacheable address spaces. An attempt to access a non-cacheable address space causes a *DAE_nc_page* exception.

Like non-SIMD store instructions, memory access semantics adhere to TSO. SIMD STFR and SIMD STDFR simultaneously execute basic and extended stores; however, the ordering between the basic and extended stores conforms to TSO.

SIMD STFR and SIMD STDFR always detect an *illegal_instruction*, *fp_disabled*, or *illegal_action* exception.

SIMD STFR and SIMD STDFR always detect *mem_address_not_aligned* or *VA_watchpoint* exceptions for both the basic and extended operations when the exception condition is detected and either of the following conditions is true.

1. Either MSB (bit 63) of basic register Fd[rs2] or extended register Fd[rs2+256] is 1.
2. Both MSBs (bit63) of Fd[rs2] and Fd[rs2+256] are 1.

SIMD STFR and SIMD STDFR detect an exception only for the corresponding basic or extended operation when the exception condition is detected (excluding *illegal_instruction*, *fp_disabled*, *illegal_action*, *mem_address_not_aligned*, *VA_watchpoint*) and the MSB (bit63) of basic register Fd[rs2] or the MSB (bit 63) of extended register Fd[rs2+256] is 1. The exception is detected for both operations only if both the MSBs of Fd[rs2] and Fd[rs2+256] are 1.

Exceptions that are always detected	Exceptions that are detected for both operations when either MSB in Fd[rs2] or Fd[rs2+256] is 1	Exceptions that are detected for the corresponding operation(s) when the MSB in Fd[rs2] or Fd[rs2+256] is 1
<i>illegal_instruction</i> <i>fp_disabled</i> <i>illegal_action</i>	<i>mem_address_not_aligned</i> <i>VA_watchpoint</i>	<i>DAE_privilege_violation</i> <i>DAE_nc_page</i> <i>DAE_nfo_page</i>

Exceptions that are detected for both operations when either MSB in Fd[rs2] or Fd[rs2+256] is 1	Detected address
<i>mem_address_not_aligned</i>	Address of basic operation (always)
<i>VA_watchpoint</i>	The detected address. When detected for both operations, address of the basic operation

Exceptions that are detected for the corresponding operation(s) when the MSB in Fd[rs2] or Fd[rs2+256] is 1	Detected address
<i>DAE_privilege_violation</i>	Address of basic operation (always)
<i>DAE_nc_page</i>	Address of basic operation (always)
<i>DAE_nfo_page</i>	Address of basic operation (always)

Exception	Target instruction	Condition
<i>illegal_instruction</i>	all	$i = 0$ or the <i>reserved</i> field is not 0
<i>fp_disabled</i>	all	$PSTATE.pef = 0$ or $FPRS.fef = 0$
<i>illegal_action</i>	all	$XAR.v = 1$ and any of the following are true $XAR.urs1 \neq 0$ $XAR.urs2<1> \neq 0$ $XAR.urs3<2> \neq 0$ $XAR.urd<1> \neq 0$ $XAR.simd = 1$ and $XAR.urs2<2> \neq 0$ $XAR.simd = 1$ and $XAR.urd<2> \neq 0$
<i>STDF_mem_address_not_aligned</i>	STDFR	MSB of $Fd[rs2]$ is 1 and address aligned on a word boundary but not a doubleword boundary when $XAR.v = 1$ and $XAR.simd = 0$, or $XAR.v = 0$
<i>mem_address_not_aligned</i>	STFR	Either of the following conditions is true - Address not aligned on word boundary when $XAR.v = 0$ and MSB of $Fs[rs2]$ is 1 - Address not aligned on word boundary when $XAR.v = 1$, $XAR.simd = 0$, and MSB of $Fd[rs2]$ is 1 - Address not aligned on a doubleword boundary when MSB of $Fd[rs2]$ or $Fd[rs2+256]$ is 1, $XAR.v = 1$ and $XAR.simd = 1$
	STDFR	Either of the following conditions is true - Address not aligned on a word boundary when MSB of $Fd[rs2]$ is 1 and $XAR.v = 1$ and $XAR.simd = 0$, or $XAR.v = 0$ - Address not aligned on a quadword boundary when MSB of $Fd[rs2]$ or $Fd[rs2+256]$ is 1 and $XAR.v = 1$ and $XAR.simd = 1$
<i>VA_watchpoint</i>	all	Refer to the description and 12.5.1.62
<i>DAE_privilege_violation</i>	all	Refer to the description and 12.5.1.8
<i>DAE_nc_page</i>	all	Access to non-cacheable space when $XAR.v = 1$, $XAR.simd = 1$, and MSB of $Fd[rs2]$ or $Fd[rs2+256]$ is 1
<i>DAE_nfo_page</i>	all	Refer to the description and 12.5.1.7

7.138. Shift Mask Or (Extension of SPARC64™ X+)

Compatibility Note For the specification of this instruction on SPARC64™ X, refer to page 222.

Instruction	var	size	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FSHIFTORX	10 ₂	11 ₂	Concatenate the values of two double-precision floating-point registers	✓	✓	fshiftorx <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rs3}</i> , <i>freg_{rd}</i>

10 ₂	rd	op3 = 11 0111 ₂	rs1	rs3	var = 10 ₂	size = 11 ₂	rs2
31 30 29	25 24	19 18	14 13	9 8	7 6	5 4	0

Non SIMD execution

Depending on the setting in Fd[rs3], FSHIFTORX performs one of the following sets of operations. In pattern A) two fields are used, and in pattern B) three fields are used.

- FSHIFTORX shifts the value of Fd[rs1] right or left and extracts part of the result, as specified by 1st_*. It also shifts the value of Fd[rs2] right or left and extracts part of the result, as specified by 2nd_*. The two extracted values are bitwise ORed, and this result is written in Fd[rd].
- FSHIFTORX shifts the value of Fd[rs1] right or left and extracts part of the result, as specified by 1st_*. It again shifts the value of Fd[rs1] right or left and extracts part of the result, this time as specified by 2nd_*. The two extracted values are bitwise ORed. Then a logical operation (AND, OR, or XOR) is performed with this result and the value of Fd[rs2] as the inputs. The final result is written into Fd[rd].

The exceptions detected for this instruction depend on the value of XASR.fed. An *illegal_instruction* exception occurs when setting the following values to Fd[rs3] and XASR.fed = 0.

- 1) Set 1 to a *reserved* field
- 2) Set 1 to set_2nd_default and set values other than 0 to <31:0>

When XASR.fed = 1, an *illegal_instruction* exception will not occur even if the above values are specified in Fd[rs3]. However, the result written in Fd[rd] is not guaranteed to be valid when such values are set.

Table 7-54 Meanings of bits in Fd[rs3]

set_2nd_default	operation	—	1st_mask_inv	1st_mask_offset	1st_mask_length	1st_shift_amount
63	62 61 60 57	56	55	48 47	40 39	32
—	—	2nd_mask_inv	2nd_mask_offset	2nd_mask_length	2nd_shift_amount	—
31	25	24 23	16 15	8 7	0	—

Bit	Field	Description
-----	-------	-------------

rs1	rs2		
63	—	set_2nd_default	Specifies set of {1st 2nd}_* fields to use for the 2nd field 0: Use the rs2_* fields for the 2nd field 1: Use values derived from 1st_* fields
62:61	—	operation	Specifies type of operation. Refer to Table 7-55
56	24	{1st 2nd}_mask_inv	Specifies whether to invert the mask 0: Do not invert 1: Invert
55:48	23:16	{1st 2nd}_mask_offset	Starting position in mask (number of bits from MSB) 8-bit signed integer
47:40	15:8	{1st 2nd}_mask_length	Mask bit length, 8-bit signed integer
39:32	7:0	{1st 2nd}_shift_amount	Shift count (positive: left shift, negative: right shift), 8-bit signed integer

Table 7-55 Operation Patterns

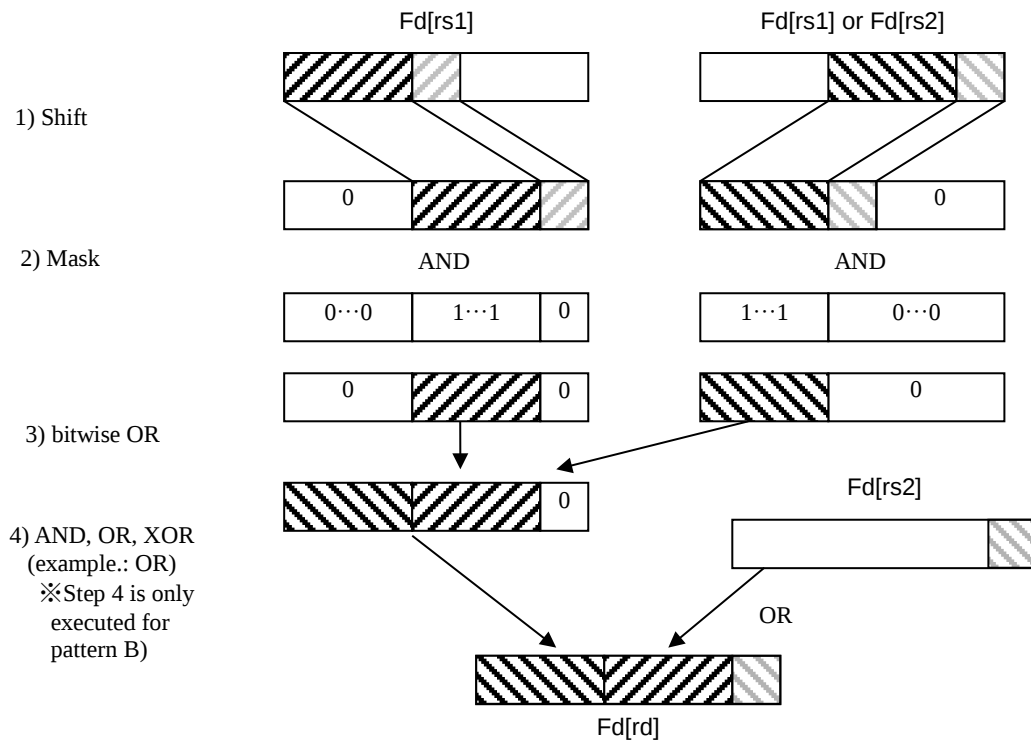
operation	1st	2nd field	3rd field	Logical operation of 3rd field and value generated from 1st and 2nd fields
00 ₂	rs1	rs2	—	—
01 ₂	rs1	rs1	rs2	AND
10 ₂	rs1	rs1	rs2	OR
11 ₂	rs1	rs1	rs2	XOR

Note FSHIFTORX detects an *illegal_instruction* exception for certain values in the register Fd[rs3]. An *fp_disabled* exception or *illegal_action* exception may be detected before the register is read. That is, for this instruction, these exceptions may have higher priority than an *illegal_instruction* exception.

The operation fields select one of two patterns for FSHIFTORX, which are divided into the operations shown below.

A: 1) Shift, 2) Mask, 3) OR

B: 1) Shift, 2) Mask, 3) OR, 4) Logical operation



The shift operation executes a logical shift. If shifting left, the vacated positions on the right are replaced by 0. If shifting right, the vacated positions on the left are replaced by 0. The {1st|2nd}_shift_amount field specifies the shift direction and shift count. The shift is leftwards if the value is positive and rightwards if the value is negative.

The mask operation extracts the specified bits from the shifted register. Two mask patterns are defined by {1st|2nd}_mask_offset, {1st|2nd}_mask_length, and {1st|2nd}_mask_inv. A mask pattern is generated from one set of these fields and bitwise ANDed with the corresponding shifted register.

A mask pattern is a generated range of contiguous 1s in a 64-bit doubleword, where no bits are 1 outside this range. The inverse of this pattern can also be specified. The {1st|2nd}_mask_offset field specifies the starting position of the mask (the range of 1s) as the number of bits from the left (MSB). The {1st|2nd}_mask_length field specifies the length of the range of bits that are set to 1. To invert this mask, specify 1 for {1st|2nd}_mask_inv.

Note The mask pattern contains bits that are 1 when $0 \leq \{1st|2nd\}_mask_offset < 64$ and $0 < \{1st|2nd\}_mask_length$. That is, the pattern fits inside a doubleword. The specified pattern can exceed the length of a doubleword if $64 \leq \{1st|2nd\}_mask_offset + \{1st|2nd\}_mask_length$. In this case, the mask pattern is a contiguous range of 1s from {1st|2nd}_mask_offset to the LSB.

Note The mask pattern may contain bits that are 1 even if the value of {1st|2nd}_mask_offset is negative. Specifically, when $|\{1st|2nd\}_mask_offset| < \{1st|2nd\}_mask_length$, the mask pattern is a contiguous range of 1s from the MSB to “{1st|2nd}_mask_length - |{1st|2nd}_mask_offset|”.

Note The 64-bit mask pattern is all 0s when {1st|2nd}_mask_offset and {1st|2nd}_mask_length satisfy any of the conditions below. The mask pattern is all 1s under the same conditions when {1st|2nd}_mask_inv is used to invert the mask pattern.

- {1st|2nd}_mask_length ≤ 0
- {1st|2nd}_mask_offset + rs{1|2}_mask_length ≤ 0
- {1st|2nd}_mask_offset ≥ 64

In pattern A), the bitwise OR operation is executed on the values obtained from Fd[rs1] and Fd[rs2] using the shift and mask operations. The result is written in Fd[rd]. In pattern B), the bitwise OR operation is executed on the two different values generated from Fd[rs1] using the shift and mask operations. A logical operation (in the example, an OR) is performed with this result and Fd[rs2] as the inputs. The final result is written in Fd[rd].

Table 7-56 shows an example of how to specify the bits in Fd[rs3]. In this example,

1. The lower 32 bits of Fd[rs1] (Fd[rs1]<31:0>) and the upper 24 bits of Fd[rs1] (Fd[rs1]<63:40>) are concatenated in the given order (Fd[rs1]<31:0>::Fd[rs1]<63:40>).
2. A bitwise OR of the concatenated value and Fd[rs2] is executed.
3. The final result is written in Fd[rd].

Table 7-56 Example of how to specify bits in Fd[rs3]

Field	Concatenate lower 32 bits of Fd[rs1] and upper 24 bits of Fd[rs1] (Fd[rs1]<31:0>::Fd[rs1]<63:40>), then OR the result and Fd[rs2]
set_2nd_default	0 (2nd_* field are used)
operation	01 (OR of 3rd field and value generated from 1st and 2nd fields)
1st_mask_inv	0 (Do not invert 1st mask)
1st_mask_offset	32 (Mask shifted 1st field starting from bit<31>)
1st_mask_length	24 (Mask length is 24 bits)
1st_shift_amount	-32 (Shift right 32 bits)
2nd_mask_inv	0 (Do not invert the 2nd mask)
2nd_mask_offset	0 (Mask shifted 2nd field starting from MSB)
2nd_mask_length	32 (Mask length is 32 bits)
2nd_shift_amount	32 (Shift left 32 bits)

SIMD execution When FSHIFTORX is executed as a SIMD instruction, any floating-point register Fd[0] – Fd[126], Fd[256] – Fd[382] can be specified for Fd[rs1]. If an extended register Fd[256] – Fd[382] is specified for Fd[rs1], the extended register Fd[n] is used for the basic operation and the basic register Fd[n – 256] is used for the extended operation. On the other hand, only basic registers Fd[0] – Fd[126] can be specified for Fd[rs2], Fd[rs3], and Fd[rd]. The basic operation uses the basic register Fd[n], and the extended operation uses the extended register Fd[n+256].

FSHIFTORX does not update any fields in FSR.

Exception	Taget instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<1> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urs3<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
<i>illegal_instruction</i>	All	Refer to the description.

7.139. SIMD Compare (Extension of SPARC64™ X+)

Compatibility Note For the specification of this instruction on SPARC64™ X, refer to page 226.

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FPCMPLE16X	0 1100 0000 ₂	Compare four 16-bit signed integers If $src1 \leq src2$ then 1	✓	✓	fpcmp16x <i>freg_{rd}</i> (fcmp16x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPULE16X	0 1100 0001 ₂	Compare four 16-bit unsigned integers If $src1 \leq src2$ then 1	✓	✓	fpcmpule16x <i>freg_{rd}</i> (fucmp16x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPUNE16X	0 1100 0011 ₂	Compare four 16-bit unsigned integers If $src1 \neq src2$ then 1	✓	✓	fpcmpune16x <i>freg_{rd}</i> (fucmpne16x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPLE32X	0 1100 0100 ₂	Compare two 32-bit signed integers If $src1 \leq src2$ then 1	✓	✓	fpcmp32x <i>freg_{rd}</i> (fcmp32x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPULE32X	0 1100 0101 ₂	Compare two 32-bit unsigned integers If $src1 \leq src2$ then 1	✓	✓	fpcmpule32x <i>freg_{rd}</i> (fucmp32x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPUNE32X	0 1100 0111 ₂	Compare two 32-bit unsigned integers If $src1 \neq src2$ then 1	✓	✓	fpcmpune32x <i>freg_{rd}</i> (fucmpne32x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPGT16X	0 1100 1000 ₂	Compare four 16-bit signed integers If $src1 > src2$ then 1	✓	✓	fpcmpgt16x <i>freg_{rd}</i> (fcmpgt16x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPUGT16X	0 1100 1001 ₂	Compare four 16-bit unsigned integers If $src1 > src2$ then 1	✓	✓	fpcmpugt16x <i>freg_{rd}</i> (fucmpgt16x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPUEQ16X	0 1100 1011 ₂	Compare four 16-bit unsigned integers If $src1 = src2$ then 1	✓	✓	fpcmpueq16x <i>freg_{rd}</i> (fucmpeq16x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPGT32X	0 1100 1100 ₂	Compare two 32-bit signed integers If $src1 > src2$ then 1	✓	✓	fpcmpgt32x <i>freg_{rd}</i> (fcmpgt32x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPUGT32X	0 1100 1101 ₂	Compare two 32-bit unsigned integers If $src1 > src2$ then 1	✓	✓	fpcmpugt32x <i>freg_{rd}</i> (fucmpgt32x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPUEQ32X	0 1100 1111 ₂	Compare two 32-bit unsigned integers If $src1 = src2$ then 1	✓	✓	fpcmpueq32x <i>freg_{rd}</i> (fucmpeq32x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,
FPCMPLE8X	0 1101 0000 ₂	Compare eight 8-bit signed integers If $src1 \leq src2$ then 1	✓	✓	fpcmp8x <i>freg_{rd}</i> (fcmp8x) [†]	<i>freg_{rs1}</i> , <i>freg_{rs2}</i> ,

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
FPCMPULE8X	0 1101 0001 ₂	Compare eight 8-bit unsigned integers If $src1 \leq src2$ then 1	✓	✓	fpcmpule8x $freg_{rd}$ (fucmple8x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPUNE8X	0 1101 0011 ₂	Compare eight 8-bit unsigned integers If $src1 \neq src2$ then 1	✓	✓	fpcmpune8x $freg_{rd}$ (fucmpne8x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPLE64X	0 1101 0100 ₂	Compare 64-bit signed integers If $src1 \leq src2$ then 1	✓	✓	fpcmple64x $freg_{rd}$ (fcmpl64x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPULE64X	0 1101 0101 ₂	Compare 64-bit unsigned integers If $src1 \leq src2$ then 1	✓	✓	fpcmpule64x $freg_{rd}$ (fucmpl64x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPUNE64X	0 1101 0111 ₂	Compare 64-bit unsigned integers If $src1 \neq src2$ then 1	✓	✓	fpcmpune64x $freg_{rd}$ (fucmpn64x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPGT8X	0 1101 1000 ₂	Compare eight 8-bit signed integers If $src1 > src2$ then 1	✓	✓	fpcmpgt8x $freg_{rd}$ (fcmpgt8x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPUGT8X	0 1101 1001 ₂	Compare eight 8-bit unsigned integers If $src1 > src2$ then 1	✓	✓	fpcmpugt8x $freg_{rd}$ (fucmpgt8x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPUEQ8X	0 1101 1011 ₂	Compare eight 8-bit unsigned integers If $src1 = src2$ then 1	✓	✓	fpcmpueq8x $freg_{rd}$ (fucmpeq8x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPGT64X	0 1101 1100 ₂	Compare 64-bit signed integer If $src1 > src2$ then 1	✓	✓	fpcmpgt64x $freg_{rd}$ (fcmpgt64x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPUGT64X	0 1101 1101 ₂	Compare 64-bit unsigned integer If $src1 > src2$ then 1	✓	✓	fpcmpugt64x $freg_{rd}$ (fucmpgt64x) [†]	$freg_{rs1}, freg_{rs2}$
FPCMPUEQ64X	0 1101 1111 ₂	Compare 64-bit unsigned integer If $src1 = src2$ then 1	✓	✓	fpcmpueq64x $freg_{rd}$ (fucmpeq64x) [†]	$freg_{rs1}, freg_{rs2}$

[†] the older mnemonic for this instruction (still recognized by the assembler)

10 ₂	rd	op3 = 11 0110 ₂	rs1	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description These instructions compare the several elements (partitions) in the two floating-point registers Fd[rs1] and Fd[rs2]. The result is written in the floating-point register Fd[rd]. The comparison results for these elements are written in the most-significant bits of Fd[rd]. 0s are written in the other bits.

The number of elements in a 64-bit input register depends on the data type of the comparison. The number of elements and their bit ranges for each data type are shown in Table 7-57.

Table 7-57 Number of elements and their bit ranges for each data type

Data type	Number of elements	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6	Element 7	Element 8
8-bit signed integer	8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
8-bit unsigned integer	8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
16-bit signed integer	4	63:48	47:32	31:16	15:0	—	—	—	—
16-bit unsigned integer	4	63:48	47:32	31:16	15:0	—	—	—	—
32-bit signed integer	2	63:32	31:0	—	—	—	—	—	—
32-bit unsigned integer	2	63:32	31:0	—	—	—	—	—	—
64-bit signed integer	1	63:0	—	—	—	—	—	—	—
64-bit unsigned integer	1	63:0	—	—	—	—	—	—	—

Elements of Fd[rs1] and Fd[rs2] which occupy the same bit range are compared. The result is written in the corresponding bit of Fd[rd]. The bit positions of Fd[rd] corresponding to each element are shown in Table 7-58.

Table 7-58 Elements and corresponding bit positions in Fd[rd]

	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6	Element 7	Element 8
Fd[rd]	63	62	61	60	59	58	57	56

FPCMPLE{8X, 16X, 32X, 64X} compare the elements of Fd[rs1] and Fd[rs2] as signed integers. If “elements of Fd[rs1]” ≤ “elements of Fd[rs2]”, the corresponding bits of Fd[rd] is set to 1.

FPCMPGT{8X, 16X, 32X, 64X} compare the elements of Fd[rs1] and Fd[rs2] as signed integers. If “elements of Fd[rs1]” > “elements of Fd[rs2]”, the corresponding bits of Fd[rd] is set to 1.

FPCMPULE{8X, 16X, 32X, 64X} compare the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “elements of Fd[rs1]” ≤ “elements of Fd[rs2]”, the corresponding bits of Fd[rd] is set to 1.

FPCMPUNE{8X, 16X, 32X, 64X} compare the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “elements of Fd[rs1]” ≠ “elements of Fd[rs2]”, the corresponding bits of Fd[rd] is set to 1.

FPCMPUGT{8X, 16X, 32X, 64X} compare the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “elements of Fd[rs1]” > “elements of Fd[rs2]”, the corresponding bits of Fd[rd] is set to 1.

FPCMPUEQ{8X, 16X, 32X, 64X} compare the elements of Fd[rs1] and Fd[rs2] as unsigned integers. If “elements of Fd[rs1]” = “elements of Fd[rs2]”, the corresponding bits of Fd[rd] is set to 1.

Note Instructions that compare whether signed integers are equal or not equal are not defined. These comparisons are equivalent to the instructions `FPCMPUEQ{8X, 16X, 32X, 64X}` and `FPCMPUNE{8X, 16X, 32X, 64X}`, which respectively compare whether unsigned integers are equal or not equal.

Compatibility Note Differences from SPARC64™ X are the following: extended floating-point registers can be specified for non SIMD instructions, HPC-ACE SIMD execution is supported, and the instruction mnemonic is changed (`FCMP*/FUCMP*`→`FPCMP*/FPCMPU*`).

SIMD Compare does not update any fields in FSR.

Exception	Taget instruction	Condition
<i>fp_disabled</i>	All	<code>PSTATE.pef = 0</code> or <code>FPRS.fef = 0</code>
<i>illegal_action</i>	All	<code>XAR.v = 1</code> and any of the following are true <code>XAR.urs1<1> ≠ 0</code> <code>XAR.urs2<1> ≠ 0</code> <code>XAR.urs3 ≠ 0</code> <code>XAR.urd<1> ≠ 0</code> <code>XAR.simd = 1</code> and <code>XAR.urs1<2> ≠ 0</code> <code>XAR.simd = 1</code> and <code>XAR.urs2<2> ≠ 0</code> <code>XAR.simd = 1</code> and <code>XAR.urd<2> ≠ 0</code>

7.140. Fixed-Point Partitioned Add (128-bit)

Compatibility Note SPARC64™ X does not support this instruction. (An *illegal_instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs.	SIMD		
FPADD128XHI	0 1001 1111 ₂	128-bit add	✓	✓	fpadd128xhi	freg _{rs1} , freg _{rs2} , freg _{rd}

10 ₂	rd			op3 = 11 0110 ₂			rs1		opf			rs2	
31 30 29	25 24			19 18			14 13		5 4			0	

Description FPADD128XHI adds a 16-byte unsigned integer, where the upper 8 bytes are in Fd[rs1] and the lower 8-bytes are in Fd[rs2], to the 8-byte unsigned integer in Fd[rd]. The upper 8 bytes of the result are written in Fd[rd].

FPADD128XHI does not update any fields in FSR.

Exception	Condition
<i>fp_disabled</i>	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.141. Integer Minimum and Maximum

Compatibility Note SPARC64™ X does not support this instruction.
(*Anillegal_instruction* exception will occur.)

Instruction	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FPMAX64x	0 1110 1100 ₂	Max. of signed 64-bit integer	✓	✓	fpmax64x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMAXu64x	0 1110 1101 ₂	Max. of unsigned 64-bit integer	✓	✓	fpmaxu64x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMIN64x	0 1110 1110 ₂	Min. of signed 64-bit integer	✓	✓	fpmmin64x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMINu64x	0 1110 1111 ₂	Min. of unsigned 64-bit integer	✓	✓	fpmminu64x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMAX32x	0 1110 0100 ₂	Max. of signed 32-bit integer	✓	✓	fpmmax32x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMAXu32x	0 1110 0101 ₂	Max. of unsigned 32-bit integer	✓	✓	fpmmaxu32x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMIN32x	0 1110 0110 ₂	Min. of signed 32-bit integer	✓	✓	fpmmin32x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>
FPMINu32x	0 1110 0111 ₂	Min. of unsigned 32-bit integer	✓	✓	fpmminu32x <i>freg_{rs1}</i> , <i>freg_{rs2}</i> , <i>freg_{rd}</i>

10 ₂	rd		op3 = 11 0110 ₂		rs1		opf		rs2	
31 30 29			25 24		19 18		14 13		5 4	0

Description

FPMAX64x compares Fd[rs1] and Fd[rs2] as signed 64-bit integers. If Fd[rs1] > Fd[rs2], then Fd[rs1] is written in Fd[rd]. Otherwise, Fd[rs2] is written in Fd[rd].

FPMAXu64x compares Fd[rs1] and Fd[rs2] as unsigned 64-bit integers. If Fd[rs1] > Fd[rs2], then Fd[rs1] is written in Fd[rd]. Otherwise, Fd[rs2] is written in Fd[rd].

FPMIN64x compares Fd[rs1] and Fd[rs2] as signed 64-bit integers. If Fd[rs1] < Fd[rs2], then Fd[rs1] is written in Fd[rd]. Otherwise, Fd[rs2] is written in Fd[rd].

FPMINu64x compares Fd[rs1] and Fd[rs2] as unsigned 64-bit integers. If Fd[rs1] < Fd[rs2], then Fd[rs1] is written in Fd[rd]. Otherwise, Fd[rs2] is written in Fd[rd].

FPMAX32x compares Fd[rs1]<63:32> and Fd[rs2]<63:32> as signed 32-bit integers. If Fd[rs1]<63:32> > Fd[rs2]<63:32>, then Fd[rs1]<63:32> is written in Fd[rd]<63:32>. Otherwise, Fd[rs2]<63:32> is written in Fd[rd]<63:32>. At the same time, Fd[rs1]<31:0> and Fd[rs2]<31:0> are compared as signed 32-bit integers. If Fd[rs1]<31:0> > Fd[rs2]<31:0>, then Fd[rs1]<31:0> is written in Fd[rd]. Otherwise, Fd[rs2]<31:0> is written in Fd[rd]<31:0>.

FPMAXu32x compares Fd[rs1]<63:32> and Fd[rs2]<63:32> as unsigned 32-bit integers. If Fd[rs1]<63:32> > Fd[rs2]<63:32>, then Fd[rs1]<63:32> is written in Fd[rd]<63:32>. Otherwise, Fd[rs2]<63:32> is written in Fd[rd]<63:32>. At the same time, Fd[rs1]<31:0> and Fd[rs2]<31:0> are compared as unsigned 32-bit integers. If Fd[rs1]<31:0> > Fd[rs2]<31:0>, then Fd[rs1]<31:0> is written in Fd[rd]<31:0>. Otherwise, Fd[rs2]<31:0> is written in Fd[rd]<31:0>.

FPMIN32x compares Fd[rs1]<63:32> and Fd[rs2]<63:32> as signed 32-bit integers. If Fd[rs1]<63:32> < Fd[rs2]<63:32>, then Fd[rs1]<63:32> is written in Fd[rd]<63:32>. Otherwise, Fd[rs2]<63:32> is written in Fd[rd]<63:32>. At the same time, Fd[rs1]<31:0> and Fd[rs2]<31:0> are compared as signed 32-bit integers. If Fd[rs1]<31:0> < Fd[rs2]<31:0>, then Fd[rs1]<31:0> is written in Fd[rd]<31:0>. Otherwise, Fd[rs2]<31:0> is written in Fd[rd]<31:0>.

FPMINu32x compares Fd[rs1]<63:32> and Fd[rs2]<63:32> as unsigned 32-bit integers. If Fd[rs1]<63:32> < Fd[rs2]<63:32>, then Fd[rs1]<63:32> is written in Fd[rd]<63:32>. Otherwise, Fd[rs2]<63:32> is written in Fd[rd]<63:32>. At the same time, Fd[rs1]<31:0> and Fd[rs2]<31:0> are compared as unsigned 32-bit integers. If Fd[rs1]<31:0> < Fd[rs2]<31:0>, then Fd[rs1]<31:0> is written in Fd[rd]<31:0>. Otherwise, Fd[rs2]<31:0> is written in Fd[rd]<31:0>.

then Fd[rs1]<31:0> is written in Fd[rd]<31:0>. Otherwise, Fd[rs2]<31:0> is written in Fd[rd]<31:0>.

FPMAX{64x|u64x|32x|u32x} and FMIN{64x|u64x|32x|u32x} do not update any fields in FSR.

Exception	Taget instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.142. Move Integer Register to Floating-Point Register (for SPARC64™ X+)

Compatibility Note SPARC64™ X does not support this instruction (An *illegal_instruction* exception will occur).

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
MOVwTOS	1 0001 1001 ₂	Copy lower 32 bits of integer register to single precision register	✓		movwtos	<i>reg_{rs2}, freg_{rd}</i>
MOVxTOD	1 0001 1000 ₂	Copy 64 bits of integer register to double precision register	✓		movxtod	<i>reg_{rs2}, freg_{rd}</i>

10 ₂	rd	op3 = 11 0110 ₂	—	opf	rs2
31 30 29	25 24	19 18	14 13	5 4	0

Description The MOVwTOS instruction copies the lower 32 bits from the general-purpose register R[rs2] to the floating-point register Fd[rd]. No conversion is performed on the copied bits.

The MOVxTOD instruction copies 64 bits from the general-purpose register R[rs2] to the floating-point register Fd[rd]. No conversion is performed on the copied bits.

MOVwTOS and MOVxTOD do not update any fields in FSR.

Exception	Target instruction	Condition
<i>fp_disabled</i>	All	PSTATE.pef = 0 or FPRS.fef = 0
<i>illegal_instruction</i>	All	iw<18:14> ≠ 0
<i>illegal_action</i>	All	XAR.v = 1 and any of the following are true • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1

8. IEEE Std. 754-1985 Requirements for SPARC-V9

8.1. Nonstandard Floating-Point Mode

This section describes the behavior of SPARC64™ X / SPARC64™ X+ in nonstandard floating-point mode, which does not conform to IEEE 754-1985. Nonstandard floating-point mode is enabled when **FSR.ns** = 1 (refer to page 27). The floating-point behavior depends on the value of **FSR.ns**.

This section also describes the conditions that generate an *fp_exception_other* exception with **FSR.ftt** = *unfinished_FPop*, even though this exception only occurs in standard floating-point mode (**FSR.ns** = 0).

SPARC64™ X / SPARC64™ X+ floating-point hardware only handles numbers in a specific range. If the hardware determines from the values of the source operands or the intermediate result that the final result may not be in the specified range, an *fp_exception_other* exception with **FSR.ftt** = 02₁₆ (*unfinished_FPop*) is generated. Subsequent processing is handled by software; an emulation routine completes the operation in accordance with IEEE 754-1985 (impl. dep. #3).

8.1.1. *fp_exception_other* (ftt = *unfinished_FPop*)

Almost all SPARC64™ X / SPARC64™ X+ floating-point arithmetic operations may cause an *fp_exception_other* exception with **FSR.ftt** = *unfinished_FPop*. Refer to the definition of the specific instruction for details. Conditions that generate this exception are described below.

- 1) When one operand is denormal and all other operands are normal (not zero, infinity, NaN), an *fp_exception_other* with *unfinished_FPop* occurs. The exception does not occur when the result is a zero or an overflow.
- 2) When all operands are denormal and the result is not a zero or an overflow, an *fp_exception_other* exception with *unfinished_FPop* occurs.
- 3) When all operands are normal, the result before rounding is denormal, **TEM.ufm** = 0, and the result is not a zero, an *fp_exception_other* exception with *unfinished_FPop* occurs.

When the result is expected to be a constant, such as zero or infinity, and the calculation can be handled by hardware, SPARC64™ X / SPARC64™ X+ performs the operation in hardware. An *unfinished_FPop* does not occur.

Table 8-1 describes the formulas used to estimate the exponent of the result so that hardware can determine whether to generate an *unfinished_FPop*. Here, **Er** is an

approximation of the biased exponent of the result before the significand is aligned and before rounding; Er is calculated using only the source exponents (esrc1, esrc2).

Table 8-1 Estimating the Exponent of the Result

Operation	Formula
fmul _s	$Er = esrc1 + esrc2 - 126$
fmul _d	$Er = esrc1 + esrc2 - 1022$
fdiv _s	$Er = esrc1 - esrc2 + 126$
fdiv _d	$Er = esrc1 - esrc2 + 1022$

esrc1 and esrc2 are the biased exponents of the source operands. When a source operand is a denormal number, the corresponding exponent is 0.

Once Er is calculated, eres can be obtained. eres is the biased exponent of the result after the significand is aligned and before rounding. That is, the significand is left-shifted or right-shifted so that an implicit 1 is immediately to the left of the binary point. eres is the value obtained from adding or subtracting the amount shifted to the value of Er.

Table 8-2 describes the conditions under which each floating-point instruction generates an *unfinished_FPop* exception.

Table 8-2 *unfinished_FPop* Conditions

Instructions	Conditions
FdT0s	$-25 < eres < 1$ and TEM.ufm = 0
FST0d	The second operand (rs2) is denormal
FADD{s d}, FSUB{s d}, FNADD{s d}	- One operand is denormal, and the other operand is normal (not zero, infinity, NaN)ⁱ - Both operands are denormal - Both operands are normal (not zero, infinity, NaN), $eres < 1$, and TEM.ufm = 0
FMUL{s d}, FNMUL{s d}	- One operands is denormal, the other operand is normal (not zero, infinity, NaN), and single precision : $-25 < Er$ double precision: $-54 < Er$ - Both operands are normal (not zero, infinity, NaN), TEM.ufm = 0, and single precision : $-25 < eres < 1$ double precision: $-54 < eres < 1$
F{ N}SMULd	- One operand is denormal, and the other operand is normal (not zero, infinity, NaN) - Both operands are denormal
FDIV{s d}	- The dividend (rs1) is normal (not zero, infinity, NaN), the divisor (rs2) is denormal, and single precision : $Er < 255$ double precision: $Er < 2047$ - The dividend (rs1) is denormal, the divisor (rs2) is normal (not zero, infinity, NaN), and single precision : $-25 < Er$ double precision: $-54 < Er$ - Both operands are denormal - Both operands are normal (not zero, infinity, NaN), TEM.ufm = 0, and single precision : $-25 < eres < 1$ double precision: $-54 < eres < 1$

ⁱ When the source operand is zero and denormal, the generated result conforms to IEEE754-1985.

FSQRT{s d}	The source operand (rs2) is positive, nonzero, and denormal
FMADD{s d}, FMSUB{s d}, FNMADD{s d}, FNMSUB{s d}	Multiply: - One operand is denormal, the other operand is normal (not zero, infinity, NaN), and single precision: $-25 < E_r$ double precision: $-54 < E_r$ - Both operands are normal (not zero, infinity, NaN), TEM.ufm = 0, and single precision: $-25 < e_{res} < 1$ double precision: $-54 < e_{res} < 1$ Add: - F[rs3] is denormal and the multiplication result is normal (not zero, infinity, NaN) - Both F[rs3] and the multiplication result are denormal - F[rs3] is normal (not zero, infinity, NaN), TEM.ufm = 0, and single precision: $-25 < e_{res} < 1$ double precision: $-54 < e_{res} < 1$
FTRIMADDd	Same as FMUL{s d} for the multiply. Not detected for the add.
FTRISMULD	When rs1 is normal (not zero, infinity, NaN), TEM.ufm = 0, and $-54 < e_{res} < 1$
FRCPA{s d}	When the operands are denormal
FRSQRTA{s d}	When the operands are positive, nonzero, and denormal

Conditions for a Zero Result

SPARC64™ X / SPARC64™ X+ generate a zero result when the result is a denormalized minimum or a zero, depending on the rounding mode (FSR.rd). This result is called a “pessimistic zero”. Table 8-3 shows the conditions for a zero result.

Table 8-3 Conditions for a Zero Result

Instructions	Conditions		
	One operand is denormal ⁱⁱ	Both are denormal	Both are normal ⁱⁱⁱ
FdTos	always	—	—
FMUL{s d}, FNMUL{s d}	single precision: $E_r \leq -25$ double precision: $E_r \leq -54$	always	single precision: $e_{res} \leq -25$ double precision: $e_{res} \leq -54$
F{N}M{ADD SUB}{s d}	Multiply: single precision: $E_r \leq -25$ double precision: $E_r \leq -54$ Add: F[rs3] is normal (not infinity, NaN), the multiplication result is denormal, and single precision: $e_{res} \leq -25$ double precision: $e_{res} \leq -54$	Multiply: always Add: never	Multiply: single precision: $e_{res} \leq -25$ double precision: $e_{res} \leq -54$ Add: single precision: $e_{res} \leq -25$ double precision: $e_{res} \leq -54$
FDIV{s d}	single precision: $E_r \leq -25$ double precision: $E_r \leq -54$	never	single precision: $e_{res} \leq -25$ double precision: $e_{res} \leq -54$
FTRIMADDd	Fd[rs2]<63> = 0 and index = 7 and $E_r \leq -54$	Fd[rs2]<63> = 0 and index = 7	Fd[rs2]<63> = 0 and index = 7 and $e_{res} \leq -54$

ⁱⁱ Except when both operands are zero, NaN, or infinity.

ⁱⁱⁱ And neither operand is NaN or infinity. If both operands are zero, e_{res} is never less than zero.

FTRISMULd	Fd[rs1] is denormal	—	Fd[rs1] is normal and $\text{eres} \leq -54$
-----------	---------------------	---	--

Conditions for an Overflow Result

SPARC64™ X / SPARC64™ X+ assume the instruction causes an overflow for the conditions listed in Table 8-4.

Table 8-4 Conditions for an Overflow Result

Instructions	Conditions
FDIVs	The divisor (rs2) is denormal and $\text{Er} \geq 255$
FDIVd	The divisor (rs2) is denormal and $\text{Er} \geq 2047$

8.1.2. Behavior when FSR.ns = 1

Compatibility Note In section 8.4 in UA2011, the behavior of some instructions (for example, FADD, FDIV, and FMUL) is required to follow IEEE Std. 754 at all times regardless of the value of FSR.ns. However, in SPARC64™ X / SPARC64™ X+, the behavior of all floating-point instructions is changed according to the value of FSR.ns (refer to page 27).

When FSR.ns = 1 (nonstandard mode), SPARC64™ X / SPARC64™ X+ replace all denormal source operands and denormal results with zeroes. This behavior is described below in greater detail:

- When one operand is denormal and none of the operands is zero, infinity, or NaN, the denormal operand is replaced with a zero of the same sign, and the operation is performed. After the operation, cexc.nxc is set to 1 unless one of the following conditions occurs; in which case, cexc.nxc = 0.
 - A division_by_zero or an invalid_operation is detected for FDIV{s|d}.
 - An invalid_operation is detected for FSQRT{s|d}.
 - The operation is FRCPA{s|d} or FRSQRTA{s|d}.

When cexc.nxc = 1 and tem.nxm = 1 in FSR, an *fp_exception_ieee_754* exception occurs.

- When the result before rounding is denormal, the result is replaced with a zero of the same sign. If tem.ufm = 1 in FSR, then cexc.ufc = 1; if tem.ufm = 0 and tem.nxm = 1, then cexc.nxc = 1. In both cases, an *fp_exception_ieee_754* exception occurs. When tem.ufm = 0 and tem.nxm = 0, both cexc.ufc and cexc.nxc are set to 1.

When FSR.ns = 1, SPARC64™ X / SPARC64™ X+ do not generate *unfinished_FPop* exceptions and do not return denormal numbers as results.

Table 8-5 summarizes the exceptions generated by the floating-point arithmetic instructions^{iv} listed in Table 8-2. All possible exceptions and masked exceptions are listed in the "Result" column. The generated exception depends on the value of FSR.ns, the source operand type, the result type, and the value of FSR.tem; it can be found by tracing the

^{iv} rs2 for FTRISMULd is not a floating-point number and cannot be denormal.

conditions from left to right. If FSR.ns = 1 and the source operands are denormal, refer to Table 8-6. In Table 8-5, the shaded areas in the “Result” column conform to IEEE754-1985.

Note In Table 8-5 and Table 8-6, lowercase exceptional conditions (nx, uf, of, dv, nv) do not signal IEEE 754 exceptions. Uppercase exceptional conditions (NX, UF, OF, DZ, NV) do signal IEEE 754 exceptions.

Table 8-5 Floating-Point Exceptional Conditions and Results

FSR.ns	Source Denormal v	Result Denormal vi	Zero Result	Overflow Result	UFM	OFM	NXM	Result
0	No	Yes	Yes	—	1	—	—	UF
					0	—	1	NX
							0	uf + nx, a signed zero, or a signed Dmin ^{vii}
			No	—	1	—	—	UF
		0			—	—	<i>unfinished_FPop</i> ^{viii}	
		No	—	—	—	—	—	Conforms to IEEE754-1985
	Yes	—	Yes	—	1	—	—	UF
					0	—	1	NX
							0	uf + nx, a signed zero, or a signed Dmin
			No	Yes	—	1	—	OF
		—			0	1	NX	
		No		—	—	—	<i>unfinished_FPop</i>	
			1	No	—	—	1	—
0	—	1					NX	
		0					uf + nx, a signed zero	
No	—	—		—	—	—	Conforms to IEEE754-1985	
Yes	—	—	—	—	—	—	Table 8-6	

Table 8-6 describes SPARC64™ X behavior when FSR.ns = 1 (nonstandard mode). Shaded

^v One operand is denormal, and the other operands are normal (not zero, infinity, NaN) or denormal.

^{vi} The result before rounding is denormal.

^{vii} Dmin = denormalized minimum.

^{viii} If the instruction is FADD{s|d} or FSUB{s|d} and the source operands are zero and denormal, SPARC64 X / SPARC64 X+ does not generate an *unfinished_FPop*; instead, the operation is performed conformant to IEEE754-1985.

^{ix} Nmax = normalized maximum.

areas in the “Result” column conform to IEEE754-1985.

Table 8-6 Operations with Denormal Source Operands when FSR.ns = 1

Instructions	Source Operands			FSR.tem				Result
	op1	op2	op3	UFM	NXM	DVM	NVM	
FSTOd	—	Denorm	—	—	1	—	—	NX
					0	—	—	nx, a signed zero
FdTOs	—	Denorm	—	1	—	—	—	UF
				0	1	—	—	NX
					0	—	—	uf + nx, a signed zero
FADD{s d}, FSUB{s d}, FNADD{s d}, FNSUB{s d}	Denorm	Normal	—	—	1	—	—	NX
	Normal	Denorm	—		0	—	—	nx, op2
					1	—	—	NX
					0	—	—	nx, op1
					1	—	—	NX
	Denorm	Denorm	—		0	—	—	nx,a signed zero
FMUL{s d}, FsmULd, FNMUL{s d}, FNSMULd	Denorm	—	—	—	1	—	—	NX
	—	Denorm	—		0	—	—	nx,a signed zero
					1	—	—	NX
					0	—	—	nx,a signed zero
FDIV{s d}	Denorm	Normal	—	—	1	—	—	NX
	Normal	Denorm	—		0	—	—	nx,a signed zero
					—	1	—	DZ
					0	—	dz, a signed infinity	
					Denorm	Denorm	—	—
	0	nv, dNaN ^x						
FSQRT{s d}	—	Denorm and op2 > 0	—	—	1	—	—	NX
					0	—	—	nx, zero
		Denorm and op2 < 0	—		—	—	1	NV
							0	nv, dNaN ^x
FMADD{s d}, FMSUB{s d}, FNMADD{s d}, FNMSUB{s d},	Denorm	—	Normal	—	1	—	—	NX
			Denorm	0	—	—	nx, op3	
		1		—	—	NX		

^x A single-precision dNaN is 7FFFFFFF₁₆, and a double-precision dNaN is 7FFFFFFFFFFFFFFF₁₆.

FTRIMADDd ^{xi}					0	—	—	nx, a signed zero
	—	Denorm	Normal	—	1	—	—	NX
					0	—	—	nx, op3
			Denorm	—	1	—	—	NX
					0	—	—	nx, a signed zero
	Normal	Normal	Denorm	—	1	—	—	NX
					0	—	—	nx, op1 × op2 ^{xii}
FTRISMULD	Denorm	—	—	—	1	—	—	NX
					0	—	—	nx, zero whose sign bit is op2<0>
FRCPA{s d}	—	Denorm	—	—	—	1	—	DZ
						0	—	dz, a signed infinity
FRSQRTA{s d}	—	Denorm	—	—	—	1	—	DZ
						0	—	dz, a signed infinity

^{xi} op3 is obtained from a table in the functional unit and is always normal.

^{xii} When op1 × op2 is denormal, op1 × op2 becomes a zero with the same sign.

9. Memory Models

Refer to Chapter 9 in UA2011.

10. Address Space Identifiers

Refer to Chapter 10 in UA2011.

10.3. ASI Assignment

10.3.1. Supported ASIs

ASIs supported on SPARC64™ X / SPARC64™ X+ are listed in Table 10-2. The notation for the Type and Sharing columns in Table 10-2 are described in Table 10-1.

Table 10-1 **Notation used in Table 10-2**

Column	Symbol	Meaning
Type	Trans.	The translation mode is determined by the privilege level and MMU settings.
	Real	Address is treated as a real address (RA).
	non-T	Not translated by MMU. VA watchpoint not detected.
Sharing (non-T only)	Chip	Register is shared by the entire CPU chip.
	Core	Register is shared by VCPUs in the same core.
	VCPU	Each VCPU has its own copy of the register.

Table 10-2 **ASI list**

ASI	VA	ASI name	Access	Type	Sharing	Page
80 ₁₆	—	ASI_PRIMARY(ASI_P)	RW	Trans.	—	
81 ₁₆	—	ASI_SECONDARY(ASI_S)	RW	Trans.	—	
82 ₁₆	—	ASI_PRIMARY_NO_FAULT(ASI_PNF)	RO	Trans.	—	
83 ₁₆	—	ASI_SECONDARY_NO_FAULT(ASI_SNF)	RO	Trans.	—	
84 ₁₆ – 87 ₁₆	—	—	—	—	—	
88 ₁₆	—	ASI_PRIMARY_LITTLE(ASI_PL)	RW	Trans.	—	
89 ₁₆	—	ASI_SECONDARY_LITTLE(ASI_SL)	RW	Trans.	—	
8A ₁₆	—	ASI_PRIMARY_NO_FAULT_LITTLE(ASI_PNFL)	RO	Trans.	—	
8B ₁₆	—	ASI_SECONDARY_NO_FAULT_LITTLE(ASI_SNFL)	RO	Trans.	—	
8C ₁₆ – BF ₁₆	—	—	—	—	—	
C0 ₁₆	—	ASI_PST8_PRIMARY(ASI_PST8_P)	WO	Trans.	—	177, 276
C1 ₁₆	—	ASI_PST8_SECONDARY(ASI_PST8_S)	WO	Trans.	—	177, 276

ASI	VA	ASI name	Access	Type	Sharing	Page
C2 ₁₆	—	ASI_PST16_PRIMARY(ASI_PST16_P)	WO	Trans.	—	177, 276
C3 ₁₆	—	ASI_PST16_SECONDARY(ASI_PST16_S)	WO	Trans.	—	177, 276
C4 ₁₆	—	ASI_PST32_PRIMARY(ASI_PST32_P)	WO	Trans.	—	177, 276
C5 ₁₆	—	ASI_PST32_SECONDARY(ASI_PST32_S)	WO	Trans.	—	177, 276
C6 ₁₆ – C7 ₁₆	—	—	—	—	—	
C8 ₁₆	—	ASI_PST8_PRIMARY_LITTLE(ASI_PST8_PL)	WO	Trans.	—	177, 276
C9 ₁₆	—	ASI_PST8_SECONDARY_LITTLE(ASI_PST8_SL)	WO	Trans.	—	177, 276
CA ₁₆	—	ASI_PST16_PRIMARY_LITTLE(ASI_PST16_PL)	WO	Trans.	—	177, 276
CB ₁₆	—	ASI_PST16_SECONDARY_LITTLE(ASI_PST16_SL)	WO	Trans.	—	177, 276
CC ₁₆	—	ASI_PST32_PRIMARY_LITTLE(ASI_PST32_PL)	WO	Trans.	—	177, 276
CD ₁₆	—	ASI_PST32_SECONDARY_LITTLE(ASI_PST32_SL)	WO	Trans.	—	177, 276
CE ₁₆ – CF ₁₆	—	—	—	—	—	
D0 ₁₆	—	ASI_FL8_PRIMARY(ASI_FL8_P)	RW	Trans.	—	132, 178, 276
D1 ₁₆	—	ASI_FL8_SECONDARY(ASI_FL8_S)	RW	Trans.	—	132, 178, 276
D2 ₁₆	—	ASI_FL16_PRIMARY(ASI_FL16_P)	RW	Trans.	—	132, 178, 276
D3 ₁₆	—	ASI_FL16_SECONDARY(ASI_FL16_S)	RW	Trans.	—	132, 178, 276
D4 ₁₆ – C7 ₁₆	—	—	—	—	—	
D8 ₁₆	—	ASI_FL8_PRIMARY_LITTLE(ASI_FL8_PL)	RW	Trans.	—	132, 178, 276
D9 ₁₆	—	ASI_FL8_SECONDARY_LITTLE(ASI_FL8_SL)	RW	Trans.	—	132, 178, 276
DA ₁₆	—	ASI_FL16_PRIMARY_LITTLE(ASI_FL16_PL)	RW	Trans.	—	132, 178, 276
DB ₁₆	—	ASI_FL16_SECONDARY_LITTLE(ASI_FL16_SL)	RW	Trans.	—	132, 178, 276
DC ₁₆ – DF ₁₆	—	—	—	—	—	
E0 ₁₆	—	ASI_BLOCK_COMMIT_PRIMARY(ASI_BLK_COMMIT_P)	WO	Trans.	—	167, 276
E1 ₁₆	—	ASI_BLOCK_COMMIT_SECONDARY(ASI_BLK_COMMIT_S)	WO	Trans.	—	167, 276

ASI	VA	ASI name	Access	Type	Sharing	Page
E2 ₁₆	—	ASI_TWIX_P/ASI_STBI_P	RW	Trans.	—	138, 166
E3 ₁₆	—	ASI_TWIX_S/ASI_STBI_S	RW	Trans.	—	138, 166
E6 ₁₆	—	—	—	—	—	
E8 ₁₆ – E9 ₁₆	—	—	—	—	—	
EA ₁₆	—	ASI_TWIX_PL/ASI_STBI_PL	RW	Trans.	—	138, 166
EB ₁₆	—	ASI_TWIX_SL/ASI_STBI_SL	RW	Trans.	—	138, 166
EC ₁₆ – EE ₁₆	—	—	—	—	—	
EF ₁₆	00 ₁₆ – 38 ₁₆	ASI_LBSY, ASI_BST	RW	non-T	VCPU	
F0 ₁₆	—	ASI_BLOCK_PRIMARY(ASI_BLK_P)	RW	Trans.	—	124, 167, 276
F1 ₁₆	—	ASI_BLOCK_SECONDARY(ASI_BLK_S)	RW	Trans.	—	124, 167, 276
F2 ₁₆	—	ASI_XFILL_PRIMARY(ASI_XFILL_P)	WO	Trans.	—	190
F3 ₁₆	—	ASI_XFILL_SECONDARY(ASI_XFILL_S)	WO	Trans.	—	190
F4 ₁₆ – F7 ₁₆	—	—	—	—	—	
F8 ₁₆	—	ASI_BLOCK_PRIMARY_LITTLE(ASI_BLK_PL)	RW	Trans.	—	124, 167, 276
F9 ₁₆	—	ASI_BLOCK_SECONDARY_LITTLE(ASI_BLK_SL)	RW	Trans.	—	124, 167, 276
FA ₁₆ – FF ₁₆	—	—	—	—	—	

10.3.2. ASI access exceptions

On SPARC64™ X / SPARC64™ X+, some exceptions that occur when an undefined ASI is specified or the combination of an instruction and an ASI is illegal are different from JPS1 Commonality.

10.3.2.1. Illegal combination of ASI and instruction

Exceptions caused by illegal combinations of ASIs and instructions are explained below in the order generated. That is, the exceptions are listed by priority from high to low.

1. An *illegal_instruction* exception may occur for LDBLOCKF, STBLOCKF, and STPARTIALF. Refer to the definition of each instruction for details. (An *illegal_instruction* exception also occurs for LDTWA, LDTXA, or STTWA when an odd-numbered register is specified for rd.)
2. The *mem_address_not_aligned* and **_mem_address_not_aligned* exceptions occur when the access does not meet the alignment requirements for the instruction.
 - a) LDBLOCKF and STBLOCKF require 64-byte alignment, and *mem_address_not_aligned* occurs when accessing an address that is not

64-byte aligned. *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* are never generated. (A *mem_address_not_aligned* exception is not generated when the Block Commit Store ASIs E0₁₆, and E1₁₆ are specified by LDDFA.)

- b) The 16-bit LDSHORTF and STSHORTF instructions (ASIs D0₁₆, D1₁₆, D8₁₆, and D9₁₆) require 2-byte alignment, and *mem_address_not_aligned* occurs when accessing an address that is not 2-byte aligned. *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* are never generated.
 - c) The 8-bit LDSHORTF and STSHORTF instructions (ASIs D2₁₆, D3₁₆, DA₁₆, and DB₁₆) require 1-byte alignment and do not cause any alignment violations.
 - d) STPARTIALF requires 8-byte alignment, and *mem_address_not_aligned* occurs when accessing an address that is not 8-byte aligned. *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* are never generated. (A *mem_address_not_aligned* exception is not generated when the Partial Store ASIs C0₁₆ – C5₁₆, C8₁₆ – CD₁₆ are specified by LDDFA.)
 - e) *LDDF_mem_address_not_aligned* and *STDF_mem_address_not_aligned* exceptions are generated when an ASI other than the instructions listed above is specified by LDDFA and STDFA, respectively, and the target address is 4-byte aligned but not 8-byte aligned.
 - f) A *mem_address_not_aligned* exception is generated if there is an alignment violation other than those described above.
3. A *DAE_invalid_asi* exception is generated when the combination of the ASI and the instruction is invalid. (*DAE_invalid_asi* is not generated for PREFETCHA, which is treated as a NOP in this case.)

10.3.2.2. Undefined ASI space

A *privileged_action* or *DAE_invalid_asi* exception is generated when an undefined ASI space is accessed. The exception that is generated is determined by the ASI number and the privileged mode at the time of access. Table 10-3 shows these exceptions. A *privileged_action* or *DAE_invalid_asi* exception is not generated when the access is misaligned, even if the ASI is undefined.

Table 10-3 Exceptions when an undefined ASI space is accessed

ASI	Non-privileged
00 ₁₆ - 2F ₁₆	<i>privileged_action</i>
30 ₁₆ - 7F ₁₆	<i>privileged_action</i>
80 ₁₆ - FF ₁₆	<i>DAE_invalid_asi</i>

11. Performance Instrumentation

11.1. Overview

Performance counters comprise one “Performance Control Register (PCR) (ASR 16)” and multiple instances of “Performance Instrumentation Counter Register (PIC) (ASR 17)”.

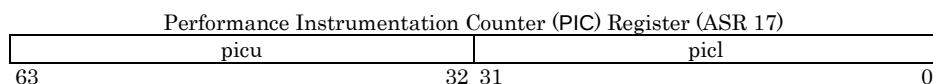
SPARC64™ X / SPARC64™ X+ implement 4 PIC registers, which are selected by PCR.SC and accessed via ASR 17. Each PIC register contains two counters.

Performance Control Register (PCR) (ASR 16)

Performance Counter Register (PC) (bits 15)																															
—		toe		—		ovf		ovro		ulro		—		nc		su		sl		—		sc		ht		ut		st		priv	
63	56	55	48	47	40	39	32	31	30	29	27	26	24	23	16	15	8	7	6	4	3	2	1	0							

Bits	Field	Access	Description																
55:48	toe<7:0>	RW	Controls whether an overflow exception is generated for performance counters. A write updates the field and a read returns the current settings. If toe<i> is 1 and the counter corresponding to ovf<i> overflows, ovf<i> = 1 and a pic_overflow exception is generated. If toe<i> is 0 and the counter corresponding to ovf<i> overflows, ovf<i> = 1 but a pic_overflow exception is not generated. When ovf<i> = 1 and the value of toe<i> is changed to 1, a pic_overflow exception is not generated.																
39:32	ovf<7:0>	RW	Overflow Clear/Set/Status. A read by RDPCR returns the overflow status of the counters, and a write by WRPCR clears or sets the overflow status bits.The following figure shows the PIC counters corresponding to the OVF bits. A write of 0 to an OVF bit clears the overflow status of the corresponding counter.																
<table><tr><td>U3</td><td>L3</td><td>U2</td><td>L2</td><td>U1</td><td>L1</td><td>U0</td><td>L0</td></tr><tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>				U3	L3	U2	L2	U1	L1	U0	L0	7	6	5	4	3	2	1	0
U3	L3	U2	L2	U1	L1	U0	L0												
7	6	5	4	3	2	1	0												
31	ovro	RW	Overflow Read-Only. A write to the PCR register with write data containing a value of ovro = 0 updates the PCR.ovf field with the OVF write data. If the write data contains a value of ovro = 1, the OVF write data is ignored and the PCR.ovf field is not updated. Reads of the PCR.ovro field return 0. The PCR.ovro field allows PCR to be updated without changing the overflow status. Hardware maintains the most recent state in PCR.ovf such that a subsequent read of the PCR returns the current overflow status.																
30	ulro	RW	su/sl Read-Only. A write to the PCR register with write data containing a value of ulro = 0 updates the PCR.su and PCR.sl fields with the su/sl write data. If the write data contains a value of ulro = 1, the su/sl write data is ignored and the PCR.su and PCR.sl																

			fields are not updated. Reads of the PCR.ulro field return 0. The PCR.ulro field allows the PIC pair selection field to be updated without changing the PCR.su and PCR.sl settings.
26:24	nc	RO	This read-only field indicates the number of PIC counter pairs.
23:16	su	RW	This field selects the event counted by PIC<63:32>. A write updates the setting, and a read returns the current setting.
15:8	sl	RW	This field selects the event counted by PIC<31:0>. A write updates the setting, and a read returns the current setting.
6:4	sc	RW	PIC Pair Selection. A write updates which PIC counter pair is selected, and a read returns the current selection. When a “1” is written to bit<6>, no counter pair is selected and a subsequent read returns “0”.
3	ht	RO	If ht = 1, events that occur while in hypervisor mode are counted. Writes to this field are ignored.
2	ut	RW	User Mode. When PSTATE.priv = 0 and ut = 1, events are counted.
1	st	RW	System Mode. When PSTATE.priv = 1 and st = 1, events are counted. If both PCR.ut and PCR.st are 1, all events are counted. If both PCR.ut and PCR.st are 0, counting is disabled. PCR.ut and PCR.st are global fields; that is, they apply to all PIC counter pairs.
0	priv	RW	Privileged. If PCR.priv = 1, executing a RDPCR, WRPCR, RDPIC, or WRPIC instruction in non-privileged mode (PSTATE.priv = 0) causes a privileged_action exception. If PCR.priv = 0, a non-privileged (PSTATE.priv = 0) attempt to update PCR.priv (that is, to write a value of 1) via a WRPCR instruction causes a privileged_action exception.



Bits	Field	Access	Description
63:32	picu	RW	32-bit counter for the event selected by PCR.su.
31:0	picl	RW	32-bit counter for the event selected by PCR.sl.

11.1.1. Sample Pseudo-codes

11.1.1.1. Counter Clear/Set

The counter fields in the PIC registers are read/write fields. Writing zero clears a counter; writing any other value sets the counter to that value. The following pseudo-code clears all PIC registers (assuming privileged access).

```

/* Clear PICs without updating SU/SL values */
pic_init = 0x0;
pcr = rd_pcr();
pcr.ulro = 0x1; /* don't update SU/SL on write */
pcr.ovf = 0x0; /* clear overflow bits */
pcr.ut = 0x0;
pcr.st = 0x0; /* disable counts */
pcr.ht = 0x0; /* non-hypervisor mode */
pcr.priv = 0x0; /* privileged access */
for (i=0; i<=pcr.nc; i++) {
/* select the PIC to be written */
pcr.sc = i;
wr_pcr(pcr);
wr_pic(pic_init); /* clear counters in PIC[i] */
}

```

11.1.1.2. Select and Enable Counter Events (SPARC64™ X)

Counter events are selected using the PCR.sc and PCR.su/PCR.sl fields. The following pseudo-code selects events and enables counters (assuming privileged access).

```

pcr.ut = 0x0; /* Disable user counts */
pcr.st = 0x0; /* Disable system counts also */
pcr.ht = 0x0; /* non-hypervisor mode */
pcr.priv = 0x0; /* privileged access */
pcr.ulro = 0x0; /* Make SU/SL writeable */
pcr.ovro = 0x1; /* Overflow is read-only */
/* Select events without enabling counters */
for(i=0; i<=pcr.nc; i++) {
    pcr.sc = i;
    pcr.sl = select an event;
    pcr.su = select an event;
    wr_pcr(pcr);
}
/* Start counting */
pcr.ut = 0x1;
pcr.st = 0x1;
pcr.ulro = 0x1; /* SU/SL is read-only */
/* Clear overflow bits here if needed */
wr_pcr(pcr);

```

11.1.1.3. Select and Enable Counter Events (SPARC64™ X+)

Counter events are selected using the PCR.sc and PCR.su/PCR.sl fields. The following pseudo-code selects events and enables counters (assuming privileged access).

```

pcr.ut = 0x0; /* Disable user counts */
pcr.st = 0x0; /* Disable system counts also */
pcr.ht = 0x0; /* non-hypervisor mode */
pcr.priv = 0x0; /* privileged access */
pcr.ulro = 0x0; /* Make SU/SL writeable */
pcr.ovro = 0x1; /* Overflow is read-only */
/* Select events without enabling counters */
for(i=0; i<=pcr.nc; i++) {
    pcr.sc = i;
    pcr.sl = select an event;
}

```

```

        pcr.su = select an event;
        wr_pcr(pcr);
    }
    /* Start counting */
    pcr.ut = 0x1;
    pcr.st = 0x1;
    pcr.ulro = 0x1; /* SU/SL is read-only */
    /* Clear overflow bits here if needed */
    wr_pcr(pcr);

```

11.1.1.4. Stop Counter and Read

The following pseudo-code disables counters and reads their values (assuming privileged access).

```

    pcr.ut = 0x0; /* Disable user counts */
    pcr.st = 0x0; /* Disable system counts also */
    pcr.ht = 0x0; /* non-hypervisor mode */
    pcr.priv = 0x0; /* privileged access */
    pcr.ulro = 0x1; /* Make SU/SL read-only */
    pcr.ovro = 0x1; /* Overflow is read-only */
    for(i=0; i<=pcr.nc; i++) {
        pcr.sc = i;
        wr_pcr(pcr);
        pic = rd_pic();
        picl[i] = pic.picl;
        picu[i] = pic.picu;
    }

```

11.2. Description of PA Events

The performance counter (PA) events can be divided into the following groups:

1. Instruction and trap statistics
2. MMU and L1 cache events
3. L2 cache events
4. Bus transaction events

There are 2 types of events that can be measured on SPARC64™ X / SPARC64™ X+, standard and supplemental events.

Standard events on SPARC64™ X / SPARC64™ X+ have been verified for correct behavior; they are guaranteed to be compatible with future processors.

Supplemental events are primarily intended for debugging the hardware.

- a. The behavior of supplemental events may not be fully verified. There is a possibility that some of these events may not behave as specified in this document.
- b. The definition of these events may change without notice. Compatibility with future processors is not guaranteed.

Table 11-1 shows the PA events defined on SPARC64™ X. Table 11-2 shows the PA events defined on SPARC64™ X+. Shaded events are supplemental events.

For details of each event, refer to the descriptions in the following sections. Unless otherwise indicated, speculative instructions are also counted by PA events.

Table 11-1 PA Events and Encodings (SPARC64™ X)

Encoding (binary)	Counter							
	pic u0	pic l0	pic u1	pic l1	pic u2	pic l2	pic u3	pic l3
0000_0000	cycle_counts							
0000_0001	instruction_counts							
0000_0010	instruction_flow_counts	<i>only_this_thread_active</i>	<i>single_mode_cycle_counts</i>	<i>single_mode_instruction_coun</i>	instruction_flow_counts	<i>d_move_wait</i>	<i>cse_priority_wait</i>	xma_inst
0000_0011	iwr_empty	<i>w_cse_window_empty</i>	<i>w_eu_comp_wait</i>	<i>w_branch_comp_wait</i>	iwr_empty	<i>w_op_stv_wait</i>	<i>w_d_move</i>	<i>w_0endop</i>
0000_0100	<i>Reserved</i>	<i>w_op_stv_wait_nc_pend</i>	<i>w_op_stv_wait_sxmiss</i>	<i>w_op_stv_wait_sxmiss_ex</i>	<i>Reserved</i>	<i>w_fl_comp_wait</i>	<i>w_cse_window_empty_sp_full</i>	<i>w_op_stv_wait_ex</i>
0000_0101	op_stv_wait							
0000_0110	effective_instruction_counts							
0000_0111	SIMD_load_store	SIMD_floating_instructions	SIMD_fma_instructions	sxar1_instructions	sxar2_instructions	unpack_sxar1	unpack_sxar2	<i>Reserved</i>
0000_1000	load_store_instructions							
0000_1001	branch_instructions							
0000_1010	floating_instructions							
0000_1011	fma_instructions							
0000_1100	prefetch_instructions							
0000_1101	<i>Reserved</i>	ex_load_instructions	ex_store_instructions	fl_load_instructions	fl_store_instructions	SIMD_fl_load_instructions	SIMD_fl_store_instructions	<i>Reserved</i>
0000_1110	<i>Reserved</i>							
0000_1111	<i>Reserved</i>							
0001_0000	<i>Reserved</i>							
0001_0001	<i>Reserved</i>							
0001_0010	rs1	flush_rs	<i>Reserved</i>					
0001_0011	1iid_use	2iid_use	3iid_use	4iid_use	<i>Reserved</i>	sync_intlk	regwin_intlk	<i>Reserved</i>
0001_0100	<i>Reserved</i>							
0001_0101	<i>Reserved</i>	toq_rsbr_phantom	<i>Reserved</i>	flush_rs	<i>Reserved</i>		rs1	<i>Reserved</i>
0001_0110	trap_all	Reserved	trap_int_level	trap_spill	trap_fill	trap_trap_inst	Reserved	Reserved
0001_0111	<i>Reserved</i>	Reserved	<i>Reserved</i>					
0001_1000	<i>only_this_thread_active</i>	<i>both_threads_active</i>	<i>both_threads_empty</i>	<i>Reserved</i>			<i>op_stv_wait_pfp_busy_swfp</i>	<i>op_stv_wait_sxmiss</i>
0001_1001	<i>Reserved</i>							
0001_1010	<i>Reserved</i>	<i>Reserved</i>	<i>single_sxar_comm_it</i>	<i>Reserved</i>				<i>suspend_cycle</i>
0001_1011	rsf_pmmi	<i>Reserved</i>	op_stv_wait_nc_pend	0iid_use	flush_rs	<i>Reserved</i>		decode_all_intlk
0001_1100	<i>Reserved</i>			<i>Reserved</i>	<i>Reserved</i>	<i>Reserved</i>	<i>Reserved</i>	<i>Reserved</i>
0001_1101	<i>op_stv_wait_pfp_busy_ex</i>	<i>Reserved</i>	op_stv_wait_sxmiss_ex	op_stv_wait_nc_pend	cse_window_empty_sp_full	op_stv_wait_pfp_busy	<i>both_threads_suspended</i>	<i>Reserved</i>

0001_1110	cse_window_empty	eu_comp_wait	branch_comp_wait	0endop	op_stv_wait_ex	fl_comp_wait	1endop	2endop
0001_1111	Reserved				3endop	Reserved	sleep_cycle	op_stv_wait_swpf
0010_0000	ITLB_write	DTLB_write	uITLB_miss	uDTLB_miss	L1I_miss	L1D_miss	L1I_wait_all	L1D_wait_all
0010_0001	Reserved							
0010_0010	Reserved							
0010_0011	L1I_thrashing	L1D_thrashing	Reserved					
0010_0100	swpf_success_all	swpf_fail_all	Reserved		swpf_lbs_hit	Reserved		
0010_0101	Reserved							
0010_0110	Reserved							
0010_0111	Reserved							
0010_1000	Reserved							
0010_1001	Reserved							
0010_1010	Reserved							
0010_1011	Reserved							
0010_1100	Reserved							
0010_1101	Reserved							
0010_1110	Reserved							
0010_1111	Reserved							
0011_0000	Reserved		L2_miss_dm	L2_miss_pf	L2_read_dm	L2_read_pf	L2_wb_dm	L2_wb_pf
0011_0001	bi_counts	cpi_counts	cpb_counts	cpd_counts	cpu_mem_read_counts	cpu_mem_write_counts	IO_mem_read_counts	IO_mem_write_counts
0011_0010	L2_miss_wait_dm_bank0	L2_miss_wait_pf_bank0	L2_miss_counts_dm_bank0	L2_miss_counts_pf_bank0	L2_miss_wait_dm_bank1	L2_miss_wait_pf_bank1	L2_miss_counts_dm_bank1	L2_miss_counts_pf_bank1
0011_0011	L2_miss_counts_dm_bank2	L2_miss_counts_pf_bank2	L2_miss_wait_dm_bank2	L2_miss_wait_pf_bank2	L2_miss_count_s_dm_bank3	L2_miss_count_s_pf_bank3	L2_miss_wait_dm_bank3	L2_miss_wait_pf_bank3
0011_0100	lost_pf_pfp_full	lost_pf_by_abort	IO_pst_counts	Reserved				
0011_0101	Reserved							
0011_0110	Reserved							
0011_0111	Reserved							
0011_1000	Reserved							
0011_1001	Reserved							
0011_1010	Reserved							
0011_1011	Reserved							
0011_1100	Reserved							
0011_1101	Reserved							
0011_1110	Reserved							
0011_1111	Reserved							
1111_1111	Disabled (Counter is not incremented)							

※Encodings not shown are Reserved.

Table 11-2 PA Events and Encodings (SPARC64™ X+)

Encoding (binary)	Counter							
	pic u0	pic l0	pic u1	pic l1	pic u2	pic l2	pic u3	pic l3
0000_0000	cycle_counts							
0000_0001	instruction_counts							
0000_0010	instruction_flow_counts	<i>only_this_thread_active</i>	<i>single_mode_cycle_counts</i>	<i>single_mode_instruction_count</i>	instruction_flow_counts	<i>d_move_wait</i>	<i>cse_priority_wait</i>	xma_inst
0000_0011	iwr_empty	<i>w_cse_window_empty</i>	<i>w_eu_comp_wait</i>	<i>w_branch_comp_wait</i>	iwr_empty	<i>w_op_stv_wait</i>	<i>w_d_move</i>	<i>w_0endop</i>
0000_0100	<i>Reserved</i>	<i>w_op_stv_wait_nc_pend</i>	<i>w_op_stv_wait_sxmiss</i>	<i>w_op_stv_wait_sxmiss_ex</i>	<i>Reserved</i>	<i>w_fl_comp_wait</i>	<i>w_cse_window_empty_sp_full</i>	<i>w_op_stv_wait_ex</i>
0000_0101	op_stv_wait							
0000_0110	effective_instruction_counts							
0000_0111	SIMD_load_store_instructions	SIMD_floating_instructions	SIMD_fma_instructions	sxar1_instructions	sxar2_instructions	unpack_sxar1	unpack_sxar2	<i>Reserved</i>
0000_1000	load_store_instructions							
0000_1001	branch_instructions							
0000_1010	floating_instructions							
0000_1011	fma_instructions							
0000_1100	prefetch_instructions							
0000_1101	<i>Reserved</i>	ex_load_instructions	ex_store_instructions	fl_load_instructions	fl_store_instructions	SIMD_fl_load_instructions	SIMD_fl_store_instructions	<i>Reserved</i>
0000_1110	<i>Reserved</i>							
0000_1111	<i>Reserved</i>							
0001_0000	<i>Reserved</i>							
0001_0001	<i>Reserved</i>							
0001_0010	rs1	flush_rs	<i>Reserved</i>					
0001_0011	1iid_use	2iid_use	3iid_use	4iid_use	<i>Reserved</i>	sync_intlk	regwin_intlk	<i>Reserved</i>
0001_0100	<i>Reserved</i>							
0001_0101	<i>Reserved</i>	toq_rsbr_phantom	<i>Reserved</i>	flush_rs	<i>Reserved</i>		rs1	<i>Reserved</i>
0001_0110	trap_all	Reserved	trap_int_level	trap_spill	trap_fill	trap_trap_inst	Reserved	Reserved
0001_0111	<i>Reserved</i>	Reserved	<i>Reserved</i>					
0001_1000	<i>only_this_thread_active</i>	<i>both_threads_active</i>	<i>both_threads_empty</i>	<i>Reserved</i>			<i>op_stv_wait_pfp_busy_swpf</i>	<i>op_stv_wait_sxmiss</i>
0001_1001	<i>Reserved</i>							
0001_1010	<i>Reserved</i>	<i>Reserved</i>	<i>single_sxar_commit</i>	<i>Reserved</i>				<i>suspend_cycle</i>
0001_1011	rsf_pmmi	<i>Reserved</i>	op_stv_wait_nc_pend	0iid_use	flush_rs	<i>Reserved</i>		decode_all_intlk
0001_1100	<i>Reserved</i>			<i>Reserved</i>	<i>Reserved</i>	<i>Reserved</i>	<i>Reserved</i>	<i>Reserved</i>
0001_1101	<i>op_stv_wait_pfp_busy_ex</i>	<i>Reserved</i>	op_stv_wait_sxmiss_ex	op_stv_wait_nc_pend	cse_window_empty_sp_full	op_stv_wait_pfp_busy	<i>both_threads_suspended</i>	<i>Reserved</i>
0001_1110	cse_window_empty	eu_comp_wait	branch_comp_wait	0endop	op_stv_wait_ex	fl_comp_wait	1endop	2endop
0001_1111	<i>Reserved</i>				3endop	<i>Reserved</i>	<i>sleep_cycle</i>	<i>op_stv_wait_swpf</i>
0010_0000	ITLB_write	DTLB_write	uITLB_miss	uDTLB_miss	L1I_miss	L1D_miss	L1I_wait_all	L1D_wait_all

0010_0001	Reserved							
0010_0010	Reserved							
0010_0011	L1I_thrashing	L1D_thrashing	Reserved					
0010_0100	swpf_success_all	swpf_fail_all	Reserved			swpf_lbs_hit	Reserved	
0010_0101	Reserved							
0010_0110	Reserved							
0010_0111	Reserved							
0010_1000	Reserved							
0010_1001	Reserved							
0010_1010	Reserved							
0010_1011	Reserved							
0010_1100	Reserved							
0010_1101	Reserved							
0010_1110	Reserved							
0010_1111	Reserved							
0011_0000	Reserved		L2_miss_dm	L2_miss_pf	L2_read_dm	L2_read_pf	L2_wb_dm	L2_wb_pf
0011_0001	bi_counts	cpi_counts	cpb_counts	cpd_counts	cpu_mem_read_counts	cpu_mem_write_counts	IO_mem_read_counts	IO_mem_write_counts
0011_0010	L2_miss_wait_dm_bank0	L2_miss_wait_pf_bank0	L2_miss_counts_dm_bank0	L2_miss_counts_pf_bank0	L2_miss_wait_dm_bank1	L2_miss_wait_pf_bank1	L2_miss_counts_dm_bank1	L2_miss_counts_pf_bank1
0011_0011	L2_miss_counts_dm_bank2	L2_miss_counts_pf_bank2	L2_miss_wait_dm_bank2	L2_miss_wait_pf_bank2	L2_miss_counts_dm_bank3	L2_miss_counts_pf_bank3	L2_miss_wait_dm_bank3	L2_miss_wait_pf_bank3
0011_0100	lost_pf_pfp_full	lost_pf_by_abort	IO_pst_counts	Reserved				
0011_0101	Reserved							
0011_0110	Reserved							
0011_0111	Reserved							
0011_1000	Reserved							
0011_1001	Reserved							
0011_1010	Reserved							
0011_1011	Reserved							
0011_1100	Reserved							
0011_1101	Reserved							
0011_1110	Reserved							
0011_1111	Reserved							
1111_1111	Disabled (Counter is not incremented)							

※Encodings not shown are Reserved.

11.2.1. Instruction and Trap Statistics

Standard PA Events

1 *cycle_counts*

Counts the number of cycles that the performance counter is enabled. This counter is similar to the TICK register but can count user cycles and system cycles separately, based on the settings of PCR.ut and PCR.st.

2 *instruction_counts* (Non-Speculative)

Counts the number of committed instructions, including SXAR1 and SXAR2. SPARC64™ X/SPARC64™ X+ commits up to 4 non-SXAR instructions per cycle and up to 2 SXAR instructions. That is, *instruction_counts* / *cycle_counts* can be greater than 4.

3 *effective_instruction_counts* (Non-Speculative)

Counts the number of committed non-SXAR instructions. Instructions per cycle (IPC) can be derived from this event and *cycle_counts*.

$$\text{IPC} = \text{effective_instruction_counts} / \text{cycle_counts}$$

If *effective_Instruction_counts* and *cycle_counts* are collected for user or system mode, the IPC in either user or system mode can be calculated.

4 *load_store_instructions* (Non-Speculative)

Counts the number of committed non-SIMD load/store instructions. Also counts atomic load-store instructions.

5 *branch_instructions* (Non-Speculative)

Counts the number of committed branch instructions. Also counts the CALL, JMPL, and RETURN instructions.

6 *floating_instructions* (Non-Speculative)

Counts the number of committed non-SIMD floating-point instructions. The counted instructions are FPop1, FPop2, FSELMOV{s|d}, and IMPDEP1 with opf<8:4> = 0A₁₆, 0B₁₆, 16₁₆, or 17₁₆.

7 *fma_instructions* (Non-Speculative)

Counts the number of committed non-SIMD floating-point multiply-and-add instructions. The counted instructions are FM{ADD|SUB}{s|d}, FNM{ADD|SUB}{s|d}, and FTRIMADDd. Two operations are executed per instruction; the number of operations is obtained by multiplying by 2.

8 *prefetch_instructions* (Non-Speculative)

Counts the number of committed prefetch instructions.

9 *SIMD_load_store_instructions* (Non-Speculative)

Counts the number of committed SIMD load/store instructions.

10 *SIMD_floating_instructions* (Non-Speculative)

Counts the number of committed SIMD floating-point instructions. The counted instructions are the same as *floating_instructions*. Two operations are executed per instruction; the number of operations is obtained by multiplying by 2.

-
- 11 *SIMD_fma_instructions* (Non-Speculative)
Counts the number of committed SIMD floating-point multiply-and-add instructions. The counted instructions are the same as *fma_instructions*. Four operations are executed per instruction; the number of operations is obtained by multiplying by 4.
 - 12 *sxar1_instructions* (Non-Speculative)
Counts the number of committed SXAR1 instructions.
 - 13 *sxar2_instructions* (Non-Speculative)
Counts the number of committed SXAR2 instructions.
 - 14 *trap_all* (Non-Speculative)
Counts the occurrences of all trap events. The number of occurrences counted equals the sum of the occurrences counted by all trap PA events.
 - 16 *trap_int_level* (Non-Speculative)
Counts the occurrences of *interrupt_level_n*.
 - 17 *trap_spill* (Non-Speculative)
Counts the occurrences of *spill_n_normal* and *spill_n_other*.
 - 18 *trap_fill* (Non-Speculative)
Counts the occurrences of *fill_n_normal* and *fill_n_other*.
 - 19 *trap_trap_inst* (Non-Speculative)
Counts the occurrences of *trap_instruction*.

Supplemental PA Events

- 23 *xma_inst* (Non-Speculative)
Counts the number of committed FPMADDX and FPMADDXHI instructions.
- 24 *unpack_sxar1* (Non-Speculative)
Counts the number of unpacked SXAR1 instructions that are committed.
- 25 *unpack_sxar2* (Non-Speculative)
Counts the number of unpacked SXAR2 instructions that are committed.
- 26 *instruction_flow_counts* (Non-Speculative)
Counts the number of committed instruction flows. On SPARC64™ X / SPARC64™ X+, some instructions are processed internally as several separate instructions, called instruction flows. This event does not count packed SXAR1 and SXAR2 instructions.
- 27 *ex_load_instructions* (Non-Speculative)
Counts the number of committed integer-load instructions. Counts the LD(S|U)B{A}, LD(S|U)H{A}, LD(S|U)W{A}, LDD{A}, and LDX{A} instructions.
- 28 *ex_store_instructions* (Non-Speculative)
Counts the number of committed integer-store and atomic instructions. Counts the STB{A}, STH{A}, STW{A}, STD{A}, STX{A}, LDSTUB{A}, SWAP{A}, and CAS{X}A instructions.

29 *fl_load_instructions* (Non-Speculative)

Counts the number of committed non-SIMD floating-point load instructions. Counts the LDF{A}, LDDF{A}, and LD{X}FSR instructions. This event does not count LDQF{A}

30 *fl_store_instructions* (Non-Speculative)

Counts the number of committed non-SIMD floating-point store instructions. Counts the STF{A}, STDF{A}, STFR, STDFR, and ST{X}FSR instructions. This event does not count STQF{A}.

31 *SIMD_fl_load_instructions* (Non-Speculative)

Counts the number of committed SIMD floating-point load instructions. Counted instructions are LDF{A} and LDDF{A}.

32 *SIMD_fl_store_instructions* (Non-Speculative)

Counts the number of committed SIMD floating-point store instructions. Counted instructions are STF{A}, STDF{A}, STFR, and STDFR.

33 *iwr_empty*

Counts the number of cycles that the IWR (Issue Word Register) is empty. The IWR is a four-entry register that holds instructions during instruction decode; the IWR may be empty if an instruction cache miss prevents instruction fetch.

34 *rs1* (Non-Speculative)

Counts the number of cycles in which normal execution is halted due to the following:

- a trap or interrupt
- update of privileged registers
- to guarantee memory ordering
- RAS-initiated hardware retry

35 *flush_rs* (Non-Speculative)

Counts the number of pipeline flushes due to branch misprediction. Since SPARC64™ X / SPARC64™ X+ support speculative execution, instructions that should not have been executed may be in flight. When it is determined that the predicted path is incorrect, these instructions are cancelled. A pipeline flush occurs at this time.

misprediction rate = $\text{flush_rs} / \text{branch_instructions}$

36 *0iid_use*

Counts the number of cycles where no instruction is issued. SPARC64™ X / SPARC64™ X+ issue up to four non-SXAR instructions per cycle; when no instruction is issued, *0iid_use* is incremented. On SPARC64™ X / SPARC64™ X+, some instructions are processed internally as several separate instructions, called instruction flows. Each of these instruction flows is counted. SXAR instructions are also counted.

37 *1iid_use*

Counts the number of cycles where one instruction is issued.

38 *2iid_use*

Counts the number of cycles where two instructions are issued.

39 *3iid_use*

Counts the number of cycles where three instructions are issued.

40 *4iid_use*

Counts the number of cycles where four instructions are issued.

41 *sync_intlk*

Counts the number of cycles where instruction issue is blocked by a pipeline sync.

42 *regwin_intlk*

Counts the number of cycles where instruction issue is blocked by a register window switch.

43 *decode_all_intlk*

Counts the number of cycles where instruction issue is blocked by a static interlock condition at the decode stage. *decode_all_intlk* includes *sync_intlk* and *regwin_intlk*; stall cycles due to dynamic conditions (such as reservation station full) are not counted.

44 *rsf_pmml* (Non-Speculative)

Counts the number of cycles where mixing single-precision and double-precision floating-point operations prevents instructions from issuing.

45 *toq_rsbr_phantom*

Counts the number of instructions that are predicted taken but are not actually branch instructions. Branch prediction on SPARC64™ X / SPARC64™ X+ is done prior to instruction decode; in other words, branch prediction occurs regardless of whether the instruction is actually a branch. Instructions that are not branch instructions may be incorrectly predicted as taken branches.

46 *op_stv_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed because the oldest, uncommitted instruction is a memory access waiting for data. *op_stv_wait* does not count cycles where a store instruction is waiting for data (atomic instructions are counted).

Note that *op_stv_wait* does not measure the cache-miss latency, since any cycles prior to becoming the oldest, uncommitted instruction are not included.

47 *op_stv_wait_nc_pend* (Non-Speculative)

Counts *op_stv_wait* for non-cacheable accesses.

48 *op_stv_wait_ex* (Non-Speculative)

Counts *op_stv_wait* for integer memory access instructions. Does not distinguish between L1 cache and L2 cache misses.

49 *op_stv_wait_sxmiss* (Non-Speculative)

Counts *op_stv_wait* caused by an L2\$ miss. Does not distinguish between integer and floating-point loads.

50 *op_stv_wait_sxmiss_ex* (Non-Speculative)

Counts *op_stv_wait* caused by an integer-load L2\$ miss.

51 *op_stv_wait_pfp_busy* (Non-Speculative)

Counts *op_stv_wait* caused by a memory access instruction that cannot be executed due to the lack of an available prefetch port.

52 *op_stv_wait_pfp_busy_ex* (Non-Speculative)

Counts *op_stv_wait* caused by an integer memory access instruction that cannot be executed due to the lack of an available prefetch port.

53 *op_stv_wait_swpf* (Non-Speculative)

Counts *op_stv_wait* caused by a prefetch instruction.

54 *op_stv_wait_pfp_busy_swpf* (Non-Speculative)

Counts *op_stv_wait* caused by a prefetch instruction that cannot be executed due to the lack of an available prefetch port.

55 *cse_window_empty_sp_full* (Non-Speculative)

Counts the number of cycles where no instructions are committed because the CSE is empty and the store ports are full.

56 *cse_window_empty* (Non-Speculative)

Counts the number of cycles where no instructions are committed because the CSE is empty.

57 *branch_comp_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed and the oldest, uncommitted instruction is a branch instruction. Measuring *branch_comp_wait* has a lower priority than measuring *eu_comp_wait*.

58 *eu_comp_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed and the oldest, uncommitted instruction is an integer or floating-point instruction. Measuring *eu_comp_wait* has a higher priority than measuring *branch_comp_wait*.

59 *fl_comp_wait* (Non-Speculative)

Counts the number of cycles where no instructions are committed and the oldest, uncommitted instruction is a floating-point instruction.

60 *0endop* (Non-Speculative)

Counts the number of cycles where no instructions are committed. *0endop* also counts cycles where the only instruction that commits is an SXAR instruction.

61 *1endop* (Non-Speculative)

Counts the number of cycles where one instruction is committed.

62 *2endop* (Non-Speculative)

Counts the number of cycles where two instructions are committed.

63 *3endop* (Non-Speculative)

Counts the number of cycles where three instructions are committed.

64 *suspend_cycle* (Non-Speculative)

Counts the number of cycles where the instruction unit is halted by a SUSPEND or SLEEP instruction.

65 *sleep_cycle* (Non-Speculative)

Counts the number of cycles where the instruction unit is halted by a SLEEP instruction

66 *single_sxar_commit* (Non-Speculative)

Counts the number of cycles where the only instruction committed is an unpacked SXAR instruction. These cycles are also counted by *0endop*.

67 *d_move_wait* (non-speculative)

Counts the number of cycles where no instructions are committed while waiting for the register window to be updated.

68 *cse_priority_wait*

Counts the number of cycles where no instructions are committed because the SMT thread is waiting for commit priority. On SPARC64™ X / SPARC64™ X+, only one thread can

commit instructions in a given cycle, and the priority is switched every cycle as long as the other thread is active. The event is counted only when there is an instruction ready to be committed for that thread.

- 69 *w_cse_window_empty* (non-speculative)
Number of cycles where *cse_window_empty* for the thread that has commit priority.
- 70 *w_eu_comp_wait* (non-speculative)
Number of cycles where *eu_comp_wait* for the thread that has commit priority.
- 71 *w_branch_comp_wait* (non-speculative)
Number of cycles where *branch_comp_wait* for the thread that has commit priority.
- 72 *w_op_stv_wait* (non-speculative)
Number of cycles where *op_stv_wait* for the thread that has commit priority.
- 73 *w_d_move_wait*
Number of cycles where *d_move_wait* for the thread that does not have commit priority.
- 74 *w_0endop* (non-speculative)
Number of cycles where *0endop* for the thread that does not have commit priority.
- 75 *w_op_stv_wait_nc_pend* (non-speculative)
Number of cycles where *op_stv_wait_nc_pend* for the thread that has commit priority.
- 76 *w_op_stv_wait_sxmiss* (non-speculative)
Number of cycles where *op_stv_wait_sxmiss* for the thread that has commit priority.
- 77 *w_op_stv_wait_sxmiss_ex* (non-speculative)
Number of cycles where *op_stv_wait_sxmiss_ex* for the thread that has commit priority.
- 78 *w_fl_comp_wait* (non-speculative)
Number of cycles where *fl_comp_wait* for the thread that has commit priority.
- 79 *w_cse_window_empty_sp_full* (non-speculative)
Number of cycles where *cse_window_empty_sp_full* for the thread that has commit priority.
- 80 *w_op_stv_wait_ex* (non-speculative)
Number of cycles where *op_stv_wait_ex* for the thread that has commit priority.
- 81 *only_this_thread_active*
Number of cycles while SMT is enabled where the CSE of this thread is not empty and the CSE of the other thread is empty.
- 82 *single_mode_cycle_counts*
Number of cycles that the thread is active in single-threaded mode (SMT disabled).
- 82 *single_mode_instructions*
Number of committed instructions in single-threaded mode (SMT disabled).
- 84 *both_threads_active*
Number of cycles while SMT is enabled where the CSEs of both threads are not empty.
- 85 *both_threads_empty*

Number of cycles where SMT is enabled where the CSEs of both threads are empty.

86 *both_threads_suspended*

Number of cycles where both threads in a core are in the suspended state.

11.2.2. MMU and L1 cache Events

Standard PA Events

1 *uITLB_miss*

Counts the occurrences of instruction uTLB misses.

2 *uDTLB_miss*

Counts the occurrences of data uTLB misses.

3 *L1I_miss*

Counts the occurrences of L1 instruction cache misses.

4 *L1D_miss*

Counts the occurrences of L1 data cache misses.

5 *L1I_wait_all*

Counts the total time spent processing L1 instruction cache misses (i.e., the total miss latency). On SPARC64™ X / SPARC64™ X+, the L1 cache is a non-blocking cache that can process multiple cache misses in parallel; *L1I_wait_all* only counts the miss latency for one of these misses. That is, the overlapped miss latencies are not counted.

6 *L1D_wait_all*

Counts the total time spent processing L1 data cache misses (i.e., the total miss latency). On SPARC64™ X / SPARC64™ X+, the L1 cache is a non-blocking cache that can process multiple cache misses in parallel; *L1D_wait_all* only counts the miss latency for one of these misses. That is, the overlapped miss latencies are not counted.

Supplemental PA Events

7 *ITLB_write*

Counts the number of ITLB writes caused by an instruction fetch ITLB miss.

8 *DTLB_write*

Counts the number of DTLB writes caused by a data access DTLB miss.

9 *swpf_success_all*

Counts the number of prefetch instructions that are not lost in the SU and are sent to the SX.

10 *swpf_fail_all*

Counts the number of prefetch instructions that are lost in the SU.

11 *swpf_lbs_hit*

Counts the number of prefetch instructions that hit in the L1 cache.
prefetch instructions sent to SU
= *swpf_success_all* + *swpf_fail_all* + *swpf_lbs_hit*

12 *L1I_thrashing*

Counts the occurrences of an L2 read request being issued twice in the period between acquiring and releasing a store port. When instruction fetch causes an L1 instruction cache miss, the requested data is updated in the L1I\$. This counter is incremented if the updated data is evicted before it can be read.

13 *L1D_thrashing*

Counts the occurrences of an L2 read request being issued twice in the period between acquiring and releasing a store port. When a memory access instruction causes an L1 data cache miss, the requested data is updated in the L1D\$. This counter is incremented if the updated data is evicted before it can be read.

14 *L1D_miss_dm*

Counts the occurrences of L1 data cache misses for a load/store instructions.

15 *L1D_miss_pf*

Counts the occurrences of L1 data cache misses for a prefetch instructions.

16 *L1D_miss_qpf*

Counts the occurrences of L1 data cache misses for hardware prefetch requests.

11.2.3. L2 cache Events

L2 cache events may be due to the actions of a VCPU, I/O or external requests. Events caused by VCPUs are counted separately for each VCPU; those caused by I/O or external requests are counted for all VCPUs.

Most L2 cache events are categorized as either demand (dm) or prefetch (pf) events, but these categories do not directly correspond to load/store/atomic and prefetch instructions, for the following reasons.

- When a load/store instruction cannot be executed due to a lack of resources needed to move data into the L1 cache, data is first moved into the L2 cache. Once L1 cache resources become available, the load/store instruction is executed. That is, the request to move data into the L2 cache is processed as a prefetch request.
- The hardware prefetch mechanism generates prefetch requests.
- L1 cache prefetch instructions are processed as demand requests.

Instead, demand and prefetch L2 cache events correspond to the following:

- A demand (dm) request to the L2 cache is an instruction fetch, load/store instruction, or L1 prefetch instruction that successfully acquired the resources needed to access memory.
- A prefetch (pf) request to the L2 cache is an instruction fetch, load/store instruction, or L1 prefetch instruction that could not acquire the resources needed to access memory; L2 prefetch instructions and hardware prefetch are also considered prefetch requests.

Standard PA Events

1 *L2_read_dm*

Counts the number of L2 cache references by demand requests.
References by external requests are not counted.

2 *L2_read_pf*

Counts the number of L2 cache references by prefetch requests.

3 *L2_miss_dm*

Counts the number of L2 cache misses caused by demand requests.

This counter is the sum of the *L2_miss_counts_dm_bank*{0,1,2,3}.

4 *L2_miss_pf*

Counts the number of L2 cache misses caused by prefetch requests.

This counter is the sum of the *L2_miss_counts_pf_bank*{0, 1, 2, 3}.

5 *L2_miss_counts_dm_bank* {0, 1, 2, 3}

Counts the number of L2 cache misses for each bank caused by demand requests.

When an L2 cache miss causes a prefetch request for an address to be issued and then a demand request for the same address is issued before the data is returned from memory or an external CPU, the demand request is not counted in *L2_miss_counts_dm_bank*{0,1,2,3}.

6 *L2_miss_counts_pf_bank* {0, 1, 2, 3}

Counts the number of L2 cache misses for each bank caused by prefetch requests.

7 *L2_miss_wait_dm_bank* {0, 1, 2, 3}

Counts the total time spent processing L2 cache misses for each bank caused by demand requests (i.e., the total miss latency for each bank). The latency of each memory access request is counted.

When an L2 cache miss causes prefetch request for an address to be issued and then a demand request for the same address is issued before the data is returned from memory or an external CPU, the cycles after the demand request but before the data is received are counted in *L2_miss_wait_dm_bank*{0,1,2,3}.

8 *L2_miss_wait_pf_bank* {0, 1, 2, 3}

Counts the total time spent processing L2 cache misses for each bank caused by prefetch requests, (i.e., the total miss latency for each bank). The latency of each memory access request is counted.

The L2 cache miss latency can be derived by summing *L2_miss_wait_** and dividing by the sum of *L2_miss_counts_**.

If individual L2 cache-miss latencies are calculated for pf/dm requests, the value obtained for the miss latency of dm requests may be higher than expected.

9 *L2_wb_dm*

Counts the occurrences of writeback to memory caused by L2 cache misses for demand requests.

10 *L2_wb_pf*

Counts the occurrences of writeback to memory caused by L2 cache misses for prefetch requests.

Supplemental PA Events

11 *lost_pf_pfp_full*

Counts the number of weak prefetch requests lost due to prefetch port full.

12 *lost_pf_by_abort*

Counts the number of weak prefetch requests lost due to L2-pipe abort.

11.2.4. Bus Transaction Events

Standard PA Events

1 *cpu_mem_read_counts*

Counts the number of memory read requests issued by the CPU.
For this event, the same value is counted by all VCPUs.

2 *cpu_mem_write_counts*

Counts the number of memory write requests issued by the CPU.
For this event, the same value is counted by all VCPUs.

3 *IO_mem_read_counts*

Counts the number of memory read requests issued by I/O.
For this event, the same value is counted by all VCPUs.

4 *IO_mem_write_counts*

Counts the number of memory write requests issued by I/O.
Only IO-FST is counted by this event. IO-PST can be counted using *IO_pst_counts*.
For this event, the same value is counted by all VCPUs.

5 *bi_counts*

Counts the number of external cache-invalidate requests received by the CPU chip.
Cache-invalidate requests caused by internal IO-FST/PST requests are also counted by this event.
These requests do not check the cache data before invalidating.
For this event, the same value is counted by all VCPUs.

6 *cpi_counts*

Counts the number of external cache-copy-and-invalidate requests received by the CPU chip.
These requests copy updated cache data to memory before invalidating; cache data that is consistent with memory does not need to be copied and is invalidated.
For this event, the same value is counted by all VCPUs.

7 *cpb_counts*

Counts the number of external cache-copyback requests received by the CPU chip.
These request copy updated cache data to memory.
For this event, the same value is counted by all VCPUs.

8 *cpd_counts*

Counts the number of internal or external IO cache-read requests (DMA read requests) received by the CPU chip.
For this event, the same value is counted by all VCPUs.

Supplemental PA Events

9 *IO_pst_counts*

Counts the number of memory write requests (IO-PST) issued by I/O.

11.3. Cycle Accounting

Cycle accounting is a method used for analyzing performance bottlenecks. The total time (number of CPU cycles) required to execute an instruction sequence can be divided into time spent in various CPU execution states (executing instructions, waiting for a memory access, waiting for execution to complete, and so on).

SPARC64™ X / SPARC64™ X+ define a large number of PA events that record detailed information about CPU execution states, enable efficient analysis of bottlenecks, and are useful for performance tuning.

In this document, cycle accounting is specifically defined as the analysis of instructions as they are committed in order. SPARC64™ X / SPARC64™ X+ execute instructions out-of-order and have multiple execution units; the CPU is generally in a state where executing and waiting instructions are thoroughly mixed together. One instruction may be waiting for data from memory, another executing a floating-point multiply, and yet another waiting for confirmation of the branch direction. Simply analyzing the reasons why individual instructions are waiting is not useful. Instead, cycle accounting classifies cycles by the number of instructions committed; when a cycle commits no instructions, the conditions that prevented instructions from committing are analyzed.

SPARC64™ X / SPARC64™ X+ commits up to 4 instructions per cycle. The more cycles that commit the maximum number of instructions, the better the execution efficiency. Cycles that do not commit any instructions have an extremely negative effect on performance, so it is important to perform a detailed analysis of these cycles. The main causes are:

- Waiting for a memory access to return data.
- Waiting for instruction execution to complete.
- Instruction fetch is unable to supply the pipeline with instructions.

Table 11-3 highlights some useful PA events and describes how these PA events can be used to analyze execution efficiency.

Figure 11-1 shows the relationship between the various *op_stv_wait_** events. The PA events marked with a † in the table and in the figure are synthetic events, which are calculated from other PA events.

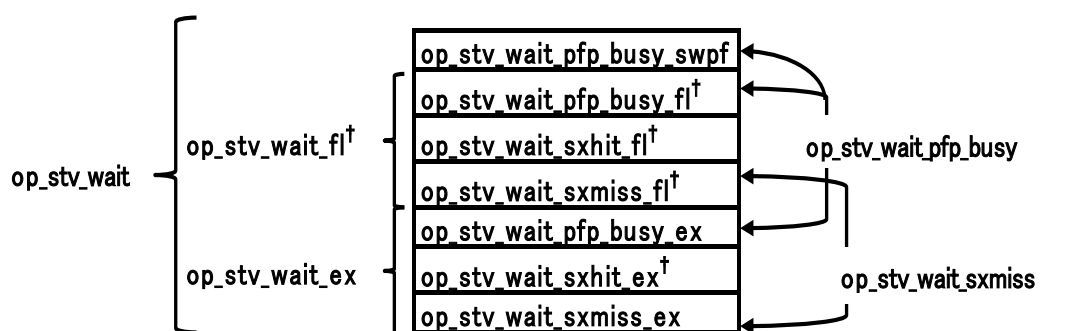


Figure 11-1 Breakdown of *op_stv_wait*

Table 11-3 Useful Performance Events for Cycle Accounting

Instructions Committed per Cycle	Cycles	Remarks
4	<i>cycle_counts</i>	N/A (maximum instructions committed)

	- <i>3endop</i> - <i>2endop</i> - <i>1endop</i> - <i>0endop</i>	
3	<i>3endop</i>	
2	<i>2endop</i>	
1	<i>1endop</i>	
0	Execution: <i>eu_comp_wait</i> + <i>branch_comp_wait</i>	<i>eu_comp_wait</i> = <i>ex_comp_wait</i> + <i>fl_comp_wait</i>
	Instruction Fetch: <i>cse_window_empty</i>	<i>cse_window_empty</i> = <i>cse_window_empty_sp_full</i> + <i>sleep_cycle</i> + misc.†
	L1D cache miss: <i>op_stv_wait</i> - L2 cache miss (see below)	
	L2 cache miss: <i>op_stv_wait_sxmiss</i> + <i>op_stv_wait_nc_pend</i>	
	Others: <i>0endop</i> - <i>op_stv_wait</i> - <i>cse_window_empty</i> - <i>eu_comp_wait</i> - <i>branch_comp_wait</i> - (<i>instruction_flow_counts</i> - <i>instruction_counts</i>)	

12. Traps

12.1. Virtual Processor Privilege Modes

When a trap occurs, the privilege level is changed depending on the trap. Refer to Table 12-2 and Table 12-3 for details.

For a VCPU running at a higher privilege level, if the exception would lower the current privilege level, the trap will be pending until the privilege level is lower than the target privilege level of the exception. Refer to Section 12.1 in UA2011 for the relationship between privileged level and TL.

The possible values of the following registers depend on the privilege level. Refer to Section 5.7.7 and Section 5.7.9 in UA2011 respectively for details.

- TL
- GL

12.5. Trap list and priorities

Symbol	Description
-x-	Trap will not occur in this mode.
P	Change to privileged mode.
P(ie)	Change to privileged mode if PSTATE.ie = 1.
H	Changes to hyperprivileged mode.

Table 12-2 Trap list, by TT value

TT	Trap name	Type	Priority	Privilege level after the traps occurs	Definition
000 ₁₆	<i>reserved</i>	—	—	—	—
006 ₁₆	<i>reserved</i>	—	—	—	—
007 ₁₆	<i>reserved</i>	—	—	—	—
008 ₁₆	<i>IAE_privilege_violation</i>	precise	3.1	H	305
00B ₁₆	<i>IAE_unauth_access</i>	precise	2.7	H	306

TT	Trap name	Type	Priority	Privilege level after the traps occurs	Definition
00C ₁₆	<i>IAE_nfo_page</i>	precise	3.3	H	305
00D ₁₆	<i>reserved</i>	—	—	—	—
00E ₁₆	<i>reserved</i>	—	—	—	—
00F ₁₆	<i>reserved</i>	—	—	—	—
010 ₁₆	<i>illegal_instruction</i>	precise	6.2	H	306
011 ₁₆	<i>privileged_opcode</i>	precise	7	P	308
012 ₁₆	<i>reserved</i>	—	—	—	—
013 ₁₆	<i>reserved</i>	—	—	—	—
014 ₁₆	<i>DAE_invalid_asl</i>	precise	12.1	H	302
015 ₁₆	<i>DAE_privilege_violation</i>	precise	12.5	H	303
016 ₁₆	<i>DAE_nc_page</i>	precise	12.6	H	303
017 ₁₆	<i>DAE_nfo_page</i>	precise	12.7	H	303
018 ₁₆ -01F ₁₆	<i>reserved</i>	—	—	—	—
020 ₁₆	<i>fp_disabled</i>	precise	8	P	305
021 ₁₆	<i>fp_exception_ieee_754</i>	precise	11.1	P	305
022 ₁₆	<i>fp_exception_other</i>	precise	11.1	P	305
023 ₁₆	<i>tag_overflow</i>	precise	14	P	309
024 ₁₆	<i>clean_window</i>	precise	10.1	P	302
025 ₁₆ -027 ₁₆	<i>reserved</i>	—	—	—	—
028 ₁₆	<i>division_by_zero</i>	precise	15	P	304
029 ₁₆	<i>reserved</i>	—	—	—	—
02C ₁₆	<i>reserved</i>	—	—	—	—
02D ₁₆	<i>reserved</i>	—	—	—	—
02E ₁₆	<i>reserved</i>	—	—	—	—
02F ₁₆	<i>reserved</i>	—	—	—	—
030 ₁₆	<i>DAE_side_effect_page</i>	precise	12.7	H	304
033 ₁₆	<i>reserved</i>	—	—	—	—
034 ₁₆	<i>mem_address_not_aligned</i>	precise	10.2	H	307
035 ₁₆	<i>LDDF_mem_address_not_aligned</i>	precise	10.1	H	307
036 ₁₆	<i>STDF_mem_address_not_aligned</i>	precise	10.1	H	309
037 ₁₆	<i>privileged_action</i>	precise	11.1	H	308
038 ₁₆	<i>reserved</i>	—	—	—	—
039 ₁₆	<i>reserved</i>	—	—	—	—
03C ₁₆	<i>reserved</i>	—	—	—	—
03D ₁₆	<i>reserved</i>	—	—	—	—

TT	Trap name	Type	Priority	Privilege level after the traps occurs	Definition
041 ₁₆ -04F ₁₆	<i>interrupt_level_n</i> (n = 1 - 15) (<i>interrupt_level_15</i> same as <i>pic_overflow</i>)	disrupting	32-n ⁱ	P(ie)	307
050 ₁₆ -05D ₁₆	<i>reserved</i>	—	—	—	—
061 ₁₆	<i>PA_watchpoint</i> (<i>RA_watchpoint</i>)	precise	12.9	H	307
062 ₁₆	<i>VA_watchpoint</i>	precise	11.2	H	310
065 ₁₆ -067 ₁₆	<i>reserved</i>	—	—	—	—
069 ₁₆ -06B ₁₆	<i>reserved</i>	—	—	—	—
06D ₁₆ -070 ₁₆	<i>reserved</i>	—	—	—	—
073 ₁₆	<i>illegal_action</i>	precise	8.5	H	306
074 ₁₆	<i>control_transfer_instruction</i>	precise	11.1	P	302
075 ₁₆	<i>reserved</i>	—	—	—	—
078 ₁₆ -07B ₁₆	<i>reserved</i>	—	—	—	—
07C ₁₆	<i>cpu_mondo</i>	disrupting	16.8	P(ie)	302
07D ₁₆	<i>dev_mondo</i>	disrupting	16.11	P(ie)	304
07E ₁₆	<i>resumable_error</i>	disrupting	33.3	P(ie)	308
07F ₁₆	<i>nonresumable_error</i> (not generated by hardware)		—	—	307
080 ₁₆ -09C ₁₆	<i>spill_n_normal</i> (n = 0 - 7)	precise	9	P	309
0A0 ₁₆ -0BC ₁₆	<i>spill_n_other</i> (n = 0 - 7)	precise	9	P	309
0C0 ₁₆ -0DC ₁₆	<i>fill_n_normal</i> (n = 0 - 7)	precise	9	P	304
0E0 ₁₆ -0FC ₁₆	<i>fill_n_other</i> (n = 0 - 7)	precise	9	P	304
100 ₁₆ -17F ₁₆	<i>trap_instruction</i>	precise	16.2	P	309

Table 12-3 Trap list, by priority

TT	Trap name	Type	Priority	Privilege level after the trap occurs	Definition
00B ₁₆	<i>IAE_unauth_access</i>	precise	2.7	H	306
008 ₁₆	<i>IAE_privilege_violation</i>	precise	3.1	H	305
00C ₁₆	<i>IAE_nfo_page</i>	precise	3.3	H	305
010 ₁₆	<i>illegal_instruction</i>	precise	6.2	H	306
011 ₁₆	<i>privileged_opcode</i>	precise	7	P	308

ⁱ In UA2011, the priorities of *interrupt_level_15* and *pic_overflow* are different. On SPARC64 X / SPARC64 X+, both have a priority of 17.

TT	Trap name	Type	Priority	Privilege level after the trap occurs	Definition
020 ₁₆	<i>fp_disabled</i>	precise	8	P	305
073 ₁₆	<i>illegal_action</i>	precise	8.5	H	306
080 ₁₆ -09C ₁₆	<i>spill_n_normal</i> (n = 0 - 7)	precise	9	P	309
0A0 ₁₆ -0BC ₁₆	<i>spill_n_other</i> (n = 0 - 7)	precise	9	P	309
0C0 ₁₆ -0DC ₁₆	<i>fill_n_normal</i> (n = 0 - 7)	precise	9	P	304
0E0 ₁₆ -0FC ₁₆	<i>fill_n_other</i> (n = 0 - 7)	precise	9	P	304
024 ₁₆	<i>clean_window</i>	precise	10.1	P	302
035 ₁₆	<i>LDDF_mem_address_not_aligned</i>	precise	10.1	H	307
036 ₁₆	<i>STDF_mem_address_not_aligned</i>	precise	10.1	H	309
034 ₁₆	<i>mem_address_not_aligned</i>	precise	10.2	H	307
021 ₁₆	<i>fp_exception_ieee_754</i>	precise	11.1	P	305
022 ₁₆	<i>fp_exception_other</i>	precise	11.1	P	305
037 ₁₆	<i>privileged_action</i>	precise	11.1	H	308
074 ₁₆	<i>control_transfer_instruction</i>	precise	11.1	P	302
062 ₁₆	<i>VA_watchpoint</i>	precise	11.2	H	310
014 ₁₆	<i>DAE_invalid_asl</i>	precise	12.1	H	302
015 ₁₆	<i>DAE_privilege_violation</i>	precise	12.5	H	303
016 ₁₆	<i>DAE_nc_page</i>	precise	12.6	H	303
017 ₁₆	<i>DAE_nfo_page</i>	precise	12.7	H	303
030 ₁₆	<i>DAE_side_effect_page</i>	precise	12.7	H	304
061 ₁₆	<i>PA_watchpoint</i> (<i>RA_watchpoint</i>)	precise	12.9	H	307
023 ₁₆	<i>tag_overflow</i>	precise	14	P	309
028 ₁₆	<i>division_by_zero</i>	precise	15	P	304
100 ₁₆ -17F ₁₆	<i>trap_instruction</i>	precise	16.2	P	309
07C ₁₆	<i>cpu_mondo</i>	disrupting	16.8	P(ie)	302
07D ₁₆	<i>dev_mondo</i>	disrupting	16.11	P(ie)	304
041 ₁₆ -04F ₁₆	<i>interrupt_level_n</i> (n = 1 - 15) (<i>interrupt_level_15</i> same as <i>pic_overflow</i>)	disrupting	32·n ⁱⁱ	P(ie)	307
07E ₁₆	<i>resumable_error</i>	disrupting	33.3	P(ie)	308
07F ₁₆	<i>nonresumable_error</i> (not by hardware)		—	—	307

ⁱⁱ In UA2011, the priorities of *interrupt_level_15* and *pic_overflow* are different. On SPARC64 X / SPARC64 X+, both have a priority of 17.

12.5.1. Trap Descriptions

Refer to Section 12.7 in UA2011.

12.5.1.2. *clean_window*

TT	024 ₁₆ – 027 ₁₆
Priority	10.1
Trap category	precise
Privilege level transition	priv

Compatibility Note JPS1 and UA2011 allow hardware to clean the register windows (Impl. Dep. #102), but SPARC64™ X / SPARC64™ X+ generate the exception so that the windows can be cleaned by software.

12.5.1.3. *control_transfer_instruction*

TT	074 ₁₆
Priority	11.1
Trap category	precise
Privilege level transition	priv

The control transfer instruction exception occurs in the following conditions.

- Conditional branch instructions (Bicc, BPcc, BPr, FBfcc, FBPfcc, CBcond) that are taken
- Unconditional branch instructions (BA, BPA, FBA, FBPA)
- CALL, JMPL, RETURN, DONE, and RETRY instructions
- TCC instructions that are taken

12.5.1.4. *cpu_mondo*

TT	07C ₁₆
Priority	16.8
Trap category	disrupting
Privilege level transition	priv (if PSTATE.ie = 1)

This exception occurs when PSTATE.ie = 1 and the head of the CPU_MONDO queue is not the same as the tail.

12.5.1.5. *DAE_invalid_asl*

TT	014 ₁₆
Priority	12.1
Trap category	precise
Privilege level transition	hpriv

12.5.1.6. *DAE_nc_page*

TT	016 ₁₆
Priority	12.6
Trap category	precise
Privilege level transition	hpriv

This exception occurs when a non-cacheable space is accessed by an atomic load-store instruction, LDTXA, LDBLOCKF, a SIMD load instruction, or a SIMD store instruction.

Compatibility Note STPARTIALF does not generate this exception.

12.5.1.7. *DAE_nfo_page*

TT	017 ₁₆
Priority	12.7
Trap category	precise
Privilege level transition	hpriv

This exception occurs when a page (TTE.nfo = 1) marked for access only by nonfaulting loads is accessed by any instruction except the following.

- Load instructions that specify ASI_PRIMARY_NO_FAULT{_LITTLE} or ASI_SECONDARY_NO_FAULT{_LITTLE}
- PREFETCH and PREFETCHA

In other word, this exception occurs for the following instructions.

- Load instructions that do not specify ASI_PRIMARY_NO_FAULT{_LITTLE} or ASI_SECONDARY_NO_FAULT{_LITTLE}
- Store and atomic load-store instructions with any ASI
- FLUSH instructions

Note When ASI_PRIMARY_NO_FAULT{_LITTLE} and ASI_SECONDARY_NO_FAULT{_LITTLE} are specified for store and atomic load-store instructions, *DAE_invalid_asl* is generated.

12.5.1.8. *DAE_privilege_violation*

TT	015 ₁₆
Priority	12.5
Trap category	precise
Privilege level transition	hpriv

Note FLUSH and PREFETCH{A} do not generate *DAE_privilege_violation*.

12.5.1.9. *DAE_side_effect_page*

TT	030 ₁₆
Priority	12.7
Trap category	precise
Privilege level transition	hpriv

12.5.1.16. *dev_mondo*

TT	07D ₁₆
Priority	16.11
Trap category	disrupting
Privilege level transition	priv (if PSTATE.ie = 1)

12.5.1.17. *division_by_zero*

TT	028 ₁₆
Priority	15
Trap category	precise
Privilege level transition	priv

12.5.1.22. *fill_n_normal, fill_n_other*

TT	0C0 ₁₆ , 0C4 ₁₆ , 0C8 ₁₆ , 0CC ₁₆ , 0D0 ₁₆ , 0D4 ₁₆ , 0D8 ₁₆ , 0DC ₁₆ , 0E0 ₁₆ , 0E4 ₁₆ , 0E8 ₁₆ , 0EC ₁₆ , 0F0 ₁₆ , 0F4 ₁₆ , 0F8 ₁₆ , 0FC ₁₆
Priority	9
Trap category	precise
Privilege level transition	priv

12.5.1.23. *fp_disabled*

TT	020 ₁₆
Priority	8
Trap category	precise
Privilege level transition	priv

12.5.1.24. *fp_exception_ieee_754*

TT	021 ₁₆
Priority	11.1
Trap category	precise
Privilege level transition	priv

Refer to FSR (page 26) regarding the trap enable mask for these exceptions.

12.5.1.25. *fp_exception_other*

TT	022 ₁₆
Priority	11.1
Trap category	precise
Privilege level transition	priv

Refer to Section 8, "IEEE Std. 754-1985 Requirements for SPARC-V9" (page 265).

12.5.1.29. *IAE_nfo_page*

TT	00C ₁₆
Priority	3.3
Trap category	precise
Privilege level transition	hpriv

12.5.1.30. *IAE_privilege_violation*

TT	008 ₁₆
Priority	3.1
Trap category	precise
Privilege level transition	hpriv

If instructions are fetched in non-privileged mode and TL > 0, this exception is generated. In this case, because the exception is detected independently of the MMU settings, the priority is different than the value shown in Table 12-2

12.5.1.31. *IAE_unauth_access*

TT	00B ₁₆
Priority	2.7
Trap category	precise
Privilege level transition	hpriv

12.5.1.32. *Illegal_action*

TT	073 ₁₆
Priority	8.5
Trap category	precise
Privilege level transition	hpriv

This exception occurs when the instruction is not XAR eligible but XAR.v = 1, or the instruction is XAR eligible but the XAR settings are not correct. If XAR is set by SXAR, this exception occurs when the instruction modified by SXAR is executed.

While the *illegal_instruction* exception has higher priority, in some cases where either *illegal_instruction* or *illegal_action* could be generated, WRASR and FSHIFTORX will generate an *illegal_action* exception. Refer to page 189 for details regarding WRASR and pages 252 and 250 for details regarding FSHIFTORX.

12.5.1.33. *Illegal_instruction*

TT	010 ₁₆
Priority	6.2
Trap category	precise
Privilege level transition	hpriv

12.5.1.41. *interrupt_level_n* (n = 1 - 15)

TT	041 ₁₆ – 04F ₁₆
Priority	17 – 31 (32 – n)
Trap category	disrupting
Privilege level transition	priv (if PSTATE.ie = 1 and PIL < n)

If $PIL < 14$, setting `SOFTINT.sm = 1` or `SOFTINT.tm = 1` generates an *interrupt_level_14* exception.

interrupt_level_15 has the same priority and trap number as *pic_overflow* which is generated when a PA counter overflows.

12.5.1.43. *LDDF_mem_address_not_aligned*

TT	035 ₁₆
Priority	10.1
Trap category	precise
Privilege level transition	hpriv

This exception occurs when an address accessed by a non-SIMD LDDF or LDDFA is word aligned but not doubleword aligned.

12.5.1.44. *mem_address_not_aligned*

TT	034 ₁₆
Priority	10.2
Trap category	precise
Privilege level transition	hpriv

12.5.1.45. *nonresumable_error*

TT	07F ₁₆
Priority	—
Trap category	—
Privilege level transition	undetected

12.5.1.47. *PIC_overflow*

TT	04F ₁₆
Priority	17
Trap category	disrupting
Privilege level transition	priv (PSTATE.ie = 1 and PIL < 15)

This exception occurs when a PA counter overflows and the overflow exception is not masked. The priority and trap number is the same as *interrupt_level_15* (page 306) on SPARC64™ X / SPARC64™ X+.

12.5.1.49. *privileged_action*

TT	037 ₁₆
Priority	11.1
Trap category	precise
Privilege level transition	hpriv

A *privileged_action* exception is generated for cases where a privilege level violation cannot be determined solely from the opcode and PSTATE settings. For example,

- An attempt to use an ASI number that is not available at that privilege level.
- An attempt to access registers (such as TICK, STICK, PIC, PCR) that are configured to prevent non-privileged access.

12.5.1.50. *privileged_opcode*

TT	011 ₁₆
Priority	7
Trap category	precise
Privilege level transition	priv

A *privileged_opcode* exception is generated for cases where the privilege level violation can be determined solely from the opcode and the PSTATE.priv setting. This exception is also generated if an instruction is executed in non-privileged mode and the executed opcode, while valid, cannot be executed when TL = 0.

12.5.1.53. *resumable_error*

TT	07E ₁₆
Priority	33.3
Trap category	disrupting
Privilege level transition	priv (if PSTATE.ie = 1)

This exception occurs when `PSTATE.ie = 1` and the head of the `RESUMABLE_ERROR` queue is not the same as the tail.

12.5.1.56. *spill_n_normal, spill_n_other*

TT	080 ₁₆ , 084 ₁₆ , 088 ₁₆ , 08C ₁₆ , 090 ₁₆ , 094 ₁₆ , 098 ₁₆ , 09C ₁₆ 0A0 ₁₆ , 0A4 ₁₆ , 0A8 ₁₆ , 0AC ₁₆ , 0A0 ₁₆ , 0A4 ₁₆ , 0A8 ₁₆ , 0AC ₁₆
Priority	9
Trap category	precise
Privilege level transition	priv

12.5.1.57. *STDF_mem_address_not_aligned*

TT	036 ₁₆
Priority	10.1
Trap category	precise
Privilege level transition	hpriv

This exception occurs when an address accessed by a non-SIMD STDF, STDFA, or STDFR is word aligned but not doubleword aligned.

12.5.1.58. *tag_overflow*

TT	023 ₁₆
Priority	14
Trap category	precise
Privilege level transition	priv

12.5.1.59. *trap_instruction*

TT	100 ₁₆ – 17F ₁₆
Priority	16.2
Trap category	precise
Privilege level transition	priv

12.5.1.62. *VA_watchpoint*

TT	062 ₁₆
Priority	11.2
Trap category	precise
Privilege level	hpriv
transition	

12.5.2. Special cases for priority

When multiple exceptions occur, generally the exception with the highest priority shown in Table 12-2 is chosen, and a trap is generated for that exception. However, in some special cases an exception with lower priority can be chosen. These special cases are described below.

- The priority of *illegal_action* is 8.5, but in some cases it takes precedence over *illegal_instruction*, which has a priority of 6.2. Refer to *WRASR* (page 189) and to *FSHIFTORX* (page 222, 252) for details.
- The *privileged_opcode* exception, not the *illegal_instruction* exception, is generated when an instruction is executed in non-privileged mode and the opcode, while valid, cannot be executed when TL = 0. Such instructions include *DONE*, *RETRY*, *RDPR*, and *WRPR*.

13. Memory Management Unit

This chapter provides information about the SPARC64™ X / SPARC64™ X+ Memory Management Unit. It describes the internal architecture of the MMU and how to program it.

13.1. Address types

The SPARC64™ X / SPARC64™ X+ MMUs support a 64-bit virtual address (VA) space (no VA hole) and a 48-bit real address space.

- VA(Virtual Address) Access to a virtual address is protected at the granularity of a page. A VA is 64 bits, and all 64 bits are available on SPARC64™ X / SPARC64™ X+ (no VA hole). It is identified by a context number.
- RA(Real Address) All 64 bits of an RA are valid for software, but only 48 bits are valid for hardware.

Refer to Section 14.1 in UA2011 for information on Virtual-to-Real Translation

Table 13-1 SPARC64™ X / SPARC64™ X+ address widths

	VA	RA
Address width	64 bits	64 bits
Legal address width	64 bits (no VA hole)	48 bits

13.4. TSB Translation Table Entry (TTE)

A TSB TTE contains the VA to RA translation for a single page mapping.

TTE Tag

context_id	—	va<63:22>
63	48 47 42 41	0

TTE Data

v	nfo	soft2	taddr<55:13>	ie	e	cp	cv	p	ep	w	soft	size
63	62 61	56 55		13	12 11 10	9 8	7 6	5	4 3	0		

Table 13-2 TSB TTE

Bit	Field	Description
Tag 63:48	context_id	
Tag 41:0	va<63:22>	
Data 63	v	
Data 62	nfo	
Data 61:56	soft2	

Data 55:13	taddr<55:13>	Target address (RA). On SPARC64™ X / SPARC64™ X+, if bits taadr<55:48> are not zero, an <i>invalid_TSB_entry</i> exception is generated.
Data 12	ie	The ie bit in the IMMU is ignored.
Data 11	e	
Data 10	cp	The cp bit is ignored on SPARC64™ X / SPARC64™ X+.
Data 9	cv	The cv bit is ignored on SPARC64™ X / SPARC64™ X+.
Data 8	p	
Data 7	ep	
Data 6	w	
Data 5:4	soft	
Data 3:0	size	Page size of this entry, encoded as shown in the table below.

Size<3:0>	Page size
0000	8KB
0001	64KB
0010	<i>reserved</i>
0011	4MB
0100	<i>reserved</i>
0101	256MB
0110-1111	<i>reserved</i>

13.6. Context Registers

SPARC64™ X / SPARC64™ X+ support a pair of primary context registers and a pair of secondary context registers per strand. These context registers are shared by the I- and D-MMUs. The size of the context ID field is 16 bits. Primary Context 0 and Primary Context 1 are the primary context registers. There is a hit in the TLB if a TLB entry for a translating primary ASI matches the context field of either Primary Context 0 or Primary Context 1. Secondary Context 0 and Secondary Context 1 are the secondary context registers. There is a hit in the TLB if a TLB entry for a translating secondary ASI matches the context field of either Secondary Context 0 or Secondary Context 1.

Compatibility Note In JPS1, a 13-bit context ID and a 51-bit VA are defined. In UA2011, a 16-bit context ID is defined.

Table 13-3 shows the usage of the context registers for the I-MMU and D-MMU.

Table 13-3 I-MMU and D-MMU Context Register Usage

TL	ASI	Instruction fetch	Data access
0	—	primary	primary
	ASI_PRIMARY*, ASI_{BLOCK PST* FL* WRBK XFILL}_PRIMARY*, ASI_{TWINX STBI}_P*, ASI_BLOCK_COMMIT_PRIMARY	N/A	primary
	ASI_SECONDARY*, ASI_{BLOCK PST* FL* WRBK XFILL}_SECONDARY*, ASI_{TWINX STBI}_S*, ASI_BLOCK_COMMIT_SECONDARY	N/A	secondary

13.8. Page sizes

SPARC64™ X / SPARC64™ X+ support four page sizes: 8 KB, 64 KB, 4 MB, and 256 MB. The TLBs can hold translations for all four sizes concurrently.

Table 13-4 Page types supported on SPARC64™ X / SPARC64™ X+

Page type	Virtual page number	Offset in page	Encoding
8KB page	51 bits	13 bits	000 ₂
64KB page	48 bits	16 bits	001 ₂
4MB page	42 bits	22 bits	011 ₂
256MB page	36 bits	28 bits	101 ₂

14. Opcode Maps

This chapter contains the instruction opcode maps for SPARC64™ X / SPARC64™ X+.

Opcodes marked with an em dash ‘—’ are reserved; an attempt to execute a reserved opcode shall cause an exception (*illegal_instruction*).

In this chapter, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in Table 7-1 (page 42).

Table 14-1 **op<1:0>**

op<1:0>			
0	1	2	3
Branches and SETHI Refer to Table 14-2	CALL	Arithmetic & Miscellaneous Refer to Table 14-3	Memory access instructions Refer to Table 14-4

Table 14-2 **Branches, SETHI, and SXAR (op<1:0> = 0)**

op2<2:0>							
0	1	2	3	4	5	6	7
ILLTRAP	BPcc Refer to Table 14-8	BiCC ^D Refer to Table 14-8	BPr Refer to Table 14-9	SETHI, NOP	FBPfcc Refer to Table 14-8	FBfcc ^D Refer to Table 14-8	SXAR1, SXAR2

Table 14-3

Arithmetic & Miscellaneous (op<1:0> = 2)

op3<3:0>	op3<5:4>			
	0	1	2	3
0	ADD	ADDcc	TADDcc	WRY ^D (rd = 0) WRCCR (rd = 2) WRASI (rd = 3) WRFPRS (rd = 6) WRPCR ^{P_{PCR}} (rd = 16) WRPIC ^{P_{PIC}} (rd = 17) WRGSR (rd = 19) WRTICK_CMPR ^P (rd = 23) WRPAUSE ⁺ (rd = 27) WRXAR (rd = 29) WRXASR (rd = 30)
1	AND	ANDcc	TSUBcc	
2	OR	ORcc	TADDccTV ^D	
3	XOR	XORcc	TSUBccTV ^D	
4	SUB	SUBcc	MULScc ^D	FPop1 (Refer to Table 14-5, Table 14-6)
5	ANDN	ANDNcc	SLL (x = 0, r = 0), SLLX (x = 1, r = 0), ROLX (x = 1, r = 1)	FPop2 (Refer to Table 14-7)
6	ORN	ORNcc	SRL (x = 0), SRLX (x = 1)	IMPDEP1 (Refer to Table 14-13)
7	XNOR	XNORcc	SRA (x = 0), SRAX (x = 1)	IMPDEP2 (Refer to Table 14-16)
8	ADDC	ADDCcc	RDY ^D (rs1 = 0, i = 0) RDCCR (rs1 = 2, i = 0) RDASI (rs1 = 3, i = 0) RDTICK ^{P_{NPT}} (rs1 = 4, i = 0) RDPC (rs1 = 5, i = 0) RDFPRS (rs1 = 6, i = 0) MEMBAR (rs1 = 15, rd = 0, i = 1) RDPCR ^{P_{PCR}} (rs1 = 16, i = 0) RDPIC ^{P_{PIC}} (rs1 = 17, i = 0) RDGSR (rs1 = 19, i = 0) RDSTICK (rs1 = 24, i = 0) RDXASR (rs1 = 30, i = 0)	JMPL
9	MULX	—		RETURN
A ₁₆	UMUL ^D	UMULcc ^D		Tcc
B ₁₆	SMUL ^D	SMULcc ^D	FLUSHW	FLUSH
C ₁₆	SUBC	SUBCcc	MOVcc	SAVE
D ₁₆	UDIVX	—	SDIVX	RESTORE
E ₁₆	UDIV ^D	UDIVcc ^D	POPC (rs1 = 0)	
F ₁₆	SDIV ^D	SDIVcc ^D	MOVR (rs1 = 0)	—

Table 14-4 Memory access instructions (op<1:0> = 3)

op3<3:0>	op3<5:4>			
	0	1	2	3
0	LDUW	LDUWA ^{PASI}	LDF	LDFA ^{PASI}
1	LDUB	LDUBA ^{PASI}	LDFSR ^D (rd = 0) LDXFSR (rd = 1) LDXEFSR ⁺ (rd = 3)	—
2	LDUH	LDUHA ^{PASI}	LDQF	LDQFA ^{PASI}
3	LDTW ^D (rd even)	LDTWA ^{D,PASI} (rd even) LDTXA (rd even)	LDDF	LDDFA ^{PASI} LDBLOCKF LDSHORTF
4	STW	STWA ^{PASI}	STF	STFA ^{PASI}
5	STB	STBA ^{PASI}	STFSR ^D (rd = 0) STXFSR (rd = 1)	—
6	STH	STHA ^{PASI}	STQF	STQFA ^{PASI}
7	STTW ^D (rd even)	STTWA ^{D,PASI} (rd even) STBI ^N XFILL ^N	STDF	STDFA ^{PASI} STBLOCKF STPARTIALF STSHORTF XFILL ^N
8	LDSW	LDSWA ^{PASI}	—	—
9	LDSB	LDSBA ^{PASI}	—	—
A ₁₆	LDSH	LDSHA ^{PASI}	—	—
B ₁₆	LDX	LDXA ^{PASI}	—	—
C ₁₆	—	—	STFR	CASA ^{PASI}
D ₁₆	LDSTUB	LDSTUBA ^{PASI}	PREFETCH	PREFETCHA ^{PASI}
E ₁₆	STX	STXA ^{PASI} STBI ^N XFILL ^N	—	CASXA ^{PASI}
F ₁₆	SWAP ^D	SWAPA ^{D,PASI}	STDFR	—

Table 14-5 FPop1 (op<1:0> = 2, op3 = 34₁₆) (1/2)

opf<8:4>	opf<3:0>							
	0	1	2	3	4	5	6	7
00 ₁₆	—	FMOVs	FMOVd	FMOVq	—	FNEGs	FNEGd	FNEGq
01 ₁₆	—	—	—	—	—	—	—	—
02 ₁₆	—	—	—	—	—	—	—	—
03 ₁₆	—	—	—	—	—	—	—	—
04 ₁₆	—	FADDs	FADDd	FADDq	—	FSUBs	FSUBd	FSUBq
05 ₁₆	—	FNADDs ⁺	FNADDd ⁺	—	—	—	—	—
06 ₁₆	—	—	—	—	—	—	—	—
07 ₁₆	—	—	—	—	—	—	—	—
08 ₁₆	—	FST0x	FdT0x	FqT0x	FxT0s	—	—	—
09 ₁₆	—	—	—	—	—	—	—	—
0A ₁₆	—	—	—	—	—	—	—	—
0B ₁₆	—	—	—	—	—	—	—	—
0C ₁₆	—	—	—	—	FiT0s	—	FdT0s	FqT0s
0D ₁₆	—	FST0i	FdT0i	FqT0i	—	—	—	—
0E ₁₆ · 1F ₁₆	—	—	—	—	—	—	—	—

Table 14-6 FPop1 (op<1:0> = 2, op3 = 34₁₆) (2/2)

opf<8:4>	opf<3:0>							
	8	9	A ₁₆	B ₁₆	C ₁₆	D ₁₆	E ₁₆	F ₁₆
00 ₁₆	—	FABSs	FABSd	FABSq	—	—	—	—
01 ₁₆	—	—	—	—	—	—	—	—
02 ₁₆	—	FSQRTs	FSQRTd	FSQRTq	—	—	—	—
03 ₁₆	—	—	—	—	—	—	—	—
04 ₁₆	—	FMULs	FMULd	FMULq	—	FDIVs	FDIVd	FDIVq
05 ₁₆	—	FNMULs ⁺	FNMULd ⁺	—	—	—	—	—
06 ₁₆	—	FsMULd	—	—	—	—	FdMULq	—
07 ₁₆	—	FNSMULd ⁺	—	—	—	—	—	—
08 ₁₆	FxT0d	—	—	—	FxT0q	—	—	—
09 ₁₆	—	—	—	—	—	—	—	—
0A ₁₆	—	—	—	—	—	—	—	—
0B ₁₆	—	—	—	—	—	—	—	—
0C ₁₆	FiT0d	FST0d	—	FqT0d	FiT0q	FST0q	FdT0q	—
0D ₁₆	—	—	—	—	—	—	—	—
0E ₁₆ · 1F ₁₆	—	—	—	—	—	—	—	—

Table 14-7 FPop2 (op<1:0> = 2, op3 = 35₁₆)

opf<8:4>	opf<3:0>								
	0	1	2	3	4	5	6	7	8·F ₁₆
00 ₁₆	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	(Reserved variation of FMOV _R)			—
01 ₁₆	—	—	—	—	—	—	—	—	—
02 ₁₆	—	—	—	—	—	FMOV _R sZ ⁱⁱⁱ	FMOV _R dZ ⁱⁱⁱ	FMOV _R qZ ⁱⁱⁱ	—

ⁱⁱⁱ iw<13> = 0

opf<8:4>	opf<3:0>							
03 ₁₆	—	—	—	—	—	—	—	—
04 ₁₆	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVRsLEZ ⁱⁱⁱ	FMOVRdLEZ ⁱⁱⁱ	FMOVRqLEZ ⁱⁱⁱ
05 ₁₆	—	FCMPs	FCMPd	FCMPq	—	FCMPes ⁱⁱⁱ	FCMPed ⁱⁱⁱ	FCMPEq ⁱⁱⁱ
06 ₁₆	—	—	—	—	—	FMOVRsLZ ⁱⁱⁱ	FMOVRdLZ ⁱⁱⁱ	FMOVRqLZ ⁱⁱⁱ
07 ₁₆	—	—	—	—	—	—	—	—
08 ₁₆	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	(Reserved variation of FMOVR)		—
09 ₁₆	—	—	—	—	—	—	—	—
0A ₁₆	—	—	—	—	—	FMOVRsNZ ⁱⁱⁱ	FMOVRdNZ ⁱⁱⁱ	FMOVRqNZ ⁱⁱⁱ
0B ₁₆	—	—	—	—	—	—	—	—
0C ₁₆	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVRsGZ ⁱⁱⁱ	FMOVRdGZ ⁱⁱⁱ	FMOVRqGZ ⁱⁱⁱ
0D ₁₆	—	—	—	—	—	—	—	—
0E ₁₆	—	—	—	—	—	FMOVRsGEZ ⁱⁱⁱ	FMOVRdGEZ ⁱⁱⁱ	FMOVRqGEZ ⁱⁱⁱ
0F ₁₆	—	—	—	—	—	—	—	—
10 ₁₆	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—
11 ₁₆ -17 ₁₆	—	—	—	—	—	—	—	—
18 ₁₆	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—
19 ₁₆ -1F ₁₆	—	—	—	—	—	—	—	—

Table 14-8 **cond<3:0>**

cond<3:0>	BPcc op = 0 op2 = 1	Bicc op = 0 op2 = 2	FBPfcc op = 0 op2 = 5	FBfcc op = 0 op2 = 6	Tcc op = 2 op3 = 3A ₁₆
0 ₁₆	BP _N	BN ^D	FBP _N	FB _N ^D	T _N
1 ₁₆	BP _E	BE ^D	FBP _{NE}	FB _{NE} ^D	T _E
2 ₁₆	BP _{LE}	BL _E ^D	FBP _{LG}	FB _{LG} ^D	T _{LE}
3 ₁₆	BP _L	BL ^D	FBP _{UL}	FB _{UL} ^D	T _L
4 ₁₆	BP _{LEU}	BL _{EU} ^D	FBP _L	FB _L ^D	T _{LEU}
5 ₁₆	BP _{CS}	BC _S ^D	FBP _{UG}	FB _{UG} ^D	T _{CS}
6 ₁₆	BP _{NEG}	BN _{EG} ^D	FBP _G	FB _G ^D	T _{NEG}
7 ₁₆	BP _{VS}	BV _S ^D	FBP _U	FB _U ^D	T _{VS}
8 ₁₆	BP _A	BA ^D	FBP _A	FB _A ^D	T _A
9 ₁₆	BP _{NE}	BN _E ^D	FBP _E	FB _E ^D	T _{NE}
A ₁₆	BP _G	BG ^D	FBP _{UG}	FB _{UG} ^D	T _G
B ₁₆	BP _{GE}	BG _E ^D	FBP _{GE}	FB _{GE} ^D	T _{GE}
C ₁₆	BP _{GU}	BG _U ^D	FBP _{UGE}	FB _{UGE} ^D	T _{GU}
D ₁₆	BP _{CC}	BC _C ^D	FBP _{LE}	FB _{LE} ^D	T _{CC}
E ₁₆	BP _{POS}	BP _{OS} ^D	FBP _{ULE}	FB _{ULE} ^D	T _{POS}
F ₁₆	BP _V C	BV _C ^D	FBP _O	FB _O ^D	T _V C

Table 14-9 rcond<2:0>

rcond<2:0>	BPr op = 0 op2 = 3 iw<28> = 0	CBcond op = 0 op2 = 3 iw<28> = 1	MOVr op = 2 op2 = 2F ₁₆	FMOVr op = 2 op2 = 35 ₁₆
0	—	—	—	—
1	BRZ	C{W X}B{NE E} ⁺	MOVRZ	FMOVr{s d q}Z
2	BRLEZ	C{W X}B{G LE} ⁺	MOVRLEZ	FMOVr{s d q}LEZ
3	BRLZ	C{W X}B{GE L} ⁺	MOVRLZ	FMOVr{s d q}LZ
4	—	C{W X}B{GU LEU} ⁺	—	—
5	BRNZ	C{W X}B{CC CS} ⁺	MOVRNZ	FMOVr{s d q}NZ
6	BRGZ	C{W X}B{POS NEG} ⁺	MOVRGZ	FMOVr{s d q}GZ
7	BRGEZ	C{W X}B{VC VS} ⁺	MOVRGEZ	FMOVr{s d q}GEZ

Table 14-10 cc, opf_cc (MOVcc, FMOVcc)

cc2	cc1	cc0	Condition code used
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc
1	0	1	—
1	1	0	xcc
1	1	1	—

Table 14-11 cc Fields (FBPfcc, FCMP, FCMPE, FLCMP and FPCMP)

cc1	cc0	Condition code used
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

Table 14-12 cc Fields (BPcc and Tcc)

cc1	cc0	Condition code used
0	0	icc
0	1	—
1	0	xcc
1	1	—

Table 14-13 IMPDEP1: VIS instructions (op<1:0> = 2, op3 = 36₁₆) (1/3)

opf<3:0>	opf<8:4>							
	00 ₁₆	01 ₁₆	02 ₁₆	03 ₁₆	04 ₁₆	05 ₁₆	06 ₁₆	07 ₁₆
0 ₁₆	EDGE8	ARRAY8	FPCMPLE16	—	—	FPADD16	FZERO	FAND
1 ₁₆	EDGE8N	—	—	FMUL8x16	—	FPADD16 S	FZEROS	FANDS

opf<3:0	opf<8:4>							
2 ₁₆	EDGE8L	ARRAY16	FPCMPNE16 FPCMPUNE16	—	FPADD64 ⁺	FPADD32	FNOR	FXNOR
3 ₁₆	EDGE8LN	—	—	FMUL8x16AU	—	FPADD32S	FNORS	FXNORS
4 ₁₆	EDGE16	ARRAY32	FPCMPLE32	—	—	FPSUB16	FANDNOT2	FSRC1
5 ₁₆	EDGE16N	—	—	FMUL8x16AL	—	FPSUB16S	FANDNOT2S	FSRC1S
6 ₁₆	EDGE16L	—	FPCMPNE32 FPCMPUNE32	FMUL8sUx16	FPSUB64 ⁺	FPSUB32	FNOT2	FORNOT2
7 ₁₆	EDGE16LN	LZD	—	FMUL8uLx16	—	FPSUB32S	FNOT2S	FORNOT2S
8 ₁₆	EDGE32	ALIGNADDRES	FPCMPGT16	FMULD8sUx16	FALIGNDATA	—	FANDNOT1	FSRC2
9 ₁₆	EDGE32N	BMASK	—	FMULD8uLx16	—	—	FANDNOT1S	FSRC2S
A ₁₆	EDGE32L	ALIGNADDRES _LITTLE	FPCMPQE16 FPCMPUEQ16	FPACK32	—	—	FNOT1	FORNOT1
B ₁₆	EDGE32LN	—	—	FPACK16	FPMERGE	—	FNOT1S	FORNOT1S
C ₁₆	—	—	FPCMPGT32	—	BSHUFFLE	—	FXOR	FOR
D ₁₆	—	—	—	FPACKFIX	FEXPAND	—	FXORS	FORS
E ₁₆	—	—	FPCMPQE32 FPCMPUEQ32	PDIST	—	—	FNAND	FONE
F ₁₆	—	—	—	—	—	—	FNANDS	FONES

Table 14-14 IMPDEP1: VIS instructions (op<1:0> = 2, op3 = 36₁₆) (2/3)

opf<3:0>	opf<8:4>							
	08 ₁₆	09 ₁₆	0A ₁₆	0B ₁₆	0C ₁₆	0D ₁₆	0E ₁₆	0F ₁₆
0 ₁₆	SHUTDOWN	FAESEN CX	FADDtd	FADDod	FCMPLE16X FPCMPLE16X ⁺	FCMPLE8X FPCMPLE8X ⁺	—	—
1 ₁₆	SIAM	FAESDECX	FSUBtd	FSUBod	FUCMPLE16X FPCMPULE16X ⁺	FUCMPLE8X FPCMPULE8X ⁺	—	—
2 ₁₆		FAESENCLX	FMULTd	FMULod	—	—	—	—
3 ₁₆	SLEEP	FAESDECLX	FDIVtd	FDIVod	FUCMPNE16X FPCMPUNE16X ⁺	FUCMPNE8X FPCMPUNE8X ⁺	—	—
4 ₁₆	—	FAESKEYX	FCMPtd	FCMPod	FCMPLE32X FPCMPLE32X ⁺	FCMPLE64X FPCMPLE64X ⁺	FPMAX 32X ⁺	—
5 ₁₆	SDIAM	FPSELMOV8 X ⁺	FCMPetd	—	FUCMPLE32X FPCMPULE32X ⁺	FUCMPLE64X FPCMPULE64X ⁺	FPMAX U32X ⁺	—
6 ₁₆	—	FPSELMOV1 6X ⁺	FQUAtd	FQUAod	—	—	FPMIN 32X ⁺	—
7 ₁₆	—	FPSELMOV3 2X ⁺	—	FRQUAod	FUCMPNE32X FPCMPUNE32X ⁺	FUCMPNE64X FPCMPUNE64X ⁺	FPMIN U32X ⁺	—
8 ₁₆	—	FDESEN CX	—	FXADDodLO	FCMPGT16X FPCMPGT16X ⁺	FCMPGT8X FPCMPGT8X ⁺	—	—
9 ₁₆	PADD32	FDESPC1X	—	FXADDodHI	FUCMPGT16X FPCMPUGT16X ⁺	FUCMPGT8X FPCMPUGT8X ⁺	—	—
A ₁₆	—	FDESI PX	—	FXMULodLO	—	—	—	—
B ₁₆	—	FDESI PX	—	—	FUCMPEQ16X FPCMPUEQ16X ⁺	FUCMPEQ8X FPCMPUEQ8X ⁺	—	—
C ₁₆	—	FDESKEYX	FbuxT0td	—	FCMPGT32X FPCMPGT32X ⁺	FCMPGT64X FPCMPGT64X ⁺	FPMAX 64X ⁺	—
D ₁₆	—	—	FtdT0bux	—	FUCMPGT32X FPCMPUGT32X ⁺	FUCMPGT64X FPCMPUGT64X ⁺	FPMAX U64X ⁺	—
E ₁₆	—	—	FbsxT0td	FodT0td	—	—	FPMIN 64X ⁺	—
F ₁₆	—	FPADD128X HI ⁺	FtdT0bsx	FtdT0od	FUCMPEQ32X FPCMPUEQ32X ⁺	FUCMPEQ64X FPCMPUEQ64X ⁺	FPMIN U64X ⁺	—

Table 14-15 IMPDEP1: VIS instructions (op<1:0> = 2, op3 = 36₁₆) (3/3)

opf<3:0>	opf<8:4>								
	10 ₁₆	11 ₁₆	12 ₁₆	13 ₁₆	14 ₁₆	15 ₁₆	16 ₁₆	17 ₁₆	18 ₁₆ -1F ₁₆
0 ₁₆	—	—	FPCMPULE8 ⁺	—	—	—	FCMPEQd	FMAXd	—
1 ₁₆	—	—	—	—	—	FLCMPs ⁺	FCMPEQs	FMAXs	—
2 ₁₆	—	—	FPCMPUNE8 ⁺	—	—	FLCMPd ⁺	FCMPEQEd	FMInd	—
3 ₁₆	—	—	—	—	—	—	FCMPEQEs	FMINs	—
4 ₁₆	FPCMP64X ⁺	—	—	—	—	—	FCMPLEEd	FRCPA d	—
5 ₁₆	FPCMPU64X ⁺	—	—	—	—	—	FCMPLEEs	FRCPA s	—
6 ₁₆	FPSLL64X ⁺	—	—	—	—	—	FCMPLTEd	FRSQRTAd	—
7 ₁₆	FPSRL64X ⁺	—	—	—	—	—	FCMPLTEs	FRSQRTAs	—
8 ₁₆	—	MOVxT0d ⁺	FPCMPUGT8 ⁺	—	—	—	FCMPNEd	FTRISSEld	—
9 ₁₆	—	MOVwT0s ⁺	—	—	—	—	FCMPNEs	—	—
A ₁₆	—	—	FPCMPUEQ8 ⁺	—	—	—	FCMPNEEd	FTRISMULd	—

B ₁₆	—	—	—	—	—	—	FCMPNEEs	—	—
C ₁₆	—	—	—	—	—	—	FCMPGTed	FEXPad	—
D ₁₆	—	—	—	—	—	—	FCMPGTES	—	—
E ₁₆	—	—	—	—	—	—	FCMPGEEd	—	—
F ₁₆	FPSRA64X ⁺	—	—	—	—	—	FCMPGEES	—	—

Table 14-16 IMPDEP2: (op<1:0> = 2, op3 = 37₁₆)

size	var			
	0	1	2	3
0 ₁₆	FPMADDX	FPMADDXHI	FTRIMADDd	FSELMOVd
1 ₁₆	FMADDs	FMSUBs	FNMSUBs	FNMADDs
2 ₁₆	FMADDd	FMSUBd	FNMSUBd	FNMADDd
3 ₁₆	—	—	FSHIFTORX	FSELMOVs

15. Assembly Language Syntax

15.1. Notation Used

15.1.1. Other Operand Syntax

The syntax for software traps has been changed from JPS1 Commonality. The updated syntax is shown below.

software_trap_number

Can be any of the following:

reg_{rs1} (equivalent to $reg_{rs1} + \%g0$)

$reg_{rs1} + imm8$

$reg_{rs1} - imm8$

$imm8$ (equivalent to $\%g0 + imm8$)

$imm8 + reg_{rs1}$ (equivalent to $reg_{rs1} + imm8$)

$reg_{rs1} + reg_{rs2}$

Here, *imm8* is a signed immediate constant that can be represented in 8 bits. The resulting operand value (software trap number) must be in the range 0 – 255, inclusive.

15.2. HPC-ACE Notation

When an instruction is executed, the value of the XAR register determines whether the instruction uses any HPC-ACE features. Generally, these features are specified by combining an arithmetic instruction with SXAR. This section defines the assembly language syntax for specifying HPC-ACE features.

HPC-ACE extends the instruction definitions to support the use of HPC-ACE floating-point registers, SIMD execution, and hardware prefetch disable. While the SXAR instructions fully specify whether these features are used, the following notation is defined to facilitate easy reading of the assembly language:

- (1) SXAR is written as *sxar1* or *sxar2*. These instructions have no arguments.
- (2) HPC-ACE floating-point registers are specified directly as arguments of the instruction.
- (3) Other HPC-ACE features are specified by appending suffixes to the instruction mnemonic.

-
- (4) The HPC-ACE features for a particular instruction are always specified by the closest preceding **SXAR** instruction. Another **SXAR** instruction in a sequence that branches to a point between an instruction and its corresponding **SXAR** never specifies features for that instruction.

An **SXAR** instruction must be placed 1 or 2 instructions before any instruction that uses the notation described in items (2) and (3). There are cases where the assembler cannot automatically determine that an **SXAR** needs to be inserted for an instruction that uses HPC-ACE features; thus, **SXAR** instructions cannot be omitted.

Whether a label can be inserted between an **SXAR** instruction and the instruction(s) that it modifies is not defined, as item (4) clearly defines which **SXAR** instruction specifies the HPC-ACE feature(s).

15.2.1. Suffixes for HPC-ACE Extensions

A comma (,) is placed after the instruction mnemonic, and the alphanumeric character(s) that immediately follow the comma specify various HPC-ACE features. These suffixes are shown in Table 15-1.

Table 15-1 Suffixes for HPC-ACE Extensions

XAR Notation	Suffix
XAR.simd	s
XAR.dis_hw_pf	d
XAR.negate_mul	n
XAR.rs1_copy	c

Suffixes are not case-sensitive. When two or more suffixes are specified, the suffixes may be specified in any order.

Example: SIMD instruction, HPC-ACE registers

```
sxar2
fadd %f0, %f2, %f510      /* HPC-ACE register specified, non-SIMD */
fadd,s %f0, %f2, %f4      /* SIMD, extended operation uses HPC-ACE
registers */
```

Example 2: SIMD load

```
sxar1
ldd,s [%i1], %f0
```