FUJITSU

SPARC64™ XII Specification

Distribution: Public
Privilege Levels: Nonprivileged

Ver 20
2017/10/06

Fujitsu Limited

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

Contents

Preface

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

This specification refers to the following documents.

- Oracle SPARC Architecture 2011. Draft D1.0.0, Jan 2016.
  http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf
  This document is referred to as UA2011.
- SPARC64™ VIIIfx Extensions Ver 15, 26 Apr. 2010
  http://img.jp.fujitsu.com/downloads/jp/jhpc/sparc64viiifx-extensions.pdf
  This document is referred to as SPARC64™ VIIIfx Extensions.
- SPARC64™ X/X+ specification ver.29, 27 Jan. 2015

  http://www.fujitsu.com/global/Images/SPARC64X_Xplus_Specification_v29.pdf
  This document is referred to as the SPARC64™ X/X+ specification.
- 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
  This document is referred to 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, 1000_2).
- Spaces may be inserted in long binary or hex numbers (for example, $1000\ 0000_{16}$) 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 with the pipe must be selected. For example, `FMUL{s|d}` expands to `FMULs` and `FMULd`. An empty charater string makes the alternatives inside the braces optional. For example, `F{|N}sMULd` is equivalent to `F{N}sMULd`.
  - The * and n symbols indicate a 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*. And *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 string formats <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 a future expansion is expected and 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 attribute

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

Table 1-1 Access attribute

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

1.1.5. Informational Notes

This document contains several different types of informational notes.

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

Note Notes provide general information.

Programming Note Programming notes provide information for writing software.

2. Definitions

- CPUID :

  A CPUID is the unique logical ID of a strand in a system. The CPUID contains the logical system board ID (LSBID), physical processor ID (chip ID) within a system board, last level cache and core unit ID (LCU ID), Core ID, and SMT ID.

- LCU :

  L3 cache is divided into four blocks. One of the L3 blocks (and three cores corresponding to that L3 block) is called as LCU in this specification.

- Virtual Processor (VCPU) :

  A virtual processor (refer to Chapter 2 of UA2011). SPARC64™ XII has eight VCPUs per physical CPU core.

3. Architecture Overview

Feature

- HPC-ACE and 8-SMT are supported.
- VA is 64bits wide and has no hole bit.
- RA is normally 64 bits wide.
- Instructions only on the local ROM can be executed for noncacheable space.

Present parameter

- 12 cores (chip) and 8-SMT (core)
- L1 instruction cache : 64KB/4way (core) ; L1 data cache : 64KB/8way (core) ; line size of L1 cache memories: 128 bytes.
- Unified L2 cache : 512KB/16way (core); line size of L2 cache memories: 128 bytes.
- Unified L3 cache : 8MB/16way (LCU); line size of L3 cache memories: 128bytes.
- For main TLB, set-associative TLB only. Instruction : 2,048 entries/16way (core); data : 2,048 entries/16way (core); page size : 6 sizes (8KB, 64KB, 4MB, 256MB, 2GB, and 16GB).

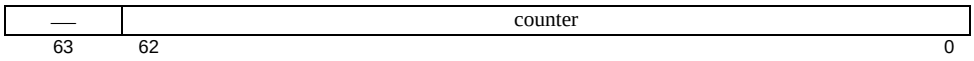
4. Data Formats

Refer to the SPARC64 X/X+ specification.

5. Register

5.5. Ancillary State Registers

5.5.4. Tick (TICK) Register (ASR 4)



Bit	Field	Access	Description
62:0	counter	R	TICK counter

The counter field of the TICK register is a 63-bit counter (SPARC V9 Impl. Dep. #105b) that counts the processor clock cycles. Reading TICK<63> returns 0.

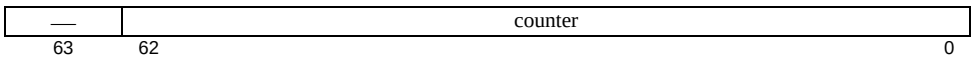
Nonprivileged software can read the TICK register using the RDTICK instruction but only if nonprivileged access to the TICK register is enabled. If nonprivileged access is disabled, an attempt by nonprivileged software to read the TICK register causes a *privileged_action* exception.

Table 5-1 shows the exceptions generated by reading or writing the TICK register.

Table 5-1 exceptions by reading or writing the TICK register

RDTICK	(WRTICK does not exist)	RDPR	WRPR
OK (if nonprivileged access is enabled) *privileged_action* (if nonprivileged access is disabled)	—	*privileged_opcode*	*privileged_opcode*

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



Bit	Field	Access	Description
62:0	counter	R	Elapsed time value

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. Reading STICK<63> returns 0.

Nonprivileged software can read the STICK register by using the RDSTICK instruction, but only if nonprivileged access to STICK register is enabled. If nonprivileged access is disabled, an attempt by nonprivileged software to read the STICK register causes a *privileged_action* exception.

Table 5-2 shows the exceptions generated by reading or writing the STICK register.

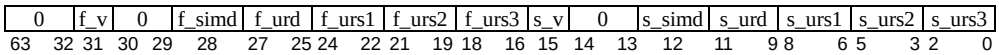
Table 5-2 exceptions by reading or writing the STICK register

RDSTICK	WRSTICK
OK (if nonprivileged access is enabled) *privileged_action* (if nonprivileged access is disabled)	*illegal_instruction* (different from TICK register)

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

A read of the STICK<62:0> register returns 63-bit data.

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



Bit	Field	Access	Description
31	f_v	RW	Indicates whether the contents of the 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 the 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 instruction fields. 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.

Aliases of the XAR field in this specification

The fields described in Table 5-3 have the following aliases.

Table 5-3 Aliases 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.f_xar_i	XAR.f_urs3<2>	For Fsimm8
XAR.s_xar_i	XAR.s_urs3<2>	For Fsimm8

XAR operation

Only some instructions can reference the XAR register. In this document, instructions that can reference XAR are called "XAR-eligible instructions".

- 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.

    Floating-point registers are referenced by 9-bit register numbers with the XAR fields specifying 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" (in the SPARC64™ X / X+ specification) for details.
  - XAR.urs2<2:1> and XAR.urs3<1:0> fields may be specified to use the newly implemented instructions in SPARC64™ XII (refer to page 18).
  - The XAR.urs3<1> field may be specified to disable hardware prefetch for integer and floating-point load/store instructions.
  - The XAR.urs3<2> field may be specified to use an 8-bit signed immediate value (Fsimm8) for some IMPDEP1 instructions (refer to page 18).
  - The XAR.urs3<2> and XAR.urd<2> fields can be specified for SIMD FMA instructions.
  - 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, XAR.s_urs1, XAR.s_urs2, XAR.s_urs3 and XAR.s_urd fields are used.
- The value 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 for the following cases.

- An attempt to execute an instruction that is not XAR-eligible while XAR.v = 1.
- XAR.simd = 1 for an instruction (including integer arithmetic) that does not support SIMD execution.
- XAR.urs1 ≠ 0 is specified for an instruction that does not use rs1. The same applies for rd.
- XAR.urs2 ≠ 0 is specified for an instruction that does not use rs2 and for an instcution whose rs2 field holds an immediate value (such as simm13 or fc*n*).
- XAR urs3 ≠ 0 is specified for an instruction that does not use rs3 except for the following.
  - XAR urs3<1:0> is specified to use the newly implemented instructions for SPARC64™ XII.
  - XAR urs3<1> is specified to disable hardware prefetch for integer and floating-point load/store instructions.
  - XAR urs3<2> is specified to use Fsimm8 for some IMPDEP1 instructions.
- XAR urs1<1> ≠ 0 is specified when XAR.urs1 is used as the upper 3 bits of the concatenated floating-point register. The same applies for XAR.urs2, XAR.urs3 and XAR.urd.
- A register number greater than or equal to F[256] is specified for the rd field of the FDIV{S|D} or FSQRT{S|D} instruction.
- XAR.simd = 1, and a register number greater than or equal to F[256] is specified. Some instructions (such as F{N}MADD{s|d}, F{N}MSUB{s|d}, and FAES*X) are exceptions to this rule and register numbers greater than or equal to F[256] can be speficied. Refer to the specification for each instruction.
- An attempt to execute STFRUW, STDFRDS, STDFRDW, FMONTMUL, and FMONTSQR while XAR.v = 0.

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

> **Programming Note** If WRXAR is used, either XAR.f_v or XAR.s_v can be set to 1. SXAR1 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 the 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 the execution is undefined.

XAR.urs2 (extended for SPARC64™ XII)

In SPARC64™ XII, XAR.urs2<2:1> can be specified to use the newly implemented instructions for SPARC64™ XII, such as LDFUW, LDFSW, LDDFDS, STFUW, and STDFDS. Refer to page 37 and page 48.

XAR.urs3 (extended for SPARC64™ XII)

In SPARC64™ XII, XAR.urs3 can be specified for the purpose stated below.

1) To use Fsimm8 by XAR.urs3<2>

2) To use the new instructions implemented in SPARC64™ XII by XAR.urs3<1:0>

In this specification, XAR.urs3<2> is also described as XAR.xar_i. If XAR.xar_i = 1, Fsimm8 is used instead of F[rs2] for some IMPDEP1 instructions. Fsimm8 is an 8-bit signed immediate value and used as a 64-bit immediate data (refer to page 27).

All IMPDEP1 instructions that can use XAR.urs3 for the purpose stated above are shown in Table 5-4.

There are some new IMPDEP1 instructions in SPARC64™ XII which have the same opecodes in SPARC64™ X/SPARC64™ X+. XAR.urs3<1:0> is specified to use those new instructions in SPARC64™ XII. Refer to page 84, page 89, page 95, page 98, and page 100.

Table 5-4 Instructions that can use XAR.urs3.

Instruction	XAR.urs3<2> (Fsimm8)	XAR.urs3<1:0> (specifying the new instruction in SPARC64™ XII)	Format of Fsimm8	Page
F{SLL|SRL|SRA}32	✓		Fsimm8_32x2	page 69
FP{ADD|SUB}64	✓		Fsimm8_64x1	
FPMERGE	✓		Fsimm8_8x8	page 35
FPMUL64	✓		Fsimm8_64x1	page 71
FPMUL32	✓		Fsimm8_32x2	page 71
FPADD16{|S}	✓		Fsimm8_16x4	
FPADD32{|S}	✓		Fsimm8_32x2	
FPSUB16{|S}	✓		Fsimm8_16x4	
FPSUB32{|S}	✓		Fsimm8_32x2	
FNORS	✓		Fsimm8_32x2	
FNOR	✓		Fsimm8_64x1	
FANDNOT{1|2}S	✓		Fsimm8_32x2	
FANDNOT{1|2}	✓		Fsimm8_64x1	
FNOT2S	✓		Fsimm8_32x2	
FNOT2	✓		Fsimm8_64x1	
FXORS	✓		Fsimm8_32x2	
FXOR	✓		Fsimm8_64x1	
FNANDS	✓		Fsimm8_32x2	
FNAND	✓		Fsimm8_64x1	
FANDS	✓		Fsimm8_32x2	
FAND	✓		Fsimm8_64x1	
FXNORS	✓		Fsimm8_32x2	
FXNOR	✓		Fsimm8_64x1	
FORNOT{1|2}S	✓		Fsimm8_32x2	
FORNOT{1|2}	✓		Fsimm8_64x1	
FSRC2S	✓		Fsimm8_32x2	
FSRC2	✓		Fsimm8_64x1	

FORS	✓		Fsimm8_32x2	
FOR	✓		Fsimm8_64x1	
FPSELMOV8FX	✓	✓	Fsimm8_8x8	page 95
FPSELMOV16FX	✓	✓	Fsimm8_16x4	page 95
FPSELMOV32FX	✓	✓	Fsimm8_32x2	page 95
FPSELMOV8X	✓		Fsimm8_8x8	
FPSELMOV16X	✓		Fsimm8_16x4	
FPSELMOV32X	✓		Fsimm8_32x2	
FPCSL8X	✓		Fsimm8_64x1	page 82
FPADD128XHI	✓		Fsimm8_64x1	
FPCMP{LE|GT}4X	✓		Fsimm8_8x8	page 63
FPCMP{LE|GT}8X	✓		Fsimm8_8x8	page 63
FPCMP{LE|GT}16X	✓		Fsimm8_16x4	page 63
FPCMP{LE|GT}32X	✓		Fsimm8_32x2	page 63
FPCMP{LE|GT}64X	✓		Fsimm8_64x1	page 63
FPCMPU{EQ|NE|LE|GT}4X	✓		Fsimm8_8x8	page 63
FPCMPU{EQ|NE|LE|GT}8X	✓		Fsimm8_8x8	page 63
FPCMPU{EQ|NE|LE|GT}16X	✓		Fsimm8_16x4	page 63
FPCMPU{EQ|NE|LE|GT}32X	✓		Fsimm8_32x2	page 63
FPCMPU{EQ|NE|LE|GT}64X	✓		Fsimm8_64x1	page 63
FPCMP{LE|GT}4FX	✓	✓	Fsimm8_8x8	page 84
FPCMP{LE|GT}8FX	✓	✓	Fsimm8_8x8	page 84
FPCMP{LE|GT}16FX	✓	✓	Fsimm8_16x4	page 84
FPCMP{LE|GT}32FX	✓	✓	Fsimm8_32x2	page 84
FPCMP{LE|GT}64FX	✓	✓	Fsimm8_64x1	page 84
FPCMPU{EQ|NE|LE|GT}4FX	✓	✓	Fsimm8_8x8	page 84
FPCMPU{EQ|NE|LE|GT}8FX	✓	✓	Fsimm8_8x8	page 84
FPCMPU{EQ|NE|LE|GT}16FX	✓	✓	Fsimm8_16x4	page 84
FPCMPU{EQ|NE|LE|GT}32FX	✓	✓	Fsimm8_32x2	page 84
FPCMPU{EQ|NE|LE|GT}64FX	✓	✓	Fsimm8_64x1	page 84
FPCMP{LE|GT}4XACC	✓	✓	Fsimm8_8x8	page 89
FPCMP{LE|GT}8XACC	✓	✓	Fsimm8_8x8	page 89
FPCMP{LE|GT}16XACC	✓	✓	Fsimm8_16x4	page 89
FPCMP{LE|GT}32XACC	✓	✓	Fsimm8_32x2	page 89
FPCMP{LE|GT}64XACC	✓	✓	Fsimm8_64x1	page 89
FPCMPU{EQ|NE|LE|GT}4XACC	✓	✓	Fsimm8_8x8	page 89
FPCMPU{EQ|NE|LE|GT}8XACC	✓	✓	Fsimm8_8x8	page 89
FPCMPU{EQ|NE|LE|GT}16XACC	✓	✓	Fsimm8_16x4	page 89

FPCMPU{EQ|NE|LE|GT}32XACC	✓	✓	Fsimm8_32x2	page 89
FPCMPU{EQ|NE|LE|GT}64XACC	✓	✓	Fsimm8_64x1	page 89
FP{MAX|MIN}{|U}32X	✓		Fsimm8_32x2	
FP{MAX|MIN}{|U}64X	✓		Fsimm8_64x1	
F{S|Z}EXTW	✓		Fsimm8_64x1	page 72
FPCMP{|U}64X	✓		Fsimm8_64x1	
FP{SLL|SRL|SRA}64X	✓		Fsimm8_64x1	
FP{ADD|SUB}8	✓		Fsimm8_8x8	page 73, page 74
FEPERM32X	✓		Fsimm8_32x2	page 75
FEPERM64X	✓		Fsimm8_64x1	page 75
MOVdTOx		✓	—	page 98
MOVsTO{uw|sw}		✓	—	page 98
MOVfwTO{uw|sw}		✓	—	page 98
MOVwTO{fuw|fsw}		✓	—	page 100

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

reserved	rng_stat	*reserved*	fed	*reserved*	xfd<5:4>	*reserved*	xfd<1:0>
63 41	40	39 37	36	35 6	5 4	3 2	1 0

Bit	Field	Access	Description
63:41	*reserved*	RO	Reserved (undefined).
40	rng_stat	RW	If rng_stat = 1, the value which is read from ASI_RANDOM_NUMBER is valid, otherwise the value is invalid.
39:37	*reserved*	RO	Reserved (undefined)
36	fed	RW	Floating-Point Exception Disable Mode No floating-point exception traps are generated.
35:6	*reserved*	RO	reserved (undefined).
5:4	xfd<5:4>	RW	Updating the floating-point registers (F[382] – F[256]) sets the appropriate bit to 1. Refer to xfd (page 22) for details.
3:2	*reserved*	RO	reserved (undefined)
1:0	xfd<1:0>	RW	Updating the floating-point registers (F[126] – F[0]) sets the appropriate bit to 1. Refer to xfd (page 22) for details.

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

fed

Setting the fed field masks all floating-point exceptions. When XASR.fed = 0, the behavior of the 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.cexc and FSR.ftt are cleared with a 0, regardless of the values of FSR.tem and FSR.ns. In addition, the FSHIFTORX intsruction does not generate an *illegal_instruction* trap.

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

Operation results for fed = 1 are the same as fed = 0, FSR.tem = 0_0000_2 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, nonprivileged 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 (for example, assembly language) should not alter this flag.

When modifying this field, it is the 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 integer registers.
- 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	*Reserved*
3	*Reserved*
4	F[256] – F[318]
5	F[320] – F[382]
6	*Reserved*
7	*Reserved*

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

Refer to the SPARC64 X/X+ specification.

7. Instructions

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

Table 7-1 Meaning of the mnemonic superscripts

Character	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 nonprivileged access is enabled in nonprivileged mode
P_{PIC}	Privileged operation when PCR.priv = 1
P_{PCR}	Privileged accesses when PCR.priv = 1
XII	Instructions supported in SPARC64™ XII only

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

Mark	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
Fq[rs1]	Quadruple-precision floating-point register encoded by the rs1 field of the instruction word	Quadruple-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 (There is no distinction among single precision, double precision, and quadruple precision.)	Floating-point register encoded by XAR.urs1 and the rs1 field of the instruction word (There is no distinction among single precision, double precision, and 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™ XII.

- **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] .

For an instruction which has ※ in this column, XAR.v must be 1 to execute as a non-SIMD instruction.

- **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 XAR-eligible are described in "XAR operation" (page 17).

Fsimm8

If XAR.xar_i = 1, Fsimm8 is used instead of F[rs2] for specific XAR-eligible IMPDEP1 instructions that use F[rs2] (refer to the Table 5-4). When XAR.xar_i = 1 is specified for the instrcutions that are not eligible to use XAR.xar_i, an *illegal_action* exception will occur.

Fsimm8 consists of XAR.urs2 and rs2, and is shown in Figure 7-1 (Fsimm8<7:5> is specified by XAR.urs2<2:0> and Fsimm8<4:0> is specified by rs2<4:0>).

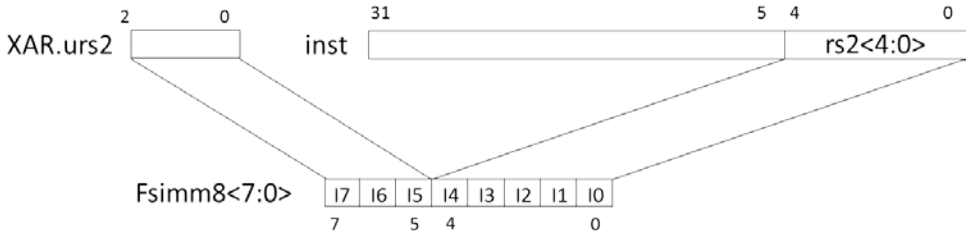


Figure 7-1 Fsimm8 (by XAR.urs2 and rs2 (in the instruction field))

Fsimm8 is used as a signed 64-bit immediate value in the format described in Figure 7-2 (Fsimm8_8x8, Fsimm8_16x4, Fsimm8_32x2, and Fsimm8_64x1).

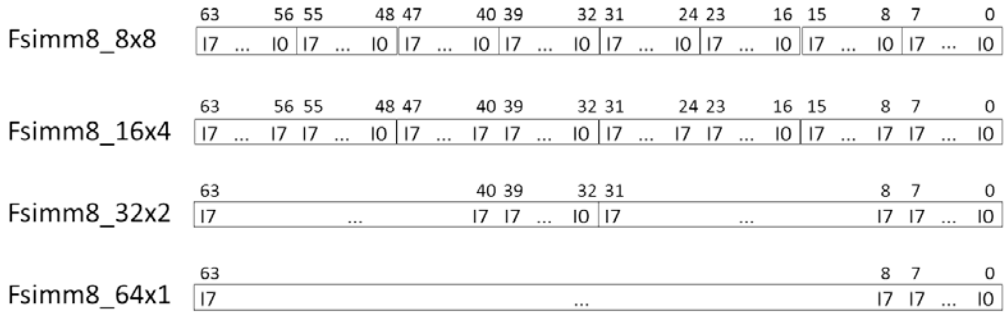


Figure 7-2 Fsimm8 Formats

Fsimm8 is treated as stated below.

- The format of Fsimm8 (Fsimm8_8x8, Fsimm8_16x4, Fsimm8_32x2, and Fsimm8_64x1) depends on the instruction.
- The lower 32-bit of Fsimm8 is ignored when a double floating-point register is used as a single floating-point register.

- Even if 1 is set to the field of Fsimm8 which is not used for executions (for example, the upper field of shift_amount), exceptions (for example, *illegal_action*, *illegal_instruction*, and so on) will not occur.
- If Fsimm8 is used for a SIMD instruction, the same value is used for both basic and extended sides.
- Assembly syntax is described as "[instruction] $freg_{rs1}$, *freg_or_fsimm*, $freg_{rd}$". For "*freg_or_fsimm*", F[rs2] or Fsimm8 can be specified ("*freg_or_fsimm8*" is a newly defined syntax).
- For Fsimm8, 0x00 ~ 0xff (-128 ~ 127) can be specified.

Table 7-3 Instruction set of SPARC64™ XII

Instruction	HPC-ACE extension		Page
	Regs.	SIMD	
ADD (ADDcc)			
ADDC (ADDCcc)			
ALIGNADDRESS{_LITTLE}			
AND (ANDcc)			
ANDN (ANDNcc)			
ARRAY{8|16|32}			
BMASK			
BPcc			
BPr			
BSHUFFLE			
Bicc[D]			
CALL			
CASA[PASI], CASXA[PASI]	✓		
CWB{NE|E|G|LE|GE|L|GU|LEU|CC|CS|POS|NEG|VC|VS}			
CXB{NE|E|G|LE|GE|L|GU|LEU|CC|CS|POS|NEG|VC|VS}			
EDGE{8|16|32}{L}N			
EDGE{8|16|32}{L}cc			
FABSq	✓		
FABS{s|d}	✓	✓	
FADDod	☆		
FADDq	✓		
FADD{s|d}	✓	✓	
FADDtd	☆		
FAESDECLX	✓	✓	
FAESDECX	✓	✓	
FAESENCLX	✓	✓	
FAESENCX	✓	✓	
FAESKEYX	✓	✓	
FALIGNDATA			
FANDNOT{1|2}{s}	✓	✓	
FAND{s}	✓	✓	
FBPfcc			
FBfcc[D]			
FCMP{E}{s|d|q}	✓		
FCMP{E}td	✓		
FCMP{LE|LT|GE|GT|EQ|NE}{E}{s|d}	✓	✓	
FCMP{LE|NE|GT|EQ}{16|32}			
FCMP{LE|GT}{8X|16X|32X|X}	☆		
FPCMP{LE|GT}{8X|16X|32X|64X}	✓	✓	63
FPCMP{LE|GT}{4X}[XII]	✓	✓	63
FCMPod	✓		
FPCMP{LE|GT}{4|8|16|32|64}FX[XII]	※	✓	84
FPCMP{LE|GT}{4|8|16|32|64}XACC[XII]	※	✓	89
FLCMP{s|d}	✓		

FDESENCX	✓	✓	
FDESIIPX	✓	✓	
FDESIPX	✓	✓	
FDESKEYX	✓	✓	
FDESPC1X	✓	✓	
FDIVod	☆		
FDIV{s|d|q}	☆		
FDIVtd	☆		
FEPERM32X[XII]	✓	✓	75
FEPERM64X[XII]	✓	✓	75
FEXPAd	✓	✓	
FEXPAND			
FLUSH			
FLUSHW			
FMADD{s|d}	✓	✓	
FMAX{s|d}	✓	✓	
FMIN{s|d}	✓	✓	
FMONTMUL[XII]			102
FMONTSQR[XII]			102
FMOVq	✓		
FMOVcc			
FMOVR			
FMOV{s|d}	✓	✓	
FMSUB{s|d}	✓	✓	
FMUL8x16			
FMUL8x16{AU|AL}			
FMUL8{SU|UL}x16			
FMULD8{SU|UL}x16			
FMULod	☆		
FMULq	✓		
FMUL{s|d}	✓	✓	
FMULtd	☆		
FNAND{s}	✓	✓	
FNEGq	✓		
FNEG{s|d}	✓	✓	
FNMADD{s|d}	✓	✓	
FNMSUB{s|d}	✓	✓	
FNADD{s|d}	✓	✓	
FNMUL{s|d}	✓	✓	
FNsMULd	✓	✓	
FNOR{s}	✓	✓	
FNOT{1|2}{s}	✓	✓	
FONE{s}	✓	✓	
FORNOT{1|2}{s}	✓	✓	
FOR{s}	✓	✓	
FPACK{16|32|FIX}			
FPADD8[XII]	✓	✓	73

FPADD{16|32}{S}	✓	✓	
FPADD64	✓	✓	
FPADD128XHI	✓	✓	
FPCSL8X[XII]	✓	✓	82
FPMADDX{HI}	✓	✓	
FPMAX{u}{32|64}	✓	✓	
FPMIN{u}{32|64}	✓	✓	
FPMERGE	✓	✓	35
FPMUL32[XII]	✓	✓	71
FPMUL64[XII]	✓	✓	71
FPSUB8[XII]	✓	✓	74
FPSUB{16|32}{S}	✓	✓	
FPSUB64	✓	✓	
F{R}QUAod	☆		
FQUAtd	☆		
FRCPA{s|d}	✓	✓	
FRSQRTA{s|d}	✓	✓	
FPSELMOV{8|16|32}X	✓	✓	
FPSELMOV{8|16|32}FX[XII]	※	✓	95
FSELMOV{s|d}	✓	✓	
FSEXTW[XII]	✓	✓	72
FZEXTW[XII]	✓	✓	72
FSHIFTORX	✓	✓	
FSQRT{s|d|q}	☆		
FSRC{1|2}{s}	✓	✓	
FSUBod	☆		
FSUBq	✓		
FSUB{s|d}	✓	✓	
FSUBtd	☆		
FTRIMADDd	✓	✓	
FTRISMULd	✓	✓	
FTRISSELd	✓	✓	
FUCMP{LE|NE|GT|EQ}{8X|16X|32X|X}	☆		
FPCMPU{LE|NE|GT|EQ}{8X|16X|32X|64X}	✓	✓	63
FPCMPU{LE|NE|GT|EQ}{4X}[XII]	✓	✓	63
FPCMPU{LE|NE|GT|EQ}8			
FPCMPU{LE|NE|GT|EQ}{4|8|16|32|64}FX[XII]	※	✓	84
FPCMPU{LE|NE|GT|EQ}{4|8|16|32|64}XACC[XII]	※	✓	89
FPCMP{64|U64}X	✓		
FXADDod{LO|HI}	☆		
FXMULodLO	☆		
FXNOR{s}	✓	✓	
FXOR{s}	✓	✓	
FZERO{s}	✓	✓	
FdMULq	✓		

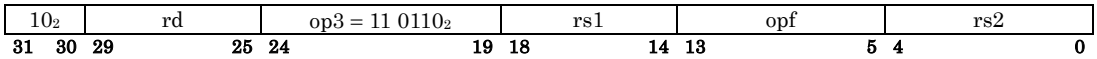
F{bsx|bux|od}TOtd	☆		
FqTO{i|x}	✓		
FsMULd	✓	✓	
F{i|x}TOq	✓		
F{i|x}TO{s|d}	✓	✓	
F{s|d}TOq	✓		
F{s|d}TO{i|x}	✓	✓	
FsTOd, FdTOs	✓	✓	
FtdTO{bsx|bux|od}	☆		
FqTO{s|d}	✓		
ILLTRAP			
JMPL			
LDBLOCKF	✓		
LDF, LDDF	✓	✓	37
LDQF	✓		37
LDFUW^{XII}	※	✓	37
LDFSW^{XII}	※	✓	37
LDDFDS^{XII}	※	✓	37
LDFA$^{P_{ASI}}$, LDDFA$^{P_{ASI}}$	✓	✓	42
LDQFA$^{P_{ASI}}$	✓		42
LDFSR^{D}	✓		
LDSHORTF			
LDSTUB	✓		
LDSTUBA$^{P_{ASI}}$	✓		
LDTW^{D}	✓		
LDTWA$^{D,P_{ASI}}$	✓		
LDTXA^{N}	✓		
LDXEFSR	✓		
LDXFSR	✓		
LD{S|U}{B|H|W}, LDX	✓		
LD{S|U}{B|H|W}A$^{P_{ASI}}$, LDXA$^{P_{ASI}}$	✓		
LZD			
MEMBAR			
MOVcc			
MOVr			
MOVwTOs	✓		
MOVxTOd	✓		
MOVdTOx^{XII}	✓		98
MOVsTOuw^{XII}	✓		98
MOVsTOsw^{XII}	✓		98
MOVfwTOuw^{XII}	※		98
MOVfwTOsw^{XII}	※		98
MOVwTOfuw^{XII}	※		100
MOVwTOfsw^{XII}	※		100
MULScc^{D}			

MULX			
NOP			
OR (Orcc)			
ORN (ORNcc)			
PADD32			
PAUSE			
PDIST			
POPC			
PREFETCH, PREFETCHA$^{P_{ASI}}$	✓		43
RDASI			
RDCCR			
RDFPRS			
RDGSR			
RDPC			
RDPCR$^{P_{PCR}}$			
RDPIC$^{P_{PIC}}$			
RDSTICK$^{P_{NPT}}$			
RDTICK$^{P_{NPT}}$			
RDXASR			
RDY^{D}			
RDASR$^{P_{ASR}}$			
RESTORE			
RETURN			
ROLX			
SAVE			
SDIAM			
SDIV^{D} (SDIVcc^{D})			
SDIVX			
SETHI			
SIAM			
SLEEP			46
SLL, SLLX			
FPSLL64X	✓	✓	
FSLL32^{XII}	✓	✓	69
SMUL^{D} (SMULcc^{D})			
SRA, SRAX			
FPSRA64X	✓	✓	
FSRA32^{XII}	✓	✓	69
SRL, SRLX			
FPSRL64X	✓	✓	
FSRL32^{XII}	✓	✓	69
STBAR^{D}			
STBI^{N}	✓		
STBLOCKF	✓		
STF, STDF	✓	✓	48
STQF	✓		48

STFUW[XII]	※	✓	48
STDFDS[XII]	※	✓	48
STFA[PASI], STDFA[PASI]	✓	✓	51
STQFA[PASI]	✓		51
STFSR[D], STXFSR	✓		
STPARTIALF			
STSHORTF			
ST{B|H|W|X}	✓		
ST{B|H|W|X}A[PASI]	✓		
ST{D}FR	✓	✓	56
STFRUW[XII]	※	✓	56
STDFRDS[XII]	※	✓	56
STDFRDW[XII]	※	✓	56
STTW[D]	✓		
STTWA[D,PASI]	✓		
SUB (SUBcc)			
SUBC (SUBCcc)			
SWAP[D], SWAPA[D,PASI]	✓		
SXAR{1|2}			
TADDcc (TADDccTV[D])			
TSUBcc (TSUBccTV[D])			
Tcc			
UDIV[D] (UDIVcc[D])			
UDIVX			
UMUL[D] (UMULcc[D])			
WRASI			
WRASR[PASR]			
WRCCR			
WRFPRS			
WRGSR			
WRPAUSE			
WRPCR[PPCR]			
WRPIC[PPIC]			
WRXAR			
WRXASR			
WRY[D]			
XFILL[N]	✓		53
XNOR (XNORcc)			
XOR (XORcc)			

7.41. Floating-Point Merge

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FPMERGE	0 0100 1011$_2$	Two 32-bit merges	✓	✓	`fpmerge` *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*



Description

FPMERGE interleaves eight 8-bit data in "Fs[rs1]<31:0> and Fs[rs2]<31:0>" or "Fd[rs1]<63:32> and Fd[rs2]<63:32>" to produce a 64-bit data in Fd[rd].

If XAR.v = 0 and xar_i = 0, Fs[rs1]<31:0> and Fs[rs2]<31:0> are divided into four 8-bit data and merged as shown in Figure 7-3.

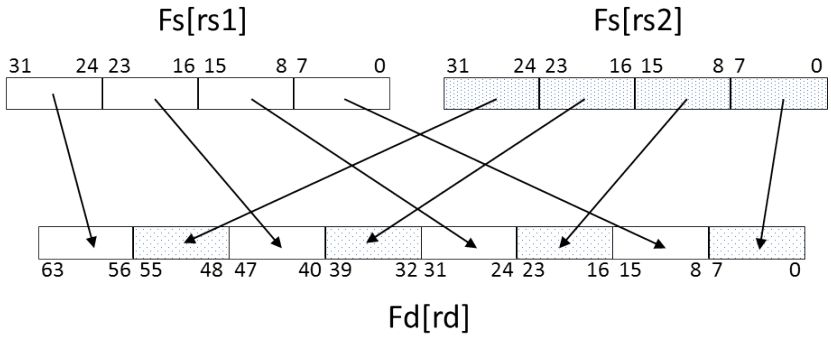


Figure 7-3 The behavior of FPMERGE (XAR.v = 0 and xar_i = 0)

If XAR.v = 1 and xar_i = 0, Fd[rs1]<63:32> and Fd[rs2]<63:32> are divided into four 8-bit data and merged as shown in Figure 7-4.

If XAR.v = 1 and xar_i = 1, Fd[rs1]<63:32> and Fsimm8_8x8<63:32> are divided into four 8-bit data and merged as shown in Figure 7-5.

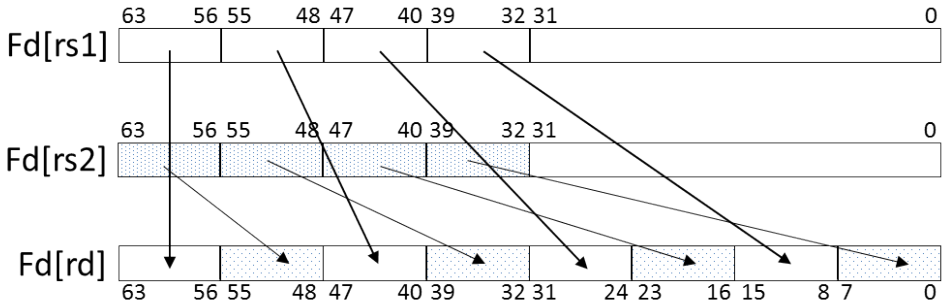


Figure 7-4 Behavior of FPMERGE (XAR.v = 1 and xar_i = 0)

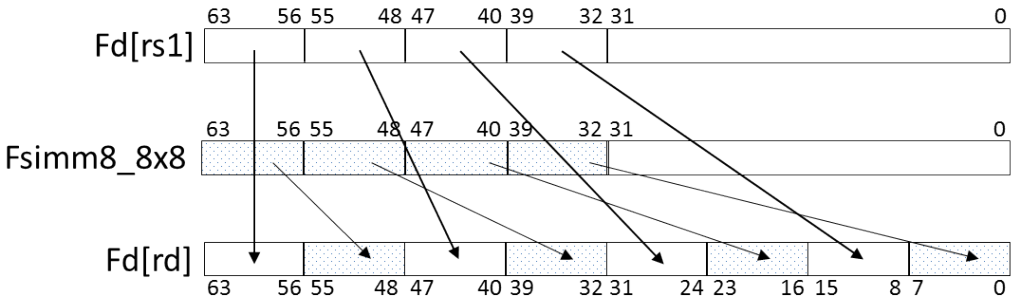


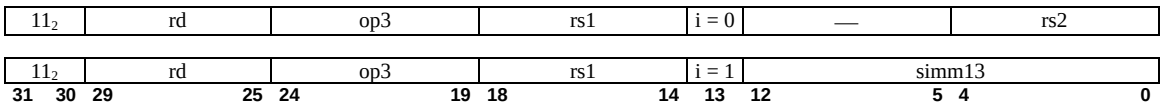
Figure 7-5 Behavior of FPMERGE (XAR.v = 1 and xar_i = 1)

FPMERGE will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.55. Load Floating-Point Register

Instruction	op3	rd[i]	urs2 <2:1>	Operation	HPC-ACE		Assembly Language Syntax
					Regs	SIMD	
LDF	$10\ 0000_2$	0 – 31	—	Load Word Data to Single Floating-Point Register (XAR.v = 0)			`ld` [*address*], *freg*$_{rd}$
LDF	$10\ 0000_2$	0 – 126, 256 – 382	00_2	Load Word Data to Double Floating-Point Register (XAR.v = 1)	✓	✓	`ld` [*address*], *freg*$_{rd}$
LDDF	$10\ 0011_2$	0 – 126, 256 – 382	00_2	Load Double Word Data to Double Floating-Point Register	✓	✓	`ldd` [*address*], *freg*$_{rd}$
LDQF	$10\ 0010_2$	0 – 126, 256 – 382	00_2	Load Quad Word Data to Quad Floating-Point Register	✓		`ldq` [*address*], *freg*$_{rd}$
LDFUW[XII]	$10\ 0000_2$	0 – 126, 256 – 382	01_2	Load Word Data to Double Floating-Point Register as Unsigned Integer	※	✓	`lduw` [*address*], *freg*$_{rd}$
LDFSW[XII]	$10\ 0000_2$	0 – 126, 256 – 382	11_2	Load Word Data to Double Floating-Point Register as Signed Integer	※	✓	`ldsw` [*address*], *freg*$_{rd}$
LDDFDS[XII]	$10\ 0011_2$	0 – 126, 256 – 382	01_2	Load Double Word Data to Double Floating-Point Register as Two Word Data	※	✓	`lddds` [*address*], *freg*$_{rd}$



Description

Non-SIMD operation

Refer to Section 7.75 in UA2011.

LDF copies a word from memory at the effective address into the 4-byte floating-point destination register F[rd]. If XAR.v = 0, LDF copies a word from memory into the 4-byte floating-point destination register, Fs[rd]. If XAR.v = 1, LDF copies a word from memory into the upper 4 bytes of the 8-byte floating-point destination register, Fd[rd]. The lower 4 bytes of Fd[rd] is filled with 0.

LDDF copies a word-aligned doubleword from memory at the effective address into the 8-byte floating-point destination register, Fd[rd].

LDQF copies a word-aligned quadword from memory at the effective address into the 16-byte floating-point destination register, Fq[rd].

LDFUW copies a word from memory at the effective address into the lower 4 bytes of the 8-byte floating-point destination register, Fd[rd]. The upper 4 bytes of Fd[rd] is filled with 0.

LDFSW copies a word from memory at the effective address into the lower 4 bytes of the 8-byte floating-point destination register, Fd[rd]. The upper 4 bytes of Fd[rd] is filled with MSB of the copied data (sign extension).

[i] Encoding is defined in 5.3.1 "Floating-Point Register Number Encoding" in the SPARC64™ X/X+ specification.

LDDFDS copies a word-aligned doubleword from memory at the effective address into the 8-byte floating-point destination register, Fd[rd] as two word data.

An attempt to execute LDF, LDDF, LDQF, LDFUW, LDFSW and LDDFDS causes a *mem_address_not_aligned* exception when the effective address is not word-aligned.

For LDDFDS, the endianness of each memory access for two words is handled separately, even if the two words are located on different pages with different endianness.

LDDFDS can only be used to access cacheable address spaces. An attempt to access noncacheable address space using LDDFDS causes a *DAE_nc_page* exception.

Programming Note LDFUW, LDFSW, and LDDFDS can be used only if XAR.v = 1. If XAR.v = 0, other XAR fields (XAR.urs1, XAR.urs2, XAR.urs3, XAR.urd, and XAR.simd) are treated as 0.

SIMD operation

In SPARC64™ XII, LDF, LDDF, LDFUW, LDFSW, and LDDFDS can be executed as a SIMD instructions. SIMD LDF, SIMD LDDF, SIMD LDFUW, SIMD LDFSW, and SIMD LDDFDS simultaneously execute basic and extended loads from the effective address. Refer to Section 5.5.15 (page 16) for details on how to specify the registers.

A SIMD LDF instruction copies a word from memory at the effective address into the upper 4 bytes of the 8-byte floating-point destination register, Fd[rd]. It then copies a word from memory at the "effective address + 4" into the upper 4 bytes of the 8-byte floating-point destination register Fd[rd + 256].

A SIMD LDDF instruction copies a doubleword-aligned doubleword from memory at the effective address into the 8-byte floating-point register, Fd[rd]. It then copies a doubleword-aligned doubleword from memory at the "effective address + 8" into the 8-byte floating-point register, Fd[rd + 256].

A SIMD LDFUW instruction copies a word from memory at the effective address into the lower 4 bytes of the 8-byte floating-point destination register, Fd[rd]. It then copies a word from memory at the "effective address + 4" into the lower 4 bytes of the 8-byte floating-point destination register, Fd[rd + 256]. The upper 4 bytes of Fd[rd] and Fd[rd + 256] are filled with 0.

A SIMD LDFSW instruction copies a word from memory at the effective address into the lower 4 bytes of the 8-byte floating-point destination register, Fd[rd]. It then copies a word from memory at the "effective address + 4" into the lower 4 bytes of the 8-byte floating-point destination register, Fd[rd + 256]. The upper 4 bytes of Fd[rd] and Fd[rd+256] are filled with MSB of the copied data (sign extension).

A SIMD LDDFDS instruction copies a word-aligned doubleword from memory at the effective address into the 8-byte floating-point destination register, Fd[rd] as two word data. It then copies a word-aligned doubleword from memory at the "effective address + 8" into the 8-byte floating-point destination register, Fd[rd + 256] as two word data.

For SIMD LDF, SIMD LDDF, SIMD LDFUW, SIMD LDFSW, and SIMD LDDFDS, a misaligned accesses causes a *mem_address_not_aligned* exception.

Note A SIMD LDDF 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, SIMD LDDF, SIMD LDFUW, SIMD LDFSW, and SIMD LDDFDS can only be used to access cacheable address spaces. An attempt to access noncacheable address spaces using these SIMD instructions causes a *DAE_nc_page* exception.

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

For SIMD load, a watchpoint can be detected in both the basic and extended loads.

For SIMD `LDF`, SIMD `LDDF`, SIMD `LDFUW`, and SIMD `LDFSW`, the endianness of each memory access for basic data and extended data is handled separately, even if the two data are located on different pages with different endianness.

For SIMD `LDDFDS`, the endianness of each memory access for four word data is handled separately, even if the data are located on different pages with different endianness.

Exception	Target instruction	Detection condition
illegal_instruction	LDF, LDDF, LDFUW, LDFSW, LDDFDS	A *reserved* is not 0.
	LDQF	Always detected. For this instruction, exceptions with a priority lower than *illegal_instruction* are intended for emulation.
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	LDF	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<0> ≠ 0 • XAR.urs2<2:1> = 10_2 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDFUW, LDFSW	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<0> ≠ 0 • XAR.urs2<2:1> = 10_2 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDDF	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<0> ≠ 0 • XAR.urs2<2> ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDDFDS	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<0> ≠ 0 • XAR.urs2<2> ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDQF	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0
fp_exception_other (FSR.ftt = *invalid_fp_register*)	LDQF	rd<1> ≠ 0
LDDF_mem_address_not_aligned	LDDF	XAR.v = 0 or XAR.simd = 0, and the address is 4-byte aligned but not 8-byte aligned.
mem_address_not_aligned	LDF, LDQF, LDFUW, LDFSW, LDDFDS	The address is not 4-byte aligned.

Exception	Target instruction	Detection condition
	LDDF	One of the following is true: • XAR.v = 0 or XAR.simd = 0, and the address is not 4-byte aligned. • XAR.v = 1 and XAR.simd = 1, and the address is not 8-byte aligned.
VA_watchpoint	All	Refer to the description and 12.5.1.62 in the SPARC64™ X/X+ specification.
DAE_privilege_violation	All	Refer to 12.5.1.62 in the SPARC64™ X/X+ specification.
DAE_nc_page	LDDFDS	Access to noncacheable space is attempted.
	LDF, LDDF, LDFUW, LDFSW	Access to noncacheable space is attempted if XAR.v = 1 and XAR.simd = 1.
DAE_nfo_page	All	Refer to 12.5.1.7 in the SPARC64™ X/X+ specification.

7.56. Load Floating-Point from Alternate Space

Description Refer to 7.56 in the SPARC64™ X/X+ specification.

Exception	Target instruction	Detection condition
illegal_instruction	LDQFA	Always detected. For this instruction, exceptions with a priority lower than *illegal_instruction* are intended for emulation.
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	LDFA, LDDFA	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	LDQFA	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0
fp_exception_other (FSR.ftt = *invalid_fp_register*)	LDQFA	rd<1> ≠ 0
LDDF_mem_address_not_aligned	LDDFA	XAR.v = 0 or XAR.simd = 0, and the address is 4-byte aligned but not 8-byte aligned.
mem_address_not_aligned	LDFA, LDQFA	Address is not 4-byte aligned.
	LDDFA	One of the following is true: • XAR.v = 0 or XAR.simd = 0, and the address is not 4-byte aligned. • XAR.v = 1 and XAR.simd = 1, and the address is not 8-byte aligned.
privileged_action	All	Refer to 12.5.1.49 in the SPARC64™ X/X+ specification.
VA_watchpoint	All	Refer to 12.5.1.62 in the SPARC™ X/X+ specification.
DAE_invalid_asi	All	Refer to UA2011 and 12.5.1.5 in the SPARC64™ X/X+ specification.
DAE_privilege_violation	All	Refer to 12.5.1.8 in the SPARC64™ X/X+ specification.
DAE_nc_page	All	Access to noncacheable space is attempted if XAR.v = 1 and XAR.simd = 1.
DAE_nfo_page	All	Refer to 12.5.1.7 in the SPARC64™ X/X+ specification.
DAE_side_effect_page	All	Refer to 12.5.1.9 in the SPARC64™ X/X+ specification.

7.75. Prefetch

Instruction	op3	Operation	HPC-ACE		Assembly Language Syntax	
			Regs	SIMD		
PREFETCH	10 1101$_2$	Prefetch Data	✓		`prefetch`	[*address*], *prefetch_fcn*
PREFETCHA$^{P_{ASI}}$	11 1101$_2$	Prefetch Data from Alternate Space	✓		`prefetch` `prefetch`	[*regaddr*], *imm_asi*, *prefetch_fcn* [*reg_plus_imm*] `%asi`, *prefetch_fcn*

Description

Refer to Section 7.104 in UA2011.

The address specified by the instruction can be arbitorary. As specified by the instruction, one cache line (128 bytes) or two cache lines (256 bytes) are copied. A *mem_address_not_aligned* exception is never generated.

The `PREFETCH{A}` instruction is treated as a `NOP` when the specified address is noncacheable or in an undefined cacheable space.

ASIs that can be specified by the `PREFETCHA` instruction are shown in Table 7-4. If an ASI other than those listed below is specified, the `PREFETCHA` instruction becomes a `NOP`.

Table 7-4 ASIs valid for `PREFETCHA`

`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`

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 confirmed.

7.75.1. Prefetch Variants

Table 7-5 shows the available `fcn`s in SPARC64™ XII and describes their operation.

Table 7-5 fcns for PREFETCH and PREFETCHA

fcn	JPS1 and UA2011 Definition	Operation in SPARC64™ XII
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 LL 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 LL cache with exclusive ownership.
4	Page mapping is performed by privileged software.	NOP
5 - 15 (05_{16} - $0F_{16}$)	An *illegal_instruction* exception is detected.	An *illegal_instruction* 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 LL 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 LL 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 LL cache. Strong prefetch.
30 ($1E_{16}$)		NOP
31 ($1F_{16}$)		256-byte data aligned on 256-byte boundary is copied into the LL cache with exclusive ownership. Strong prefetch.

Note In SPARC64™ XII, LL cache (= Last Level cache) means L3 cache.

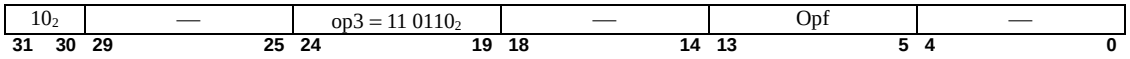
7.75.2. Weak versus Strong Prefetches

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

Exception	Target instruction	Condition
illegal_instruction	All	One of the following is true: • A *reserved* field is not 0. • fcn = 5 – 15
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2, 0> ≠ 0 • XAR.urd<1> ≠ 0

7.89. Sleep

Instruction	Opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
SLEEP	0 1000 0011$_2$	VCPU is stopped during the fixed time.			`sleep`



Description

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

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

- The fixed period of time, which depends on the implementation, has passed.
- An interrupt is pending or has occured.

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

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

Exception	Condition
illegal_instruction	A *Reserved* field is not 0.
illegal_action	XAR.v = 1

7.94. Block Initializing Store

Description

UA2011 defines ASI_STBI_*. In SPARC64™ XII, 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.

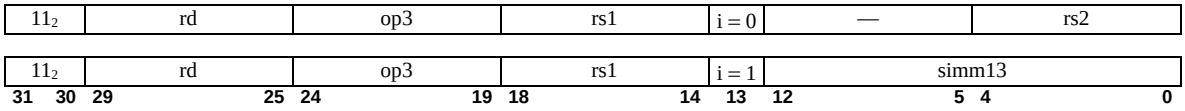
The behavior of the Block Initializing Stores is as follows.

ASI number	**ASI name**	**Integer store** (STBA, STHA, STWA, STXA, and 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
$F2_{16}$	ASI_STBIMRU_PRIMARY	ASI_P
$F3_{16}$	ASI_STBIMRU_SECONDARY	ASI_S
FA_{16}	ASI_STBIMRU_PRIMARY_LITTLE	ASI_PL
FB_{16}	ASI_STBIMRU_SECONDARY_LITTLE	ASI_SL

Only a *DAE_invalid_ASI* exception and a *mem_address_not_aligned* exception are generated. *DAE _** exceptions, except for *DAE_invalid_ASI*, do not occur.

7.96. Store Floating-Point

Instruction	op3	rd[ii]	urs2<1>	Operation	HPC-ACE Regs	HPC-ACE SIMD	Assembly Language Syntax	
STF	$10\ 0100_2$	0 – 31	0	Stores single floating-point register (XAR.v = 0)			st	*freg*$_{rd}$, [*address*]
STF	$10\ 0100_2$	0 – 126, 256 – 382	0	Stores the upper 4 bytes of double floating-point register (XAR.v = 1)	✓	✓	st	*freg*$_{rd}$, [*address*]
STDF	$10\ 0111_2$	0 – 126, 256 – 382	0	Stores double floating-point register	✓	✓	std	*freg*$_{rd}$, [*address*]
STQF	$10\ 0110_2$	0 – 126, 256 – 382	0	Stores quad floating-point register	✓		stq	*freg*$_{rd}$, [*address*]
STFUW[XII]	$10\ 0100_2$	0 – 126, 256 – 382	1	Stores the lower 4 bytes of double floating-point register	※	✓	stuw	*freg*$_{rd}$, [*address*]
STDFDS[XII]	$10\ 0111_2$	0 – 126, 256 – 382	1	Stores double floating-point register as Two Word Data	※	✓	stdds	*freg*$_{rd}$, [*address*]



Description

Non-SIMD operation

Refer to Section 7.122 in UA2011.

STF copies 4 bytes of the floating-point register F[rd] into a word-aligned word to memory at the effective address. If XAR.v = 0, STF copies the contents of the 4-byte floating-point register Fs[rd] to memory. If XAR.v = 1, STF copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] to memory.

STDF copies the contents of the 8-byte floating-point register Fd[rd] into a word-aligned doubleword to memory at the effective address.

STQF copies the contents of the 16-byte floating-point register Fq[rd] into a word-aligned quadword to memory at the effective address.

STFUW copies the lower 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address.

STDFDS copies the contents of the 8-byte floating-point register Fd[rd] into a word-aligned doubleword as two word data to memory at the effective address.

For STF, STDF, STQF, STDFUW, and STDFDS, a misaligned accesses causes a *mem_address_not_aligned* exception.

The STQF instruction is defined by SPARC V9 but is not implemented in SPARC64™ XII. If STQF is executed, an *illegal_instruction* exception occurs.

[ii] Encoding is defined in 5.3.1 "Floating-Point Register Number Encoding" in the SPARC64™ X/X+ specification.

STDFDS can only write to cacheable address spaces. An attempt to access noncacheable space causes a *DAE_nc_page* exception.

> **Programming Note** STFUW and STDFDS can be used only when XAR.v = 1. When XAR.v = 0, other XAR fields (XAR.urs1, XAR.urs2, XAR.urs3, XAR.urd, and XAR.simd) are treated as all 0.

For STDFDS, the endianness of each memory access for two words is handled separately, even if the two words are located on different pages with different endianness.

SIMD operation

In SPARC64™ XII, STF, STDF, STFUW, and STDFDS can be executed as a SIMD instruction. SIMD STF, SIMD STDF, SIMD STFUW, and SIMD STDFDS simultaneously execute basic and extended stores. Refer to Section 5.5.15 for details on how to specify the registers.

SIMD STF copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address and copies the upper 4 bytes of the 8-byte floating-point register Fd[rd + 256] into a word-aligned word to memory at the "effective address + 4".

SIMD STDF copies the contents of the 8-byte floating-point register Fd[rd] into a doubleword-aligned doubleword to memory at the effective address and copies the contents of the 8-byte floating-point register Fd[rd + 256] into a doubleword-aligned doubleword to memory at the "effective address + 8".

SIMD STFUW copies the lower 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address and copies the lower 4 bytes of the 8-byte floating-point register Fd[rd + 256] into a word-aligned word to memory at the "effective address + 4".

SIMD STDFDS copies the contents of the 8-byte floating-point register Fd[rd] into a word-aligned doubleword to memory at the effective address as two word data and copies the contents of the 8-byte floating-point register Fd[rd + 256] into a word-aligned doubleword to memory at "the effective address + 8" as two word data.

For SIMD STF, SIMD STDF, SIMD STDFUW, and SIMD STDFDS, a misaligned access causes 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, SIMD STDF, SIMD STFUW, and SIMD STDFDS can only write to cacheable address spaces. An attempt to access noncacheable space causes a *DAE_nc_page* exception.

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

A *VA_watchpoint* exception can be detected in either the basic or extended operation of SIMD STF, SIMD STDF, SIMD STFUW, and SIMD STDFDS.

For SIMD STF, SIMD STDF, and SIMD STFUW, the endianness of each memory access for basic data and extend data is handled separately, even if the two data are located on different pages with different endianness.

For SIMD STDFDS, the endianness of each memory access for four word data is handled separately, even if the data are located on different pages with different endianness.

,

Exception	Target instruction	Detection condition
illegal_instruction	STF, STDF, STFUW, STDFDS	i = 0 and *reserved* is not 0.
	STQF	Always detected. For this instruction, exceptions with a priority lower than *illegal_instruction* are intended for emulation.
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	STF, STDF, STFUW, STDFDS	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<2,0> ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	STQF	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0
fp_exception_other (FSR.ftt = *invalid_fp_register*)	STQF	rd<1> ≠ 0
STDF_mem_address_not_aligned	STDF	Address is aligned on a 4-byte boundary but not an 8-byte boundary when XAR.v = 0 or XAR.simd = 0.
mem_address_not_aligned	STF, STQF, STFUW, STDFDS	Address is not aligned on a 4-byte boundary
	STDF	One of the following is true: • Address is not aligned on a 4-byte boundary when XAR.v = 0 or XAR.simd = 0. • Address is not aligned on an 8-byte boundary when XAR.v = 1 and XAR.simd = 1.
VA_watchpoint	All	Refer to the description and to 12.5.1.62 in the SPARC64™ X/X+ specification.
DAE_privilege_violation	All	Refer to 12.5.1.8 in the SPARC64™ X/X+ specification.
DAE_nc_page	STF, STDF, STFUW	An access to noncacheable space is attempted when XAR.v = 1 and XAR.simd = 1.
	STDFDS	An access to noncacheable space is attempted.
DAE_nfo_page	All	Refer to 12.5.1.7 in the SPARC64™ X/X+ specification.

7.97. Store Floating-Point into Alternate Space

Compatibility Note Only the differences between the specification of SPARC64™ XII and SPARC64™ X/SPARC64™ X+ is described.

Desciption Refer to Section 7.97 in the SPARC64™ X/X+ specification.

SIMD operation

SIMD STFA copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address and copies the upper 4 bytes of the 8-byte floating-point register Fd[rd + 256] into a word-aligned word to memory at the "effective address + 4". A misaligned access causes a *mem_address_not_aligned* exception.

Programming Note In SPARC64™ X/X+, the address must be doubleword-aligned.

SIMD STDFA copies the contents of the 8-byte floating-point register Fd[rd] into a doubleword-aligned doubleword to memory at the effective address and copies the contents of the 8-byte floating-point register Fd[rd + 256] into a doubleword-aligned doubleword to memory at the "effective address + 8". A misaligned access causes a *mem_address_not_aligned* exception.

Programming Note In SPARC64™ X/X+, the address must be quadword-aligned.

Exception	Target instruction	Detection condition
illegal_instruction	STQFA	Always detected. For this instruction, exceptions with a priority lower than *illegal_instruction* are intended for emulation.
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	STFA, STDFA	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	STQFA	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0
fp_exception_other (FSR.ftt = *invalid_fp_register*)	STQFA	rd<1> ≠ 0
STDF_mem_address_not_aligned	STDFA	Address is aligned on a 4-byte boundary but not an 8-byte boundary when XAR.v = 0 or XAR.simd = 0.
mem_address_not_aligned	STFA, STQFA	Address is not aligned on a 4-byte boundary.
	STDFA	One of the following is true: • Address is not aligned on a 4-byte boundary when XAR.v = 0 or XAR.simd = 0. • Address is not aligned on an 8-byte boundary when XAR.v = 1 and XAR.simd = 1.
privileged_action	All	Refer to 12.5.1.49 in the SPARC64™ X/X+ specification.
VA_watchpoint	All	Refer to 7.97 in the the SPARC64™ X/X+ specification.
DAE_invalid_asi	All	Refer to 7.97 and 12.5.1.5 in the SPARC64™ X/X+ specification.
DAE_privilege_violation	All	Refer to 12.5.1.8 in the SPARC64™ X/X+ specification.
DAE_nc_page	All	An access to noncacheable space is attempted when XAR.v = 1 and XAR.simd = 1.
DAE_nfo_page	All	Refer to 12.5.1.7 in the SPARC64™ X/X+ specification.

7.114. Cache Line Fill with Undetermined Values

Instruction	ASI	op3	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
XFILL[N]	ASI_XFILL_P (ASI = 0x$F4_{16}$) ASI_XFILL_S (ASI = 0x$F5_{16}$)	01 1110_2 01 0100_2 01 0110_2 01 0111_2 01 0101_2 11 0100_2 11 0111_2	Accesses the cache at the specified address and fills the cache line with undetermined values.	✓		stxa *reg$_{rd}$*, [*reg_plus_imm*] %*asi* stxa *reg$_{rd}$*, [*regaddr*] *imm_asi* stwa *reg$_{rd}$*, [*reg_plus_imm*] %*asi* stwa *reg$_{rd}$*, [*regaddr*] *imm_asi* stha *reg$_{rd}$*, [*reg_plus_imm*] %*asi* stha *reg$_{rd}$*, [*regaddr*] *imm_asi* sttwa *reg$_{rd}$*, [*reg_plus_imm*] %*asi* sttwa *reg$_{rd}$*, [*regaddr*] *imm_asi* stba *reg$_{rd}$*, [*reg_plus_imm*] %*asi* stba *reg$_{rd}$*, [*regaddr*] *imm_asi* sta *freg$_{rd}$*, [*reg_plus_imm*] %*asi* sta *freg$_{rd}$*, [*regaddr*] *imm_asi* stda *freg$_{rd}$*, [*reg_plus_imm*] %*asi* stda *freg$_{rd}$*, [*regaddr*] *imm_asi*

11_2	rd	op3	rs1	i = 0	imm_asi	rs2
31 30	29 25	24 19	18 14	13	12 5	4 0

11_2	rd	op3	rs1	i = 1	simm13
31 30	29 25	24 19	18 14	13	12 0

Description

If ASI_XFILL_P or ASI_XFILL_S is specified for STXA, STWA, STHA, STTWA, STBA, STFA, or STDFA instruction, the cache line for the specified address is ensured for writing and is filled with an undefined value. Data is not transferred to the CPU from memory. Any address in the cache line can be specified.

Programming Note In SPARC64™ X/X+, XFILL is implemented as NOP.

Programming Note In SPARC64™ X/X+, XFILL is implemented for only 8-byte store instructions (STXA, STTWA, and STDFA). In SPARC64™ XII, XFILL is implemented for 1-byte store instruction (STBA), 2-byte store instruction (STHA), 4-byte store instructions (STFA and STWA), and 8-byte store instructions (STXA, STTWA, and STDFA).

STXA and STTWA cause *mem_address_not_aligned* exceptions if the effective memory address is not doubleword-aligned.

STFA, STWA, and STDFA cause *mem_address_not_aligned* exceptions if the effective memory address is not word-aligned.

STHA causes *mem_address_not_aligned* exceptions if the effective memory address is not halfword-aligned.

STDFA causes *STDF_mem_address_not_aligned* exceptions if the effective memory address is word-aligned but not doubleword-aligned.

The ordering between XFILL and the following memory access conforms to the TSO.

An attempt to access a page for noncacheable address space using XFILL can cause an exception, but a cache line fill is not performed. In addition, XFILL for noncacheable address space does not cause a *DAE_nc_page* exception.

A watchpoint is detected for all 128 bytes in the cache line.

If a subsequent access to the same cache line occurs while the cache line is being filled, the access is delayed until the cache line fill commits.

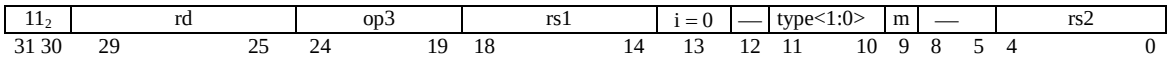
Programming Note MEMBAR is not required between XFILL and the following access. The performance can be negatively affected because the following access is delayed.

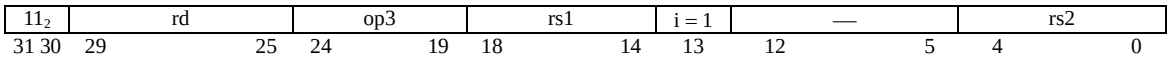
Programming Note When performance is required, it is important for the compiler or the assembler to issue XFILL well in advance of the actual store. The time required to commit XFILL depends on the system. Therefore there may be cases where XFILL is executed reasonably early in one system, but not in another (such as future versions of the processor).

Exception	Target instruction	Detection condition
illegal_instruction	STTWA	Odd-numbered destination register (rd)
fp_disabled	STFA, STDFA	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	STXA, STTWA, STWA, STBA, STHA	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd ≠ 0
	STFA, STDFA	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0
STDF_mem_address_not_aligneed	STDFA	Address is aligned on a 4-byte boundary but not an 8-byte boundary.
mem_address_not_aligned	STXA, STTWA	Address is not aligned on an 8-byte boundary.
	STFA, STWA, STDFA	Address is not aligned on a 4-byte boundary.
	STHA	Address is not aligned on a 2-byte boundary.
VA_watchpoint	All	When the watchpoint address matches any address in the cache line. Refer to 12.5.1.62 in the SPARC64™ X/X+ specification.
DAE_privilege_violation	All	Refer to 12.5.1.8 in the SPARC64™ X/X+ specification.
DAE_nfo_page	All	Refer to 12.5.1.7 in the SPARC64™ X/X+ specification.

7.137. Store Floating-Point Register on Register Condition

Instruction	op3	rs2, rd	i	type <1:0>	m	Operation	HPC-ACE		Assembly Language Syntax
							Regs	SIMD	
STFR	$10\ 1100_2$	0 – 31	0_2, 1_2	00_2	0_2	Stores single-precision floating-point register on register condition (XAR.v = 0)			`stfr` *freg$_{rd}$, freg$_{rs2}$,* [*reg$_{rs1}$*]
STFR	$10\ 1100_2$	0 – 126, 256 – 382	0_2, 1_2	00_2	0_2	Stores the upper 4 bytes of double-precision floating-point register on register condition (XAR.v = 1)	✓	✓	`stfr` *freg$_{rd}$, freg$_{rs2}$,* [*reg$_{rs1}$*]
STDFR	$10\ 1111_2$	0 – 126, 256 – 382	0_2, 1_2	00_2	0_2	Stores double-precision floating-point register on register condition	✓	✓	`stdfr` *freg$_{rd}$, freg$_{rs2}$,* [*reg$_{rs1}$*]
STFRUW[XII]	$10\ 1100_2$	0 – 126, 256 – 382	0_2	01_2	0_2	Stores the lower 4 bytes of double-precision floating-point register on register condition	※	✓	`stfruw` *freg$_{rd}$, freg$_{rs2}$,* [*reg$_{rs1}$*]
STDFRDS[XII]	$10\ 1111_2$	0 – 126, 256 – 382	0_2	01_2	0_2	Stores double-precision floating-point register as two words on register condition	※	✓	`stdfrds` *freg$_{rd}$, freg$_{rs2}$,* [*reg$_{rs1}$*]
STDFRDW[XII]	$10\ 1111_2$	0 – 126, 256 – 382	0_2	01_2	1_2	Stores double-precision floating-point register as two words on register condition	※	✓	`stdfrdw` *freg$_{rd}$, freg$_{rs2}$,* [*reg$_{rs1}$*]





Description

non-SIMD operation

STFR copies the contents of the 4-byte floating-point register Fs[rd] into a word-aligned word to memory at the effective address if XAR.v = 0 and Fs[rs2]<31> = 1, and copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address if XAR.v = 1 and Fd[rs2]<63> = 1.

> **Compatibility Note** The behavior of STFR when i = 0 is the same as that when i = 1.

STDFR copies the contents of the 8-byte floating-point register Fd[rd] into a word-aligned doubleword to memory at the effective address if Fd[rs2]<63> = 1.

> **Compatibility Note** The behavior of STDFR when i = 0 is the same as that when i = 1.

STFRUW copies the lower 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address if Fd[rs2]<63> = 1.

STDFRDS copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address if Fd[rs2]<63> = 1, and copies the lower 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the "effective address + 4" if Fd[rs2]<31> = 1.

STDFRDW copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address if Fd[rs2]<63> = 1, and copies the lower 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the "effective address + 4" if Fd[rs2]<62> = 1.

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

For STFR, STDFR, STFRUW, STDFRDS, and STDFRDW, a misaligned access causes a *mem_address_not_aligned* exception.

When a non-SIMD STDFR is executed, 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, an *STDF_mem_address_not_aligned* exception will occur. The trap handler must emulate the STDFR instruction when this exception occurs.

STDFRDS and STDFRDW are able to write only to cacheable space. A *DAE_nc_page* exception will occur when writing to noncacheable space.

STFR does not cause any exceptions other than *illegal_instruction*, *fp_disabled*, and *illegal_action* if "XAR.v = 1 and Fd[rs2]<63> = 0" or if "XAR.v = 0 and Fs[rs2]<31> = 0".

STDFR does not cause any exceptions other than *illegal_instruction*, *fp_disabled*, and *illegal_action* if Fd[rs2]<63> = 0.

STFRUW does not cause any exceptions other than *illegal_instruction*, *fp_disabled*, and *illegal_action* if Fd[rs2]<63> = 0.

STDFRDS does not cause any exceptions other than *illegal_instruction*, *fp_disabled*, and *illegal_action* for Fd[rd]<63:32> if Fd[rs2]<63> = 0, and for Fd[rd]<31:0> if Fd[rs2]<31> = 0.

STDFRDW does not cause any exceptions other than *illegal_instruction*, *fp_disabled*, and *illegal_action* for Fd[rd]<63:32> if Fd[rs2]<63> = 0, and for Fd[rd]<31:0> if Fd[rs2]<62> = 0.

Exceptions that are always detected	Exceptions that are detected when the corresponding condition for each instructiuon is satisfied (Fs[rs2]<31> = 1, Fd[rs2]<63> = 1, Fd[rs2]<62> = 1, or Fd[rs2]<31> = 1).
Illegal_instruction *fp_disabled* *illegal_action*	*mem_address_not_aligned* *STDF_mem_address_not_aligned* *VA_watchpoint* *DAE_privilege_violation* *DAE_nc_page* *DAE_nfo_page*

For STDFRDS and STDFRDW, the endianness of each memory access for two words is handled separately, even if the two words are located on different pages with different endianness.

SIMD operation

STFR, STDFR, STFRUW, STDFRDS, and STDFRDW support SIMD execution in SPARC64™ XII. SIMD STFR, SIMD STDFR, SIMD STFRUW, SIMD STDFRDS, and SIMD STDFRDW simultaneously execute basic and extended stores. Refer to Section 5.5.15 for details on how to specify the registers.

SIMD STFR copies the upper 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address when Fd[rs2]<63> = 1. It then copies the upper 4 bytes of the 8-byte floating-point register Fd[rd + 256] into a word-aligned word to memory at the "effective address + 4" when Fd[rs2+256]<63> = 1. A misaligned access causes a *mem_address_not_aligned* exception.

Compatibility Note The behavior of STFR when i = 0 is the same as that when i = 1.

SIMD STDFR copies the contents of the 8-byte floating-point register Fd[rd] into a doubleword-aligned doubleword to memory at the effective address when Fd[rs2]<63> = 1. It then copies the contents of the 8-byte floating-point register Fd[rd + 256] into a doubleword-aligned doubleword to memory at the "effective address + 8" when Fd[rs2 + 256]<63> = 1. A misaligned access causes a *mem_address_not_aligned* exception.

Compatibility Note The behavior of STDFR when i = 0 is the same as that when i = 1.

SIMD STFRUW copies the lower 4 bytes of the 8-byte floating-point register Fd[rd] into a word-aligned word to memory at the effective address when Fd[rs2]<63> = 1. It then copies the lower 4 bytes of the 8-byte floating-point register Fd[rd + 256] into a word-aligned word to memory at the "effective address + 4" when Fd[rs2 + 256]<63> = 1. A misaligned access causes a *mem_address_not_aligned* exception.

SIMD STDFRDS copies Fd[rd]<63:32>, Fd[rd]<31:0>, Fd[rd+256]<63:32>, and Fd[rd+256]<31:0> into a word-aligend word to memory at the effective address, "effective address + 4", "effective address + 8" , and "effective address + 12" respectively under the conditions stated below. If the conditions are not satisfied, the corresponding data is not copied. A misaligned access causes a *mem_address_not_aligned* exception.

Data	Condition
Fd[rd]<63:32>	Fd[rs2]<63> = 1
Fd[rd]<31:0>	Fd[rs2]<31> = 1
Fd[rd+256]<63:32>	Fd[rs2+256]<63> = 1

Fd[rd+256]<31:0>	Fd[rs2+256]<31> = 1

A SIMD STDFRDW copies Fd[rd]<63:32>, Fd[rd]<31:0>, Fd[rd+256]<63:32>, and Fd[rd+256]<31:0> into a word-aligned word to memory at the effective address, "effective address + 4", "effective address + 8" , and "effective address + 12" respectively under the condition stated below. If the condition is not satisfied, the corresponding data is not copied. A misaligned access causes a *mem_address_not_aligned* exception.

data	the condition
Fd[rd]<63:32>	Fd[rs2]<63> = 1
Fd[rd]<31:0>	Fd[rs2]<62> = 1
Fd[rd+256]<63:32>	Fd[rs2+256]<63> = 1
Fd[rd+256]<31:0>	Fd[rs2+256]<62> = 1

These floating-point store instructions use implicit ASI (Refer to 6.3.1.3 in UA2011) to access the memory.

For SIMD STFR, SIMD STDFR, SIMD STFRUW, SIMD STDFRDS, and SIMD STDFRDW, a misaligned access causes a *mem_address_not_aligned* exception.

> **Note** SIMD STDFR does not cause an *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, SIMD STDFR, SIMD STFRUW, SIMD STDFRDS, and SIMD STDFRDW can only be used to access cacheable address spaces. An attempt to access noncacheable address space causes a *DAE_nc_page* exception.

SIMD STFR, SIMD STDFR, SIMD STFRUW, SIMD STDFRDS, and SIMD STDFRDW always detect *illegal_instruction*, *fp_disabled*, and *illegal_action* exceptions if the detection condition is met. Other exceptions can be detected if the detection condition and the condition of Fd[rs2] or Fd[rs2+256] are met.

SIMD STFR, SIMD STDFR, SIMD STFRUW, SIMD STDFRDS, and SIMD STDFRDW cause an exception which is found in corresponding basic or extended elements when detection conditions for exceptions other than *illegal_instruction*, *fp_disabled*, and *illegal_action* are met and either the condition of Fd[rs2] corresponding to basic elements or the condition of Fd[rs2 + 256] corresponding to extended elemtns is satisfied. In addition, they cause exceptions in both basic and extended elements when both the condition of Fd[rs2] and Fd[rs2 + 256] are satisfied.

Exceptions that are always detected	Exceptions that are detected when the corresponding conditions for each instruction is satisfied (Fd[rs2]<63> = 1, Fd[rs2]<62> = 1, Fd[rs2]<31> = 1, Fd[rs2 + 256]<63> = 1, Fd[rs2 + 256]<62> = 1, or Fd[rs2 +256]<31> = 1).
Illegal_instruction *fp_disabled* *illegal_action*	*mem_address_not_aligned* *VA_watchpoint* *DAE_privilege_violation* *DAE_nc_page* *DAE_nfo_page*

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

For SIMD STFR, SIMD STDFR, and SIMD STFUW, the endianness of each memory access for basic data and extended data is handled separately, even if the two data are located on different pages with different endianness.

For SIMD STDFRDS and SIMD STDFRDW, the endianness of each memory access for four word data is handled separately, even if they are located on different pages with different endianness.

Exception	Target instruction	Detection condition
illegal_instruction	STFR, STDFR, STFRUW	• i = 1 and iw<12:5> ≠ 0. • If i = 0 and one of the following is true: ・iw<12, 8:5> ≠ 0 ・type<1> = 1 ・m = 1
	STDFRDS, STDFRDW	• i = 1 and iw<12:5> ≠ 0. • If i = 0 and one of the following is true: ・iw<12, 8:5> ≠ 0 ・type<1> = 1
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	STFR, STDFR	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<2,0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0
	STFRUW, STDFRDS, STDFRDW	• XAR.v = 0 • If XAR.v = 1 and one of the following is true: ・XAR.urs1 ≠ 0 ・XAR.urs2<1> ≠ 0 ・XAR.urs3<2,0> ≠ 0 ・XAR.urd<1> ≠ 0 ・XAR.simd = 1 and XAR.urs2<2> ≠ 0 ・XAR.simd = 1 and XAR.urd<2> ≠ 0
STDF_mem_address_not_aligned	STDFR	MSB of Fd[rs2] is 1 and the address is aligned on a word bounrary but not on a doubleword boundary when XAR.v = 1 and XAR.simd = 0, or XAR.v = 0.
mem_address_not_aligned	STFR	One of the following is true: • Address is not aligned on a word boundary when XAR.v = 0 and the MSB of (bit 31) of Fs[rs2] is 1. • Address is not aligned on a word boundary when XAR.v = 1, XAR.simd = 0, and the MSB (bit 63) of Fd[rs2] is 1. • Address is not aligned on a word boundary when the MSB (bit 63) of Fd[rs2] or Fd[rs2+256] is 1, XAR.v = 1, and XAR.simd = 1.
	STDFR	One of the following is true: • Address is not aligned on a doubleword boundary when the MSB (bit 63) of Fd[rs2] is 1 and XAR.v = 0. • Address is not aligned on a doubleword boundary when the MSB (bit 63) of Fd[rs2] is 1, XAR.v = 1, and XAR.simd = 0. • Address is not aligned on a doubleword boundary when the MSB (bit 63) of Fd[rs2] or Fd[rs2+256] is 1, XAR.v = 1, and XAR.simd = 1.

	STFRUW	One of the following is true: • Address is not aligned on a word boundary when the MSB (bit 63) of Fd[rs2] is 1, XAR.v = 1, and XAR.simd = 0. • Address is not aligned on a word boundary when the MSB (bit 63) of Fd[rs2] or Fd[rs2 + 256] is 1, XAR.v = 1, and XAR.simd = 1.
	STDFRDS	One of the following is true: • Address is not aligned on a word boundary when Fd[rs2]<63, 31> ≠ 0, XAR.v = 1, and XAR.simd = 0. • Address is not aligned on a word boundary when Fd[rs2]<63, 31> ≠ 0, Fd[rs2+256]<63, 31> ≠ 0, XAR.v = 1, and XAR.simd = 1.
	STDFRDW	One of the following is true: • Address is not aligned on a word boundary when Fd[rs2]<63> or Fd[rs2]<62> is 1, XAR.v = 1, and XAR.simd = 0. • Address is not aligned on a word boundary when Fd[rs2]<63>, Fd[rs2]<62>, Fd[rs2+256]<63>, or Fd[rs2+256]<62> is 1, XAR.v = 1, and XAR.simd = 1.
VA_watchpoint	All	Refer to 7.137 and to 12.5.1.62 in the SPARC64™ X/X+ specification.
DAE_privilege_violation	All	Refer to 12.5.1.8 in the SPARC64™ X/X+ specification.
DAE_nc_page	STFR, STDFR, STFRUW	An access to noncacheable space is attempted when XAR.v = 1, XAR.simd = 1, and MSB of Fd[rs2] is 1.
	STDFRDS, STDFRDW	An access to noncacheable space is attemped.
DAE_nfo_page	All	Refer to 12.5.1.7 in the SPARC64™ X/X+ specification.

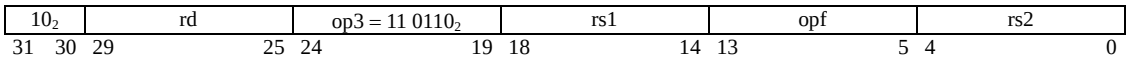
7.139. SIMD Compare (type A)

Instruction	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPCMPLE16X	0 1100 0000_2	00_2	Compares four 16-bit signed integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	`fpcmple16x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fcmple16x)`†
FPCMPULE16X	0 1100 0001_2	00_2	Compares four 16-bit unsigned integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	`fpcmpule16x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmple16x)`†
FPCMPLE4X[XII]	0 1100 0010_2	00_2	Compares sixteen 4-bit signed integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	`fpcmple4x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE16X	0 1100 0011_2	00_2	Compares four 16-bit unsigned integers If *src1* ≠ *src2*, the corresponding result is 1.	✓	✓	`fpcmpune16x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmpne16x)`†
FPCMPLE32X	0 1100 0100_2	00_2	Compares two 32-bit signed integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	`fpcmple32x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fcmple32x)`†
FPCMPULE32X	0 1100 0101_2	00_2	Compares two 32-bit unsigned integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	`fpcmpule32x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmple32x)`†
FPCMPULE4X[XII]	0 1100 0110_2	00_2	Compares sixteen 4-bit unsigned integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	`fpcmpule4x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE32X	0 1100 0111_2	00_2	Compares two 32-bit unsigned integers If *src1* ≠ *src2*, the corresponding result is 1.	✓	✓	`fpcmpune32x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmpne32x)`†
FPCMPGT16X	0 1100 1000_2	00_2	Compares four 16-bit signed integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	`fpcmpgt16x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fcmpgt16x)`†
FPCMPUGT16X	0 1100 1001_2	00_2	Compares four 16-bit unsigned integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	`fpcmpugt16x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmpgt16x)`†
FPCMPUEQ16X	0 1100 1011_2	00_2	Compares four 16-bit unsigned integers If *src1* = *src2*, the corresponding result is 1.	✓	✓	`fpcmpueq16x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmpeq16x)`†
FPCMPGT32X	0 1100 1100_2	00_2	Compares two 32-bit signed integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	`fpcmpgt32x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fcmpgt32x)`†
FPCMPUGT32X	0 1100 1101_2	00_2	Compares two 32-bit unsigned integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	`fpcmpugt32x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmpgt32x)`†

Instruction	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPCMPUNE4X[XII]	0 1100 1110_2	00_2	Compares sixteen 4-bit unsigned integers If *src1* ≠ *src2*, the corresponding result is 1.	✓	✓	fpcmpune4x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUEQ32X	0 1100 1111_2	00_2	Compares two 32-bit unsigned integers If *src1* = *src2*, the corresponding result is 1.	✓	✓	fpcmpueq32x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmpeq32x)[†]
FPCMPLE8X	0 1101 0000_2	00_2	Compares eight 8-bit signed integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	fpcmple8x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fcmple8x)[†]
FPCMPULE8X	0 1101 0001_2	00_2	Compares eight 8-bit unsigned integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	fpcmpule8x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmple8x)[†]
FPCMPGT4X[XII]	0 1101 0010_2	00_2	Compares sixteen 4-bit signed integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	fpcmpgt4x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE8X	0 1101 0011_2	00_2	Compares eight 8-bit unsigned integers If *src1* ≠ *src2*, the corresponding result is 1.	✓	✓	fpcmpune8x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmpne8x)[†]
FPCMPLE64X	0 1101 0100_2	00_2	Compares 64-bit signed integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	fpcmple64x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fcmplex)[†]
FPCMPULE64X	0 1101 0101_2	00_2	Compares 64-bit unsigned integers If *src1* ≤ *src2*, the corresponding result is 1.	✓	✓	fpcmpule64x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmplex)[†]
FPCMPUGT4X[XII]	0 1101 0110_2	00_2	Compares sixteen 4-bit unsigned integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	fpcmpugt4x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE64X	0 1101 0111_2	00_2	Compares 64-bit unsigned integers If *src1* ≠ *src2*, the corresponding result is 1.	✓	✓	fpcmpune64x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmpnex)[†]
FPCMPGT8X	0 1101 1000_2	00_2	Compares eight 8-bit signed integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	fpcmpgt8x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fcmpgt8x)[†]
FPCMPUGT8X	0 1101 1001_2	00_2	Compares eight 8-bit unsigned integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	fpcmpugt8x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmpgt8x)[†]
FPCMPUEQ8X	0 1101 1011_2	00_2	Compares eight 8-bit unsigned integers If *src1* = *src2*, the corresponding result is 1.	✓	✓	fpcmpueq8x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fucmpeq8x)[†]
FPCMPGT64X	0 1101 1100_2	00_2	Compares 64-bit signed integers If *src1* > *src2*, the corresponding result is 1.	✓	✓	fpcmpgt64x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ (fcmpgtx)[†]
FPCMPUGT64X	0 1101 1101_2	00_2	Compares 64-bit unsigned integers	✓	✓	fpcmpugt64x *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$

Instruction	opf	urs3 <1:0>	Operation	HPC-ACE Regs	HPC-ACE SIMD	Assembly Language Syntax
			If *src1* > *src2*, the corresponding result is 1.			`(fucmpgtx)` †
`FPCMPUEQ4X`[XII]	0 1101 1110_2	00_2	Compares sixteen 4-bit unsigned integers If *src1* = *src2*, the corresponding result is 1.	✓	✓	`fpcmpueq4x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
`FPCMPUEQ64X`	0 1101 1111_2	00_2	Compares 64-bit unsigned integers If *src1* = *src2*, the corresponding result is 1.	✓	✓	`fpcmpueq64x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$ `(fucmpeqx)` †

† former mnemonic for this instruction (still recognized by the assembler)



Description

These instructions compare several elements (partitions) in the two floating-point registers "Fd[rs1] and Fd[rs2]" or "Fd[rs1] and Fsimm8". The results are written to the floating-point register Fd[rd]. The comparison results for these elements are written to Fd[rd] from the MSB, and 0s are written to the other bits.

A 64-bit input register includes elements corresponding to the data type. The number of elements and bit range of the elements corresponding to the data type are shown in Table 7-6 and Table 7-7.

Table 7-6 Number of elements and their bit range for each data type (4-bit)

Data type	Number of elements	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6	Element 7	Element 8
4-bit signed integer	16	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
4-bit unsigned integer	16	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32

Data type	Element9	Element 10	Element 11	Element 12	Element 13	Element 14	Element 15	Element 16
4-bit signed integer	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
4-bit unsigned integer	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0

Table 7-7 Number of elements and their bit range for each data type (8-bit, 16-bit, 32-bit, and 64-bit)

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	—	—	—	—	—	—	—

The elements of "Fd[rs1] and Fd[rs2]" (or "Fd[rs1] and Fsimm8"), which are in the same position, are compared. The results are then written to the corresponding bits of Fd[rd]. The bits corresponding to each element of Fd[rd] are shown in Table 7-8 and Table 7-9.

Note The results are written to Fd[rd] from the MSB. In this specification, these instructions are called "SIMD compare type A". See also "SIMD compare type B (page 84)".

Table 7-8 Element and their corresponding bit positions in Fd[rd] (4-bit data)

	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

	Element 9	Element 10	Element 11	Element 12	Element 13	Element 14	Element 15	Element 16
Fd[rd]	55	54	53	52	51	50	49	48

Table 7-9 Elements and their corresponding bit positions in Fd[rd] (8-bit, 16-bit, 32-bit, and 64-bit data)

	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

If xar_i = 0, FPCMPLE{4|8|16|32|64}X compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the signed integers. If "elements of Fd[rs1]" ≤ "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPLE{4|8|16|32|64}X compares the elements of Fd[rs1] and

Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the signed integers. If "elements of Fd[rs1]" ≤ "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPGT{4|8|16|32|64}X compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the signed integers. If "elements of Fd[rs1]" > "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPGT{4|8|16|32|64}X compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the signed integers. If "elements of Fd[rs1]" > "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPULE{4|8|16|32|64}X compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≤ "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPULE{4|8|16|32|64}X compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≤ "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPUNE{4|8|16|32|64}X compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≠ "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPUNE{4|8|16|32|64}X compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≠ "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPUGT{4|8|16|32|64}X compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" > "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPUGT{4|8|16|32|64}X compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" > "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPUEQ{4|8|16|32|64}X compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" = "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPUEQ{4|8|16|32|64}X compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" = "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0.

Note Instructions that compare whether signed integers are equal or not are not defined. These comparisons are equivalent to the instructions FPCMPUEQ{4|8|16|32|64}X and FPCMPUNE{4|8|16|32|64}X, which compare whether unsigned integers are equal or not, respectively.

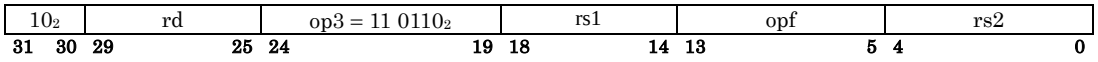
These instructions will not update any fields in the FSR.

Note To use these instructions, "XAR.v must be 0" or "XAR.v must be 1 and XAR.urs3<1:0> must be 00_2". If XAR.v is 1 and XAR.urs3<1:0> is 10_2, FPCMP*{4|8|16|32|64}FX will be executed (page 84). If XAR.v is 1 and XAR.urs3<1:0> is 11_2, FPCMP*{4|8|16|32|64}XACC will be executed(page 89). If XAR.v is 1 and XAR.urs3<1:0> is 01_2, *illegal_action* will occur.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> = 01_2 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.143. Partitioned Shift

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FSLL32[XII]	0 0010 0101_2	32-bit partitioned shift left	✓	✓	`fsll32` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FSRL32[XII]	0 0010 0111_2	32-bit partitioned shift right logical	✓	✓	`fsrl32` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FSRA32[XII]	0 0010 1111_2	32-bit partitioned shift right arithmetic	✓	✓	`fsra32` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$



Description

These instructions shift right or left the upper 32 bits and the lower 32 bits of Fd[rs1], and store the result into Fd[rd]. The shift count is specified by Fd[rs2] if xar_i = 0 and by Fsimm8 if xar_i = 1.

If xar_i = 0, the shift count of the upper 32 bits and the lower 32 bits of Fd[rs1] is specified by Fd[rs2]<36:32> and Fd[rs2]<4:0> respectively. If xar_i = 1, the shift count of the upper 32 bits and the lower 32 bits of Fd[rs1] is specified by Fsimm8_32x2<36:32> and Fsimm8_32x2<4:0> respectively (in case of these instructions, Fsimm8_32x2<63:37, 31:5> is ignored). The operation is illustrated in Figure 7-6.

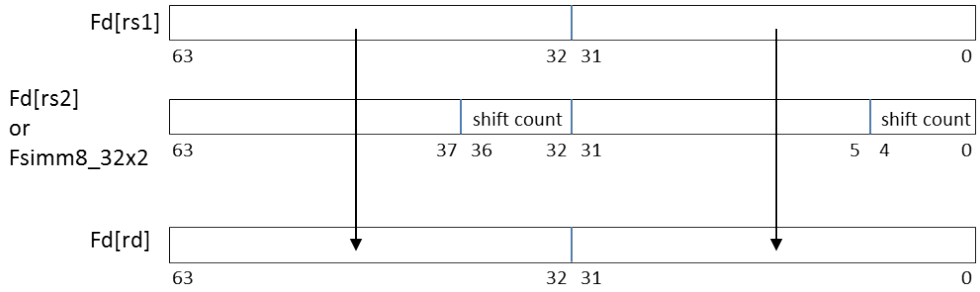


Figure 7-6 The behavior of F{SLL|SRL|SRA}32

FSLL32 shifts the upper and the lower 32 bits of Fd[rs1] left (toward the higher-order), replacing the right (low-order) vacated positions with 0 and stores the result into Fd[rd].

FSRL32 shifts the upper and the lower 32 bits of Fd[rs1] right (toward the lower-order), replacing the left (high-order) vacated positions with 0 and stores the result into Fd[rd].

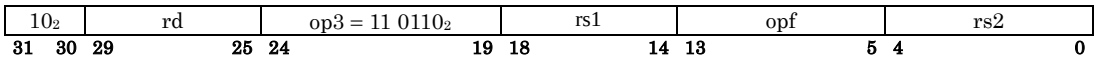
FSRA32 shifts the upper and the lower 32 bits of Fd[rs1] right (toward the lower-order), replacing the left (high-order) vacated positions with the value of Fd[rs1]<63> and Fd[rs1]<31> respectively and stores the result into Fd[rd].

F{SLL|SRL|SRA}32 will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.144. Partitioned Multiply

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FPMUL32[XII]	0 0100 1111$_2$	32-bit partitioned multiply	✓	✓	`fpmul32` *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPMUL64[XII]	0 0100 1110$_2$	64-bit partitioned multilpy	✓	✓	`fpmul64` *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*



Description

Multiplication for 32-bit integers or 64-bit integers stored in floating-point registers.

If xar_i = 0, FPMUL32 multiplies the two 32-bit integers in the same position of Fd[rs1] and Fd[rs2]. The lower 32 bits of the results will be stored in the same position of Fd[rd]. If xar_i = 1, FPMUL32 multiplies the two 32-bit integers in the same position of Fd[rs1] and Fsimm8_32x2. The lower 32 bits of the results will be stored in the same position of Fd[rd]. The operation is illustrated in Figure 7-7.

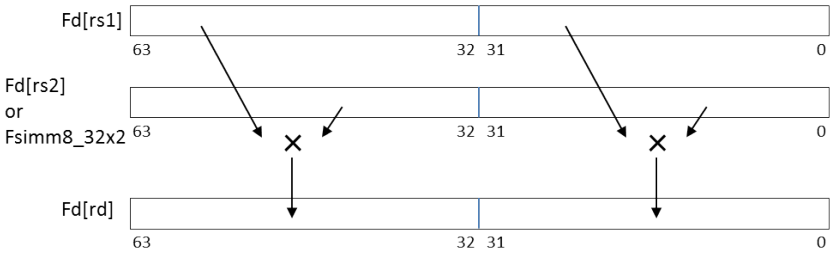


Figure 7-7 The behavior of FPMUL32

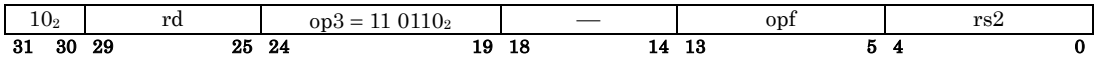
If xar_i = 0, FPMUL64 multiplies the 64-bit integers of Fd[rs1] and Fd[rs2]. The lower 64 bits of the result will be stored in Fd[rd]. If xar_i = 1, FPMUL64 multiplies the 64-bit integers of Fd[rs1] and Fsimm8_64x1. The lower 64 bits of the result will be stored in Fd[rd].

FPMUL32 and FPMUL64 will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.145. Integer Sign/Zero Extension

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FSEXTW[XII]	$1\ 0000\ 0000_2$	Sign extension for 32-bit integers in double floating-point registers	✓	✓	`fsextw` *freg_or_fsimm*, *freg*$_{rd}$
FZEXTW[XII]	$1\ 0000\ 0001_2$	Zero extension for 32-bit integers in double floating-point registers	✓	✓	`fzextw` *freg_or_fsimm*, *freg*$_{rd}$



Description

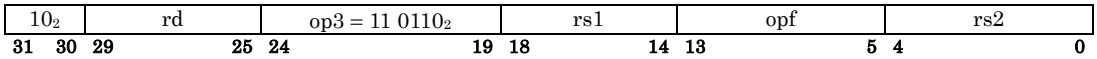
FSEXTW extends the lower 32 bits of the input integer to the sign-extended 64-bit integer and stores the result in Fd[rd]. If xar_i = 0, Fd[rs2]<31:0> is copied into Fd[rd]<31:0> and Fd[rd]<63:32> is filled with Fd[rs2]<31>. If xar_i = 1, Fsimm8_64x1 is copied to Fd[rd]<63:0>.

FZEXTW extends the lower 32 bits of the input integer to the zero-extended 64-bit integer and stores the result in Fd[rd]. If xar_i = 0, Fd[rs2]<31:0> is copied into Fd[rd]<31:0> and Fd[rd]<63:32> is zero-filled. If xar_i = 1, Fsimm8_64x1<31:0> is copied to Fd[rd]<31:0> and Fd[rd]<63:32> is zero-filled.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_instruction	All	iw<18:14> ≠ 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.146. Fixed-Point Partitioned Add (8-bit)

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FPADD8[XII]	$1\ 0010\ 0100_2$	Eight 8-bit adds	✓	✓	`fpadd8` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$



Description

Addition for 8-bit integers stored in a floating-point register.

If xar_i = 0, FPADD8 adds each element in the same 8-bit integer position of Fd[rs1] to Fd[rs2]. The results are stored in the same position in Fd[rd].

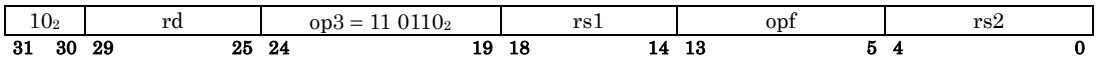
If xar_i = 1, FPADD8 adds each element in the same 8-bit integer position of Fd[rs1] to Fsimm8_8x8. The results are stored in the same position in Fd[rd].

FPADD8 will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.147. Fixed-Point Partitioned Subtract (8-bit)

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FPSUB8^{XII}	1 0101 0100$_2$	Eight 8-bit subtracts	✓	✓	`fpsub8` *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*



Description

Subtraction for 8-bit integers stored in a floating-point register.

If xar_i = 0, FPSUB8 subtracts each element in the same 8-bit integer position of Fd[rs2] from Fd[rs1]. The results are stored in the same position in Fd[rd].

If xar_i = 1, FPSUB8 subtracts each element in the same 8-bit integer position of Fsimm8_8x8 from Fd[rs1]. The results are stored in the same position in Fd[rd].

FPSUB8 will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.148. Full Element Permutation

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FEPERM32X[XII]	1 1000 0100_2	Sorts 32-bit data among double floating-point registers	✓	✓	`feperm32x` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FEPERM64X[XII]	1 1000 0101_2	Sorts 64-bit data among double floating-point registers	✓	✓	`feperm64x` *freg*$_{rs1}$, *freg_or_fsimm*, *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

non-SIMD operation

FEPERM32X and FEPERM64X are mainly used to permutate or mask the SIMD data. These instructions can be used in non-SIMD operations but the purpose is different from SIMD operations.

If xar_i = 0, FEPERM32X copies one of the 32-bit data ((1) - (3) as stated below) to Fd[rd]<63:32> according to Fd[rs2]<63, 32>, and to Fd[rd]<31:0> according to Fd[rs2]<31, 0>.

(1) data in Fd[rs1]<63:32>

(2) data in Fd[rs1]<31:0>

(3) all 0

The behavior of FEPERM32X is described in Figure 7-8, Table 7-10, and Table 7-11. The value of Fd[rs2]<62:33, 30:1> is ignored.

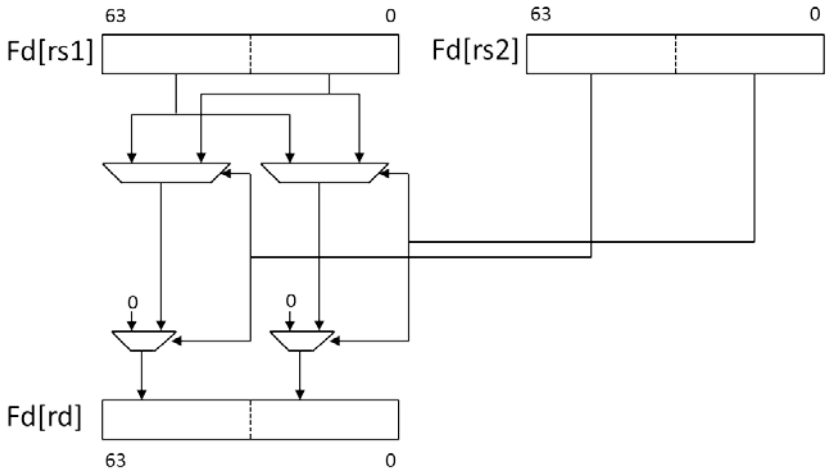


Figure 7-8 Behavior of FEPERM32X (xar_i = 0)

Table 7-10 Results of FEPERM32X (Fd[rd]<63:32>, xar_i = 0)

Fd[rs2]<63>	Fd[rs2]<32>	Fd[rd]<63:32>
0	0	Fd[rs1]<63:32>
	1	Fd[rs1]<31:0>
1	–	all 0

Table 7-11 Results of FEPERM32X (Fd[rd]<31:0>, xar_i = 0)

Fd[rs2]<31>	Fd[rs2]<0>	Fd[rd]<31:0>
0	0	Fd[rs1]<63:32>
	1	Fd[rs1]<31:0>
1	–	all 0

If xar_i = 1, FEPERM32X broadcasts one of the 32-bit data ((1) - (3) as stated below) to Fd[rd]<63:32> and Fd[rd]<31:0> according to Fsimm8<7, 0>.

(1) data in Fd[rs1]<63:32>

(2) data in Fd[rs1]<31:0>

(3) all 0

The behavior of FEPERM32X is described in Figure 7-9 and Table 7-12. The value of Fsimm8<6:1> is ignored.

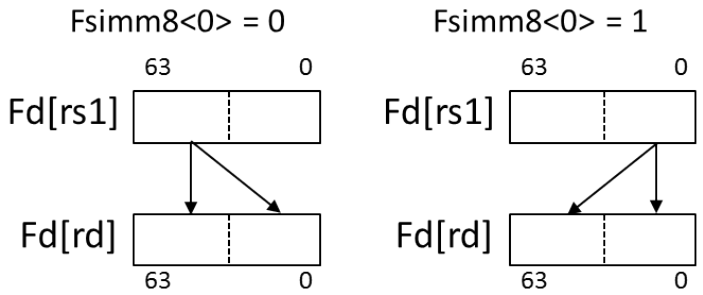


Figure 7-9 Behavior of FEPERM32X (xar_i = 1)

Table 7-12 Results of FEPERM32X (xar_i = 1)

Fsimm8<7>	Fsimm8<0>	Fd[rd]<63:32>, Fd[rd]<31:0>
0	0	Fd[rs1]<63:32>
	1	Fd[rs1]<31:0>
1	–	all 0

If xar_i = 0, FEPERM64X copies one of the 64-bit data ((1) or (2) as stated below) to Fd[rd]<63:0> according to Fd[rs2]<63>.

(1) data in Fd[rs1]<63:0>

(2) all 0

The behavior of FEPERM64X is described in Figure 7-10 and Table 7-13. The value of Fd[rs2]<62:0> is ignored.

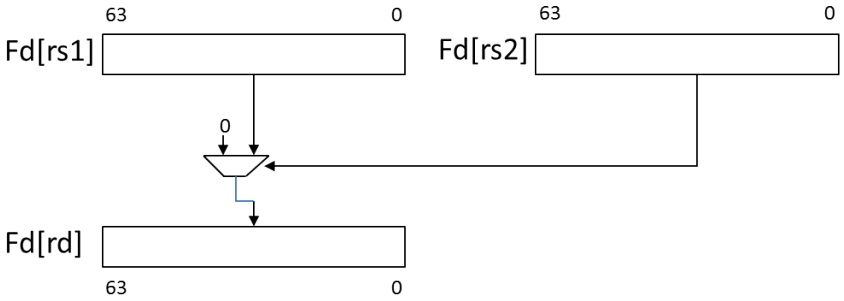


Figure 7-10 Behavior of FEPERM64X (xar_i = 0)

Table 7-13 Results of FEPERM64X (xar_i = 0)

Fd[rs2]<63>	Fd[rd]<63:0>
0	Fd[rs1]<63:0>
1	all 0

If xar_i = 1 and Fsimm8<7> = 0, FEPERM64X copies Fd[rs1] to Fd[rd]. If if xar_i = 1 and Fsimm8<7> = 1, Fd[rd] is filled with 0.

These instructions will not update any fields in the FSR.

SIMD operation

In this section 7.148, Fd[rs1][BASIC] and Fd[rs1][EXTEND] mean Fd[rs1] and Fd[rs1 + 256] respectively. The same applies to Fd[rs2] and Fd[rd].

If xar_i = 0, FEPERM32X copies one of the 32-bit data ((1) - (5) as stated below) to Fd[rd]<63:32>[BASIC] according to Fd[rs2]<63, 33:32>[BASIC], and to Fd[rd]<31:0>[BASIC] according to Fd[rs2]<31, 1:0>[BASIC].

(1) data in Fd[rs1][BASIC]<63:32>

(2) data in Fd[rs1][BASIC]<31:0>

(3) data in Fd[rs1][EXTEND]<63:32>

(4) data in Fd[rs1][EXTEND]<31:0>

(5) all 0

The behavior of FEPERM32X is described in Figure 7-11, Table 7-14, and Table 7-15. The value of Fd[rs2]<62:34, 30:2>[BASIC] is ignored.

The same applies to Fd[rs1][EXTEND], Fd[rs2][EXTEND], and Fd[rd][EXTEND].

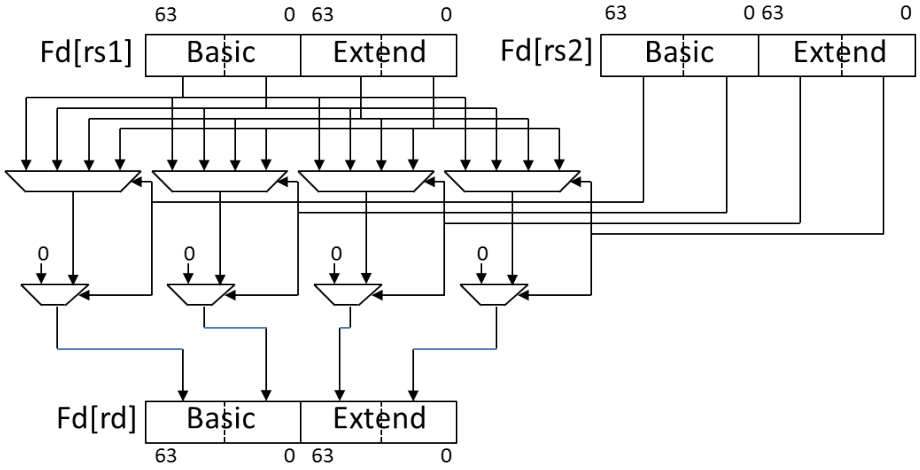


Figure 7-11 Behavior of FEPERM32X (xar_i = 0)

Table 7-14 Results of FEPERM32X (Fd[rd]<63:32>[BASIC], xar_i = 0)

Fd[rs2]<63>[BASIC]	Fd[rs2]<33:32>[BASIC]	Fd[rd]<63:32>[BASIC]
0	0	Fd[rs1][BASIC]<63:32>
	1	Fd[rs1][BASIC]<31:0>
	2	Fd[rs1][EXTEND]<63:32>
	3	Fd[rs1][EXTEND]<31:0>
1	–	all 0

Table 7-15 Results of FEPERM32X (Fd[rd]<31:0>[BASIC], xar_i = 0)

Fd[rs2]<31>[BASIC]	Fd[rs2]<1:0>[BASIC]	Fd[rd]<31:0>[BASIC]
0	0	Fd[rs1][BASIC]<63:32>
	1	Fd[rs1][BASIC]<31:0>
	2	Fd[rs1][EXTEND]<63:32>
	3	Fd[rs1][EXTEND]<31:0>
1	–	all 0

If xar_i = 1, FEPERM32X broadcasts one of the 32-bit data ((1) - (5) as stated below) to Fd[rd]<63:32>[BASIC] and Fd[rd]<31:0>[BASIC] according to Fsimm8<7, 1:0>.

(1) data in Fd[rs1][BASIC]<63:32>

(2) data in Fd[rs1][BASIC]<31:0>

(3) data in Fd[rs1][EXTEND]<63:32>

(4) data in Fd[rs1][EXTEND]<31:0>

all 0

The behavior of FEPERM32X is described in Figure 7-12 and Table 7-16. The value of Fsimm8<6:2> is ignored.

The same applies to Fd[rs1][EXTEND] and Fd[rd][EXTEND].

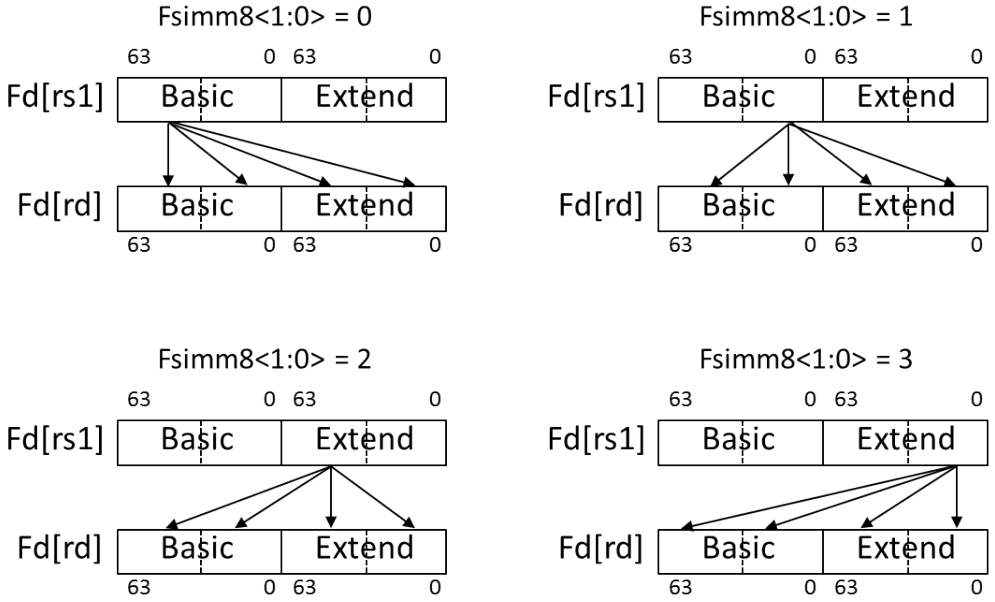


Figure 7-12 Behavior of FEPERM32X (xar_i = 1)

Table 7-16 Results of FEPERM32X (Fd[rd]<63:32, 31:0>[BASIC], xar_i = 1)

Fsimm8<7>	Fsimm8<1:0>	Fd[rd]<63:32>[BASIC], Fd[rd]<31:0>[BASIC]
0	0	Fd[rs1][BASIC]<63:32>
	1	Fd[rs1][BASIC]<31:0>
	2	Fd[rs1][EXTEND]<63:32>
	3	Fd[rs1][EXTEND]<31:0>
1	–	all 0

If xar_i = 0, FEPERM64X copies one of the 64-bit data ((1) - (3) as stated below) to Fd[rd]<63:0>[BASIC] according to Fd[rs2]<63, 0>[BASIC].

(1) data in Fd[rs1][BASIC]<63:0>

(2) data in Fd[rs1][EXTEND]<63:0>

(3) all 0

The behavior of FEPERM64X is described in Figure 7-13 and Table 7-17. The value of Fd[rs2]<62:1>[BASIC] is ignored.

The same applies to Fd[rs1][EXTEND], Fd[rs2][EXTEND], and Fd[rd][EXTEND].

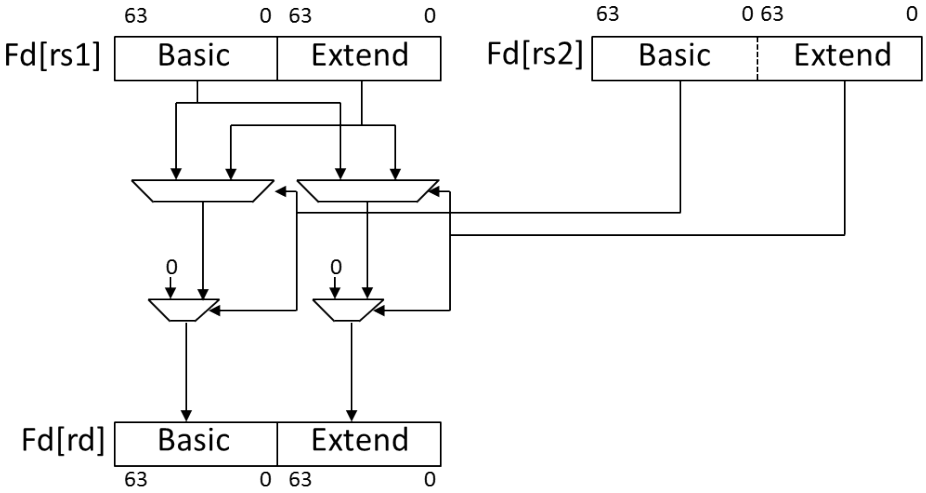


Figure 7-13 Behavior of FEPERM64X (xar_i = 0)

Table 7-17 Results of FEPERM64X (Fd[rd]<63:0>[BASIC], xar_i = 0)

Fd[rs2]<63>[BASIC]	Fd[rs2]<0>[BASIC]	Fd[rd]<63:0>[BASIC]
0	0	Fd[rs1][BASIC]<63:0>
	1	Fd[rs1][EXTEND]<63:0>
1	–	all 0

If xar_i = 1, FEPERM64X broadcasts one of the 64-bit data ((1) - (3) as stated below) to Fd[rd]<63:0>[BASIC] according to Fsimm8<7, 0>.

(1) data in Fd[rs1][BASIC]<63:0>

(2) data in Fd[rs1][EXTEND]<63:0>

(3) all 0

The behavior of FEPERM64X is described in Figure 7-14 and Table 7-18. The value of Fsimm8<6:1> is ignored.

The same applies to Fd[rs1][EXTEND] and Fd[rd][EXTEND].

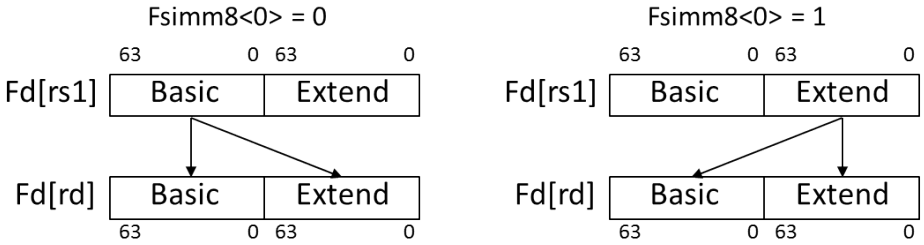


Figure 7-14 Behavior of FEPERM64X (xar_i = 1)

Table 7-18 Results of FEPERM64X (Fd[rd]<63:0>[BASIC], xar_i = 1)

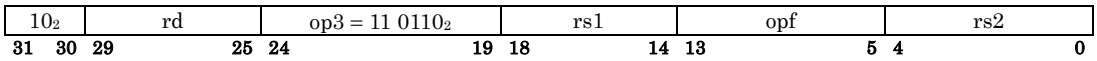
Fsimm8<7>	Fsimm8<0>	Fd[rd]<63:0>[BASIC]
0	0	Fd[rs1][BASIC]<63:0>
	1	Fd[rs1][EXTEND]<63:0>
1	–	all 0

These instructions will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.149. Partition Concatenate Shift Left

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FPCSL8X[XII]	0 1001 1110$_2$	Concatenates two registers and shifts left	✓	✓	fpcsl8x *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*



Description FPCSL8X concatenates the lower bits of Fd[rd] and the upper bits of Fd[rs1] to form a 64-bit value and stores it in Fd[rd].

Non-SIMD operation

If xar_i = 0, "shift_amount" is specified by "Fd[rs2]<2:0> × 8" bits and if xar_i = 1, "shift_amount" is specified by "Fsimm8<2:0> × 8" bits. Fd[rs2]<63:3> and Fsimm8<63:3> are ignored.

If shift_amount is not 0, FPCSL8X concatenates Fd[rd]<63 – shift_amount:0> and Fd[rs1]<63:63 – shift_amount + 1>, and stores it in Fd[rd]<63:0>. If shift_amount is 0, the value of Fd[rd] remains unchanged.

The operation is illustrated in Figure 7-15.

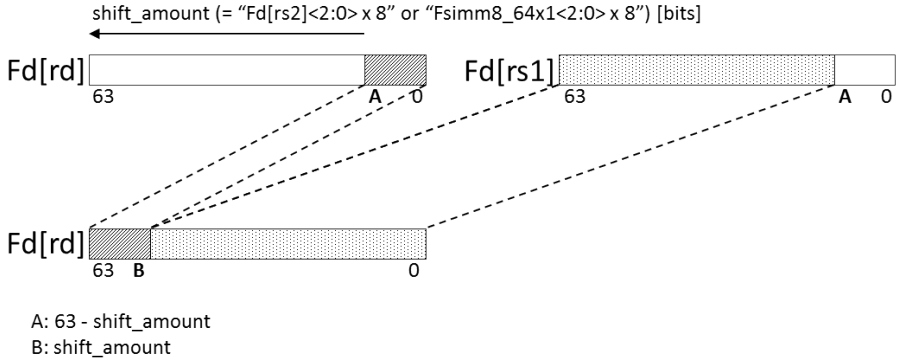


Figure 7-15 Behavior of FPCSL8X (non-SIMD)

SIMD operation

In this section 7.149, Fd[rs1][BASIC] and Fd[rs1][EXTEND] mean Fd[rs1] and Fd[rs1 + 256] respectively. The same applies to Fd[rs2] and Fd[rd].

For the basic side, if xar_i = 0, "shift_amount" is specified by "Fd[rs2][BASIC]<2:0> × 8" bits and if xar_i = 1, "shift_amount" is specified by "Fsimm8<2:0> × 8" bits. Fd[rs2][BASIC]<63:3> and Fsimm8<63:3> are ignored.

If shift_amount is not 0, FPCSL8X concatenates Fd[rd][BASIC]<63 – shift_amount:0> and Fd[rs1][BASIC]<63:63 – shift_amount + 1>, and stores it in Fd[rd][BASIC]<63:0>. If shift_amount is 0, the value of Fd[rd][BASIC] remains unchanged.

For the extended side, if xar_i = 0, "shift_amount" is specified by "Fd[rs2][EXTEND]<2:0> × 8" and if xar_i = 1, "shift_amount" is specified by "Fsimm8<2:0> × 8". Fd[rs2][EXTEND]<63:3> and Fsimm8<63:3> are ignored.

The behavior of the extended side is the same as that of basic side.

The operation is illustrated in Figure 7-16.

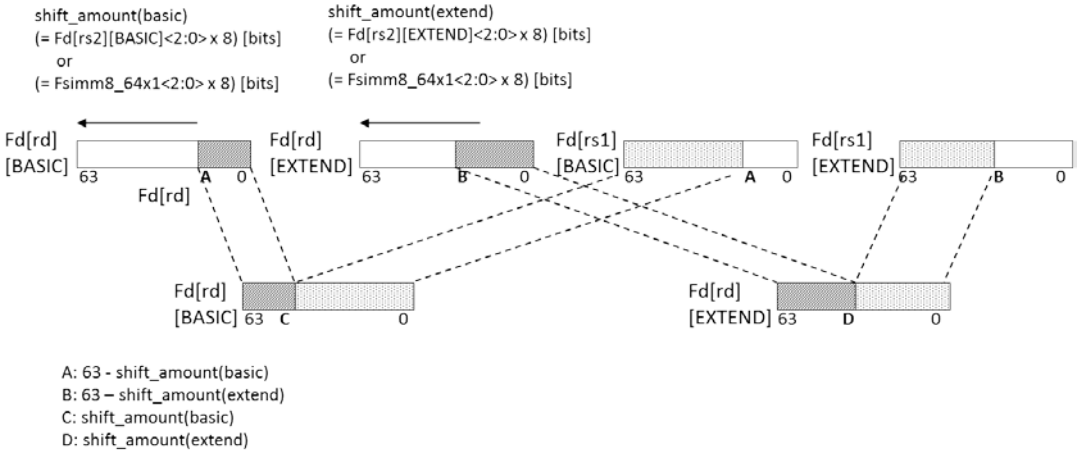


Figure 7-16 Behavior of FPCSL8X (SIMD)

FPCSL8X will not update any fields in the FSR.

Programming Note FPCSL8X is mainly used to support unaligned loads. Refer to the pseudo-code below.

```
/* pseudo-code */

   x = addr & ~(8-1)
   y = addr & (8-1)
   ldd,s data0, [x]
   ldd,s data1, [x+8]
   fpcsl8x,s data0, data1, y
```

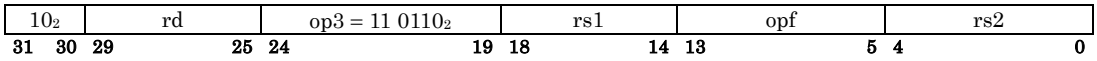
Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.150. SIMD Compare (type B)

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPCMPLE16FX[XII]	0 1100 0000$_2$	10$_2$	Compares four 16-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple16fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPULE16FX[XII]	0 1100 0001$_2$	10$_2$	Compares four 16-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule16fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPLE4FX[XII]	0 1100 0010$_2$	10$_2$	Compares sixteen 4-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple4fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPUNE16FX[XII]	0 1100 0011$_2$	10$_2$	Compares four 16-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	fpcmpune16fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPLE32FX[XII]	0 1100 0100$_2$	10$_2$	Compares two 32-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple32fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPULE32FX[XII]	0 1100 0101$_2$	10$_2$	Compares two 32-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule32fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPULE4FX[XII]	0 1100 0110$_2$	10$_2$	Compares sixteen 4-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule4fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPUNE32FX[XII]	0 1100 0111$_2$	10$_2$	Compares two 32-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	fpcmpune32fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPGT16FX[XII]	0 1100 1000$_2$	10$_2$	Compares four 16-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt16fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPUGT16FX[XII]	0 1100 1001$_2$	10$_2$	Compares four 16-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt16fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPUEQ16FX[XII]	0 1100 1011$_2$	10$_2$	Compares four 16-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	fpcmpueq16fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPGT32FX[XII]	0 1100 1100$_2$	10$_2$	Compares two 32-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt32fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*
FPCMPUGT32FX[XII]	0 1100 1101$_2$	10$_2$	Compares two 32-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt32fx *freg$_{rs1}$*, *freg_or_fsimm*, *freg$_{rd}$*

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPCMPUNE4FX[XII]	0 1100 1110_2	10_2	Compares sixteen 4-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	`fpcmpune4fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUEQ32FX[XII]	0 1100 1111_2	10_2	Compares two 32-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	`fpcmpueq32fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPLE8FX[XII]	0 1101 0000_2	10_2	Compares eight 8-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	`fpcmple8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPULE8FX[XII]	0 1101 0001_2	10_2	Compares eight 8-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	`fpcmpule8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPGT4FX[XII]	0 1101 0010_2	10_2	Compares sixteen 4-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	`fpcmpgt4fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE8FX[XII]	0 1101 0011_2	10_2	Compares eight 8-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	`fpcmpune8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPLE64FX[XII]	0 1101 0100_2	10_2	Compares 64-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	`fpcmple64fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPULE64FX[XII]	0 1101 0101_2	10_2	Compares 64-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	`fpcmpule64fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUGT4FX[XII]	0 1101 0110_2	10_2	Compares sixteen 4-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	`fpcmpugt4fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE64FX[XII]	0 1101 0111_2	10_2	Compares 64-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	`fpcmpune64fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPGT8FX[XII]	0 1101 1000_2	10_2	Compares eight 8-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	`fpcmpgt8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUGT8FX[XII]	0 1101 1001_2	10_2	Compares eight 8-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	`fpcmpugt8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUEQ8FX[XII]	0 1101 1011_2	10_2	Compares eight 8-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	`fpcmpueq8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPGT64FX[XII]	0 1101 1100_2	10_2	Compares 64-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	`fpcmpgt64fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUGT64FX[XII]	0 1101 1101_2	10_2	Compares 64-bit unsigned integers	※	✓	`fpcmpugt64fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE Regs	HPC-ACE SIMD	Assembly Language Syntax
			If src1 > src2, the corresponding result is 1.			
FPCMPUEQ4FX[XII]	0 1101 1110$_2$	10$_2$	Compares sixteen 4-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	`fpcmpueq4fx` *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUEQ64FX[XII]	0 1101 1111$_2$	10$_2$	Compares 64-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	`fpcmpueq64fx` *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*



Description

These instructions compare several elements (partitions) in the two floating-point registers "Fd[rs1] and Fd[rs2]" or "Fd[rs1] and Fsimm8". The results are written to the floating-point register Fd[rd].

A 64-bit input register includes elements corresponding to the data type. The number of elements and bit positions of the elements corresponding to the data type are shown in Table 7-19 and Table 7-20.

Table 7-19 Number of elements and the bit position of the elements (4-bit data)

Data type	Number of elements	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6	Element 7	Element 8
4-bit signed integer	16	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
4-bit unsigned integer	16	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32

Data type	Element9	Element 10	Element 11	Element 12	Element 13	Element 14	Element 15	Element 16
4-bit signed integer	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
4-bit unsigned integer	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0

Table 7-20 Number of elements and the bit position of the elements corresponding to the data type (8-bit, 16-bit, 32-bit, and 64-bit)

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	—	—	—	—	—	—	—

The elements of "Fd[rs1] and Fd[rs2]" (or "Fd[rs1] and Fsimm8"), which are in the same position, are compared. The results are then written to the corresponding elements of Fd[rd]. The bits corresponding to the element of Fd[rd] are set to all 0 or all 1. For example, with 32-bit unsigned integers, 0x00000000 or 0xffffffff is set to Fd[rd]<63:32> and Fd[rd]<31:0> according to the compared results of each element.

> **Note** The results (all 0 or all 1) are written to the corresponding elements of Fd[rd]. In this specification, these instructions are called "SIMD compare type B". See also "SIMD compare type A (page 63)".

If xar_i = 0, FPCMPLE{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the signed integers. If "elements of Fd[rs1]" ≤ "elements of Fd[rs2]", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPLE{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the signed integers. If "elements of Fd[rs1]" ≤ "elements of Fsimm8", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPGT{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the signed integers. If "elements of Fd[rs1]" > "elements of Fd[rs2]", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPGT{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the signed integers. If "elements of Fd[rs1]" > "elements of Fsimm8", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPULE{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≤ "elements of Fd[rs2]", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPULE{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned

integers. If "elements of Fd[rs1]" ≤ "elements of Fsimm8", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPUNE{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≠ "elements of Fd[rs2]", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPUNE{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≠ "elements of Fsimm8", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPUGT{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" > "elements of Fd[rs2]", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPUGT{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" > "elements of Fsimm8", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0.

If xar_i = 0, FPCMPUEQ{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" = "elements of Fd[rs2]", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0. If xar_i = 1, FPCMPUEQ{4|8|16|32|64}FX compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" = "elements of Fsimm8", the corresponding elements of Fd[rd] are all set to 1, otherwise they are all set to 0.

These instructions will not update any field in the FSR.

Note To use these instructions, XAR.v must be 1 and XAR.urs3<1:0> must be 10_2. If "XAR.v is 0" or "XAR.v is 1 and XAR.urs3<1:0> is 00_2", FPCMP*{4|8|16|32|64}X will be executed (page 63). If XAR.v is 1 and XAR.urs3<1:0> is 11_2, FPCMP*{4|8|16|32|64}XACC will be executed (page 89). If XAR.v is 1 and XAR.urs3<1:0> is 01_2, *illegal_action* will occur.

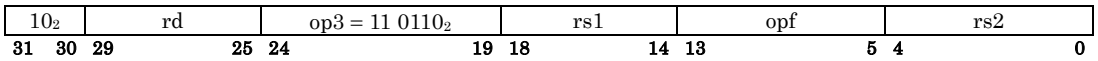
Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> = 01_2 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.151. SIMD Compare and Accumulate Results

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPCMPLE16XACC[XII]	0 1100 0000_2	11_2	Compares four 16-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple16xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPULE16XACC[XII]	0 1100 0001_2	11_2	Compares four 16-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule16xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPLE4XACC[XII]	0 1100 0010_2	11_2	Compares sixteen 4-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple4xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE16XACC[XII]	0 1100 0011_2	11_2	Compares four 16-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	fpcmpune16xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPLE32XACC[XII]	0 1100 0100_2	11_2	Compares two 32-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple32xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPULE32XACC[XII]	0 1100 0101_2	11_2	Compares two 32-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule32xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPULE4XACC[XII]	0 1100 0110_2	11_2	Compares sixteen 4-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule4xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE32XACC[XII]	0 1100 0111_2	11_2	Compares two 32-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	fpcmpune32xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPGT16XACC[XII]	0 1100 1000_2	11_2	Compares four 16-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt16xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUGT16XACC[XII]	0 1100 1001_2	11_2	Compares four 16-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt16xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUEQ16XACC[XII]	0 1100 1011_2	11_2	Compares four 16-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	fpcmpueq16xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPGT32XACC[XII]	0 1100 1100_2	11_2	Compares two 32-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt32xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUGT32XACC[XII]	0 1100 1101_2	11_2	Compares two 32-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt32xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPCMPUNE4XACC[XII]	0 1100 1110_2	11_2	Compares sixteen 4-bit unsigned integers If src1 ≠ src2, the	※	✓	fpcmpune4xacc *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
			corresponding result is 1.			
FPCMPUEQ32XACC[XII]	0 1100 1111_2	11_2	Compares two 32-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	fpcmpueq32xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPLE8XACC[XII]	0 1101 0000_2	11_2	Compares eight 8-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple8xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPULE8XACC[XII]	0 1101 0001_2	11_2	Compares eight 8-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule8xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPGT4XACC[XII]	0 1101 0010_2	11_2	Compares sixteen 4-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt4xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUNE8XACC[XII]	0 1101 0011_2	11_2	Compares eight 8-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	fpcmpune8xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPLE64XACC[XII]	0 1101 0100_2	11_2	Compares 64-bit signed integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmple64xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPULE64XACC[XII]	0 1101 0101_2	11_2	Compares 64-bit unsigned integers If src1 ≤ src2, the corresponding result is 1.	※	✓	fpcmpule64xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUGT4XACC[XII]	0 1101 0110_2	11_2	Compares sixteen 4-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt4xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUNE64XACC[XII]	0 1101 0111_2	11_2	Compares 64-bit unsigned integers If src1 ≠ src2, the corresponding result is 1.	※	✓	fpcmpune64xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPGT8XACC[XII]	0 1101 1000_2	11_2	Compares eight 8-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt8xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUGT8XACC[XII]	0 1101 1001_2	11_2	Compares eight 8-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt8xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUEQ8XACC[XII]	0 1101 1011_2	11_2	Compares eight 8-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	fpcmpueq8xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPGT64XACC[XII]	0 1101 1100_2	11_2	Compares 64-bit signed integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpgt64xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUGT64XACC[XII]	0 1101 1101_2	11_2	Compares 64-bit unsigned integers If src1 > src2, the corresponding result is 1.	※	✓	fpcmpugt64xacc *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPCMPUEQ4XACC[XII]	0 1101 1110_2	11_2	Compares sixteen 4-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	`fpcmpueq4xacc` *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*
FPCMPUEQ64XACC[XII]	0 1101 1111_2	11_2	Compares 64-bit unsigned integers If src1 = src2, the corresponding result is 1.	※	✓	`fpcmpueq64xacc` *freg$_{rs1}$, freg_or_fsimm, freg$_{rd}$*



Description

These instructions compare several elements (partitions) in the two floating-point registers "Fd[rs1] and Fd[rs2]" or "Fd[rs1] and Fsimm8". The results are written to the floating-point register Fd[rd]. The comparison results for these elements are written to Fd[rd] from the MSB. Before new results are written, the previous results are shifted to the right, that is, the previous results are accumulated.

A 64-bit input register includes elements corresponding to the data type. The number of elements and bit range of the elements corresponding to the data type are shown in Table 7-21 and Table 7-22.

Table 7-21 Number of elements and their bit range for each data type (4-bit data)

Data type	Number of elements	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6	Element 7	Element 8
4-bit signed integer	16	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
4-bit unsigned integer	16	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32

Data type	Element9	Element 10	Element 11	Element 12	Element 13	Element 14	Element 15	Element 16
4-bit signed integer	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
4-bit unsigned integer	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0

Table 7-22 Number of elements and their bit range for each data type (8-bit, 16-bit, 32-bit, and 64-bit)

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	—	—	—	—	—	—	—

The elements of "Fd[rs1] and Fd[rs2]" (or "Fd[rs1] and Fsimm8"), which are in the same position, are compared. The results are then written to the corresponding bits of Fd[rd]. The bits corresponding to each element of Fd[rd] are shown in Table 7-23 and Table 7-24.

Table 7-23 Elements and their corresponding bit positions in Fd[rd] (4-bit data)

	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

	Element 9	Element 10	Element 11	Element 12	Element 13	Element 14	Element 15	Element 16
Fd[rd]	55	54	53	52	51	50	49	48

Table 7-24 Elements and their corresponding bit positions in Fd[rd] (8-bit, 16-bit, 32-bit, and 64-bit data)

	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

Before new results are written to Fd[rd], previous results are shifted to the right as shown in Figure 7-17.

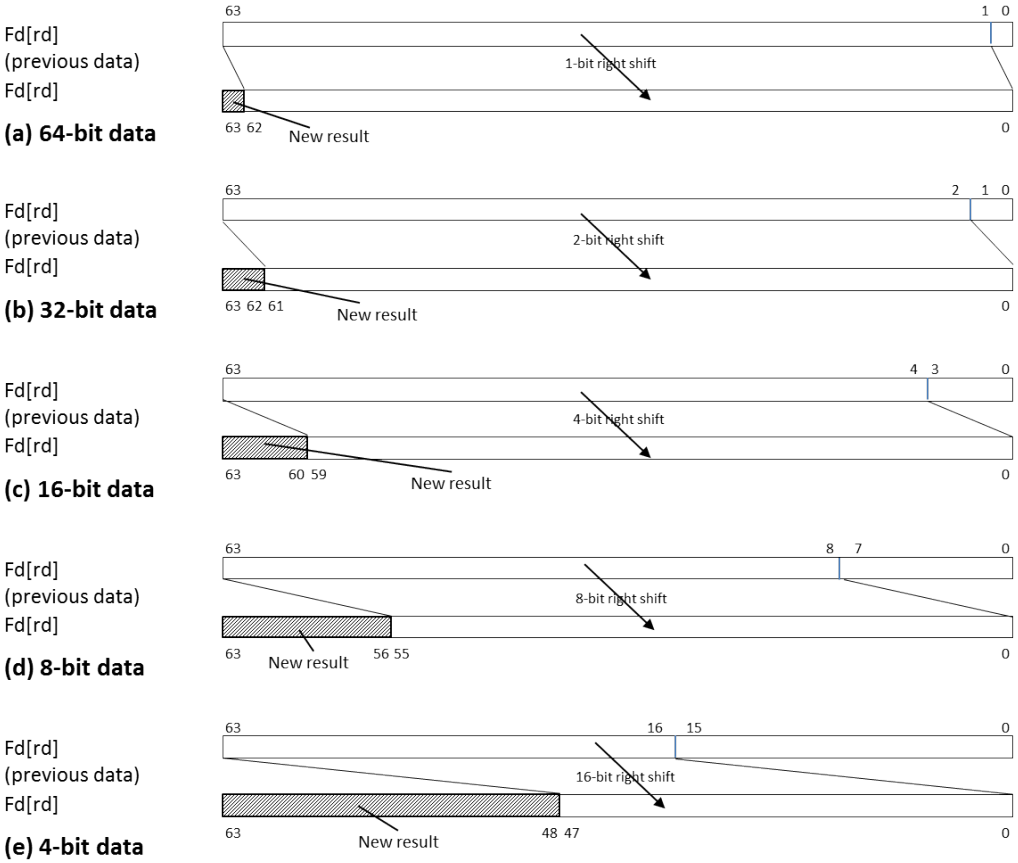


Figure 7-17 shift amount for each data

If xar_i = 0, FPCMPLE{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the signed integers. If "elements of Fd[rs1]" ≤ "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right. If xar_i = 1, FPCMPLE{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the signed integers. If "elements of Fd[rs1]" ≤ "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right.

If xar_i = 0, FPCMPGT{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the signed integers. If "elements of Fd[rs1]" > "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right. If xar_i = 1, FPCMPGT{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the signed integers. If "elements of Fd[rs1]" > "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right.

If xar_i = 0, FPCMPULE{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≤ "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right. If xar_i = 1, FPCMPULE{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≤ "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to

1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right.

If xar_i = 0, FPCMPUNE{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≠ "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right. If xar_i = 1, FPCMPUNE{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" ≠ "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right.

If xar_i = 0, FPCMPUGT{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" > "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right. If xar_i = 1, FPCMPUGT{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" > "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right.

If xar_i = 0, FPCMPUEQ{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fd[rs2], which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" = "elements of Fd[rs2]", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right. If xar_i = 1, FPCMPUEQ{4|8|16|32|64}XACC compares the elements of Fd[rs1] and Fsimm8_{8x8|8x8|16x4|32x2|64x1}, which are in the same position, as the unsigned integers. If "elements of Fd[rs1]" = "elements of Fsimm8", the corresponding bits of Fd[rd] are all set to 1, otherwise they are all set to 0. Before new results are written, previous results are shifted to the right.

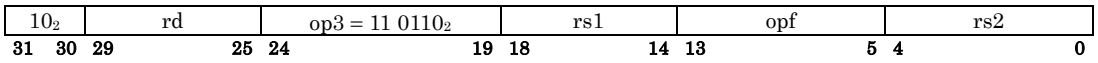
These instructions will not update any fields in the FSR.

Note To use these instructions, XAR.v must be 1 and XAR.urs3<1:0> must be 11_2. If "XAR.v is 0" or "XAR.v is 1 and XAR.urs3<1:0> is 00_2", FPCMP*{4|8|16|32|64}X will be executed (page 63). If XAR.v is 1 and XAR.urs3<1:0> is 10_2, FPCMP*{4|8|16|32|64}FX will be executed (page 84). If XAR.v is 1 and XAR.urs3<1:0> is 01_2, *illegal_action* will occur.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<1:0> = 01_2 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.152. Partitioned Move for Selected Floating-Point Register on Floating-Point Register's Condition (extended for SPARC64™ XII)

Opcode	opf	urs3 <1>	Operation	HPC-ACE		Assembly Language Syntax
				Regs	SIMD	
FPSELMOV8FX[XII]	0 1001 0101_2	1	Select eight 8-bit data from the registers	※	✓	`fpselmov8fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPSELMOV16FX[XII]	0 1001 0110_2	1	Select four 16-bit data from the registers	※	✓	`fpselmov16fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$
FPSELMOV32FX[XII]	0 1001 0111_2	1	Select two 32-bit data from the registers	※	✓	`fpselmov32fx` *freg*$_{rs1}$, *freg_or_fsimm*, *freg*$_{rd}$



Description

For FPSELMOV8FX, if xar_i = 0, the data in Fd[rs2] and Fd[rd] are divided into eight 8-bit data. According to the value of Fd[rs1] (corresponding to each data), each divided 8-bit data in Fd[rs2] or Fd[rd] is selected and stored in Fd[rd]. If the corresponding bit for Fd[rs1] is 1, the data in Fd[rs2] is selected. If it is 0, the data in Fd[rd] is selected.

If xar_i = 1, the data in Fsimm8_8x8 and Fd[rd] are divided into eight 8-bit data. According to the value of Fd[rs1] (corresponding to each data), each divided 8-bit data in Fsimm8_8x8 or Fd[rd] is selected and stored in Fd[rd]. If the corresponding bit for Fd[rs1] is 1, the data in Fsimm8_8x8 is selected. If it is 0, the data in Fd[rd] is selected.

For FPSELMOV16FX, if xar_i = 0, the data in Fd[rs2] and Fd[rd] are divided into four 16-bit data. According to the value of Fd[rs1] (corresponding to each data), each divided 16-bit data in Fd[rs2] or Fd[rd] is selected and stored in Fd[rd]. If the corresponding bit for Fd[rs1] is 1, the data in Fd[rs2] is selected. If it is 0, the data in Fd[rd] is selected.

If xar_i = 1, the data in Fsimm8_16x4 and Fd[rd] are divided into four 16-bit data. According to the value of Fd[rs1] (corresponding to each data), each divided 16-bit data in Fsimm8_16x4 or Fd[rd] is selected and stored in Fd[rd]. If the corresponding bit for Fd[rs1] is 1, the data in Fsimm8_16x4 is selected. If it is 0, the data in Fd[rd] is selected.

For FPSELMOV32FX, if xar_i = 0, the data in Fd[rs2] and Fd[rd] are divided into two 32-bit data. According to the value of Fd[rs1] (corresponding to each data), each divided 32-bit data in Fd[rs2] or Fd[rd] is selected and stored in Fd[rd]. If the corresponding bit for Fd[rs1] is 1, the data in Fd[rs2] is selected. If it is 0, the data in Fd[rd] is selected.

If xar_i = 1, the data in Fsimm8_32x2 and Fd[rd] are divided into two 32-bit data. According to the value of Fd[rs1] (corresponding to each data), each divided 32-bit data in Fsimm8_32x2 or Fd[rd] is selected and stored in Fd[rd]. If the corresponding bit for Fd[rs1] is 1, the data in Fsimm8_32x2 is selected. If it is 0, the data in Fd[rd] is selected.

The bit ranges of Fd[rs2] and Fd[rd] that are selected by Fd[rs1] are shown below.

	Fd[rs1] bit 63	Fd[rs1] bit 55	Fd[rs1] bit 47	Fd[rs1] bit 39	Fd[rs1] bit 31	Fd[rs1] bit 23	Fd[rs1] bit 15	Fd[rs1] bit 7
Corresponding bits of Fd[rs2], Fsimm8, and Fd[rd] for FPSELMOV8FX	<63:56>	<55:48>	<47:40>	<39:32>	<31:24>	<23:16>	<15:8>	<7:0>
Corresponding bits of Fd[rs2], Fsimm8, and Fd[rd] for FPSELMOV16FX	<63:48>		<47:32>		<31:16>		<15:0>	
Corresponding bits of Fd[rs2], Fsimm8, and Fd[rd] for FPSELMOV32FX	<63:32>				<31:0>			

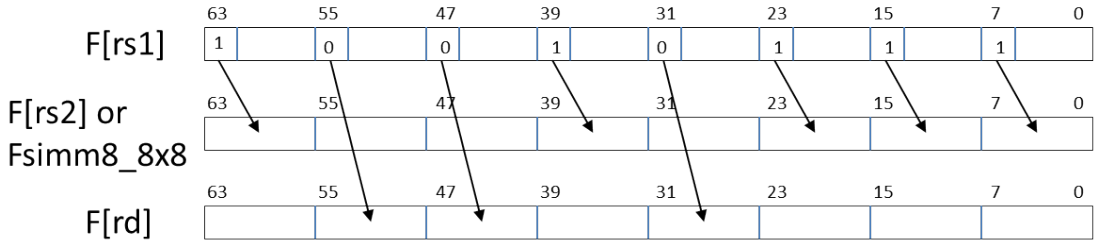


Figure 7-18 Behavior of FPSELMOV8FX (example)

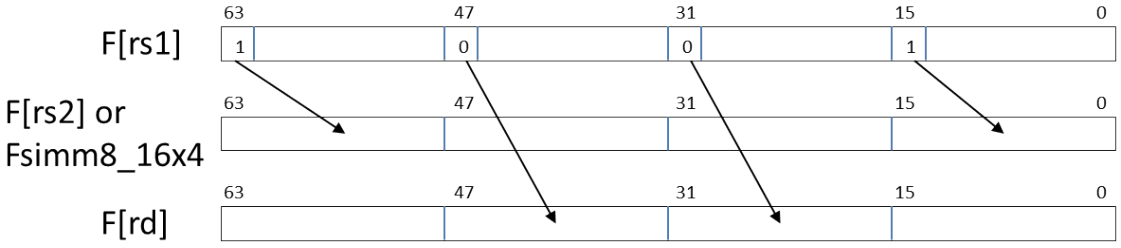


Figure 7-19 Behavior of FPSELMOV16FX (example)

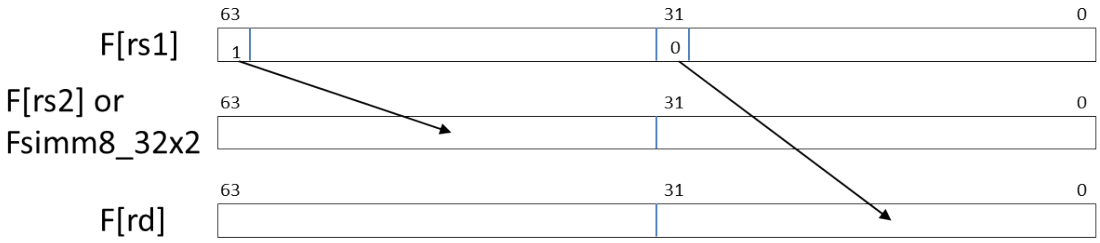


Figure 7-20 Behavior of `FPSELMOV32FX` (example)

These instructions will not update any fields in the FSR.

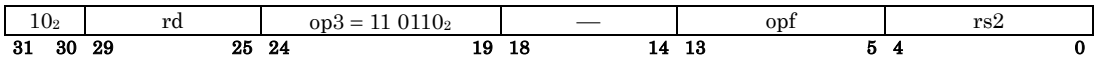
Note The field of Fd[rs1]<62:56, 54:48, 46:40, 38:32, 30:24, 22:16, 14:8, 6:0> for `FPSELMOV8FX`, Fd[rs1]<62:48, 46:32, 30:16, 14:0> for `FPSELMOV16FX`, and Fd[rs1]<62:32, 30:0> for `FPSELMOV32FX` are ignored and have no effect.

Note XAR.v must be 1 and XAR.urs3<1> must be 1 to use these instructions. If "XAR.v is 0" or "XAR.v is 1 and XAR.urs3<1> is 0", `FPSELMOV{8|16|32}X` will be executed (refer to 7.134 in the SPARC64™ X/X+ specification).

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.urs1<1> ≠ 0 • XAR.urs2<1> ≠ 0 and XAR.urs3<2> = 0 • XAR.urs3<0> ≠ 0 • XAR.urd<1> ≠ 0 • XAR.simd = 1 and XAR.urs1<2> ≠ 0 • XAR.simd = 1 and XAR.urs2<2> ≠ 0 and XAR.urs3<2> = 0 • XAR.simd = 1 and XAR.urd<2> ≠ 0

7.153. Move Floating-Point Register to Integer Register

Opcode	opf	urs3 <0>	Operation	HPC-ACE Regs	HPC-ACE SIMD	Assembly Language Syntax
MOVdTOx[XII]	1 0001 0000_2		Copies 64 bits of a double floating-point register to an integer register	✓		`movdtox` *freg*$_{rs2}$, *reg*$_{rd}$
MOVsTOuw[XII]	1 0001 0001	0	Copies 32 bits of a floating-point register to an integer register (without sign-extension)	✓		`movstouw` *freg*$_{rs2}$, *reg*$_{rd}$
MOVsTOsw[XII]	1 0001 0011	0	Copies 32 bits of a floating-point register to an integer register (with sign-extension)	✓		`movstosw` *freg*$_{rs2}$, *reg*$_{rd}$
MOVfwTOuw[XII]	1 0001 0001	1	Copies 32 bits of a double floationg-point register to an integer register (without sign-extension)	※		`movfwtouw` *freg*$_{rs2}$, *reg*$_{rd}$
MOVfwTOsw[XII]	1 0001 0011	1	Copies 32 bits of a double floatint-point register to an integer register (with sign-extension)	※		`movfwtosw` *freg*$_{rs2}$, *reg*$_{rd}$



Description

MOVdTOx copies 64 bits of a double floating-point register Fd[rs2] to a general-purpose register R[rd]. No conversion is performed on the copied 64 bits.

If XAR.v = 0, MOVsTOuw copies 32 bits of a single floating-point register Fs[rs2] to the lower 32 bits of a general-purpose register R[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of R[rd] is set to 0 (without sign-extension).

If XAR.v = 1 and XAR.urs3<0> = 0, MOVsTOuw copies the upper 32 bits of a double floating-point register Fd[rs2] to the lower 32 bits of a general-purpose register R[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of R[rd] is set to 0 (without sign-extension).

If XAR.v = 0, MOVsTOsw copies 32 bits of a single floating-point register Fs[rs2] to the lower 32 bits of a general-purpose register R[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of R[rd] is set to Fs[rs2]<31> (with sign-extension).

If XAR.v = 1 and XAR.urs3<0> = 0, MOVsTOsw copies the upper 32 bits of a double floating-point register Fd[rs2] to the lower 32 bits of a general-purpose register R[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of R[rd] is set to Fd[rs2]<63> (with sign-extension).

MOVfwTOuw copies the lower 32 bits of a double floating-point register Fd[rs2] to the lower 32 bits of a general-purpose register R[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of R[rd] is set to 0 (without sign-extension).

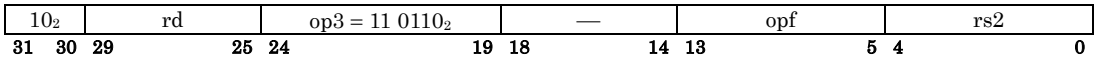
MOVfwTOsw copies the lower 32 bits of a double floating-point register Fd[rs2] to the lower 32 bits of a general-purpose register R[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of R[rd] is set to Fd[rs2]<31> (with sign-extension).

These instructions will not update any fields in the FSR.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_instruction	All	iw<18:14> ≠ 0
illegal_action	MOVdTOx	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3 ≠ 0 • XAR.urd ≠ 0
	MOVsTOuw, MOVsTOsw, MOVfwTOuw, MOVfwTOsw	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2<1> ≠ 0 • XAR.urs3<2:1> ≠ 0 • XAR.urd ≠ 0

7.154. Move Integer Register to Floating-Point Register

Opcode	opf	urs3 <1:0>	Operation	HPC-ACE Regs	HPC-ACE SIMD	Assembly Language Syntax
MOVwTOfuw[XII]	1 0001 1001_2	01_2	Copies the lower 32 bits of an integer register to a double floating-point register (without sign-extension)	※		movwtofuw *reg_{rs2}*, *$freg_{rd}$*
MOVwTOfsw[XII]	1 0001 1001_2	11_2	Copies the lower 32 bits of an integer register to a double floating-point register (with sign-extension)	※		movwtofsw *reg_{rs2}*, *$freg_{rd}$*



Description

MOVwTOfuw copies the lower 32 bits of a general-purpose register R[rs2] to the lower 32 bits of a double floating-point register Fd[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of Fd[rd] is set to 0 (without sign-extension).

MOVwTOfsw copies the lower 32 bits of a general-purpose register R[rs2] to the lower 32 bits of a double floating-point register Fd[rd]. No conversion is performed on the copied 32 bits. The upper 32 bits of Fd[rd] is set to R[rs2]<31> (with sign-extension).

These instructions will not update any fields in the FSR.

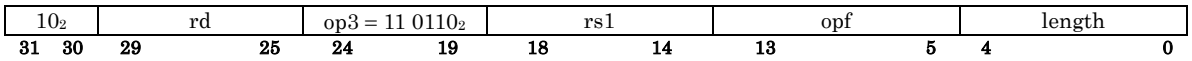
Note To use these instructions, XAR.v must be 1. In addition, XAR.urs3<1:0> must be 01_2 for MOVwTOfuw and must be 11_2 for MOVwTOfsw. In other cases, an another instruction will be executed or an exception will occur as follows.

- XAR.v = 0: MOVwTOs will be executed (refer to 7.142 in the SPARC64™ X/X+ specification).
- XAR.v = 1 and XAR.urs3<1:0> = 00_2: MOVwTOs will be executed (refer to 7.142 in the SPARC64™ X/X+ specification).
- XAR.v = 1 and XAR.urs3<1:0> = 10_2: *illegal_action* will occur.

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_instruction	All	iw<18:14> ≠ 0
illegal_action	All	If XAR.v = 1 and one of the following is true: • XAR.simd = 1 • XAR.urs1 ≠ 0 • XAR.urs2 ≠ 0 • XAR.urs3<2> ≠ 0 • XAR.urs3<1:0> = 10_2 • XAR.urd<1> ≠ 0

7.155. Montgomery Multiplication

Opcode	opf	Operation	HPC-ACE		Assembly Language Syntax
			Regs	SIMD	
FMONTMUL[XII]	0 1000 1110_2	Montgomery Multiplication	iii	iv	fmontmul *freg*$_{rs1}$, *length*, *freg*$_{rd}$
FMONTSQR[XII]	0 1000 1110_2	Montgomery Multiplication (squared)	iii	iv	fmontmul *freg*$_{rs1}$, *length*, *freg*$_{rd}$



Description

FMONTMUL and FMONTSQR performs a calculation shown below (as pseudo-code) with length equal to "length + 1" (the value of "length" is specified in the instruction field). The data size for data A, B, and N used in the calculation is "length × 64" bits. The maximum length is 32. If the length of FMONTMUL and FMONTSQR is less than 32, the remaining operand locations are not used and the remaining result locations are unchanged. The combinations of basic and extended double precision floating-point registers are used as input data and output data.

For FMONTMUL and FMONTSQR, data A, B, N, and N' are used as input data and data A is overwritten as the output data. For FMONTSQR, data B is the same as data A. Refer to Figure 7-21 and Figure 7-24.

The first number of registers used for data A, B, N, and N' are fixed. The rd field is used for specfy the first number of registers for data A. The value of 0x00 (that means "%f0") must be specified to this field. In addition the rs1 field is used for specify the first number of registers for data B. The value of 0x01 (that means "%f32") must be specified to this field for FMONTMUL and the value of 0x00 (that means "%f0") for FMONTSQR. The first number of registers for N and N' is fixed to "%f64" and "%f352" respectively.

The pseudo-code of FMONTMUL and FMONTSQR is shown below.

```
/* Pseudo-code of FMONTMUL and             */
/* Pseudo-code of FMONTSQR (in case A = B) */

input  : A, B, N, n'0
output : Y

/* multi-precision integers                                    */
/* Each element of A, B, N, X, Y, and N' is 64-bit integer.    */
/* C and tmp : 64-bit integer                                  */

A = (ak-1, … , a0)
B = (bk-1, … , b0)
N = (nk-1, … , n0)
X = (xk+1, … , x0)
Y = (Yk, …, Y0)
N' = (n'k-1, … , n'0)

/* Algorithm */
```

[iii] The registers cannot be extended, but XAR.v must be set to 1 in order to execute these instructions.
[iv] XAR.simd must be set to 1 in order to execute these instructions.

```
r = 2^64
Y = (0, 0, … , 0)

for j = 0 to k-1

   C = 0
   for i = 0 to k-1
      (C, x_i) = y_i + C + a_i × b_j  ! A × b_j
   next i
   (x_k+1, x_k) = C + y_k

   m = x_0 × n'_0 (mod r)
   (C, tmp) = x_0 + n_0 × m  ! tmp is not used

   for i = 1 to k-1
      (C, y_i-1) = x_i + C + n_i × m  ! N × m
   next i
   (C, y_k-1) = C + x_k
   y_k = C + x_k+1

next j

if Y >= N then Y = Y – N
return Y
```

If A, B, N, and N' satisfy the all of the following conditions, the result of FMONTMUL and FMONTSQR is the same as the result of the Montgomery multiplication.

・N is an odd number.

・NN' mod R = −1 (R = 2^k > N, k = 64 × length)

・A < N, B < N

The Montgomery multiplication is calculated with the following formula.

Y = A ⊗ B = A × B × R^{-1} mod N (⊗ : Montogomery Multiplication)

(A, B: input, R = 2^k > N, k = 64 × length, RR^{-1} = 1 mod N)

Programming Note RSA encryption requires to use a lot of multiplications and modular arithmetics (A^D mod N), and it requires a lot of clock cycles. The calculations can be accelerated using FMONTMUL and FMONTSQR.

・FMONTMUL

The input data A, B, N, and N' are stated in Table 7-25.

Table 7-25 Data A, B, N, and N' for FMONTMUL (the length is 32)

Data	Corresponding Registers
A	%f30::%f286::%f28::%f284:: … ::%f0::%f256
B	%f62::%f318::%f60::%f316:: … ::%f32::%f288

N	%f94::%f350::%f92::%f348:: … ::%f64::%f320
N’	%f352

The registers corresponding to data A, B, N, and N’ must be set before FMONTMUL is executed. The result of FMONTMUL is overwritten to the registers corresponding to data A.

The register assignments used for FMONTMUL are shown in Figure 7-21.

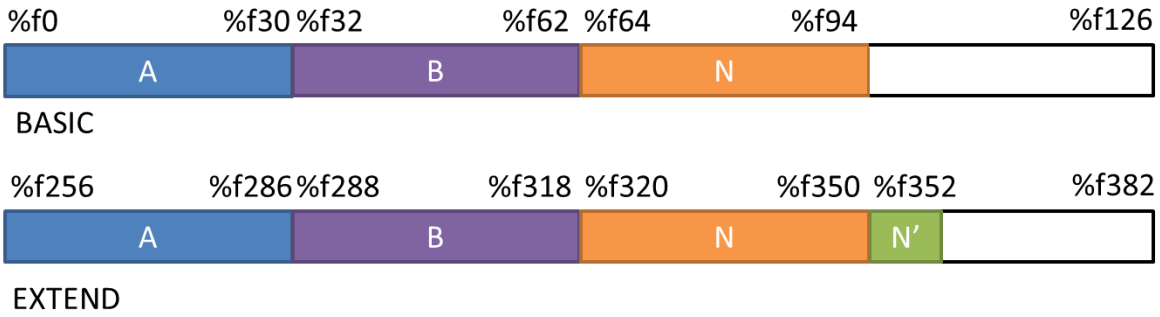


Figure 7-21 Register Assignments for FMONTMUL

Data A, B, and N are composed of the corresponding basic and extended registers. For example, the register corresponding to data A is shown in Figure 7-22.

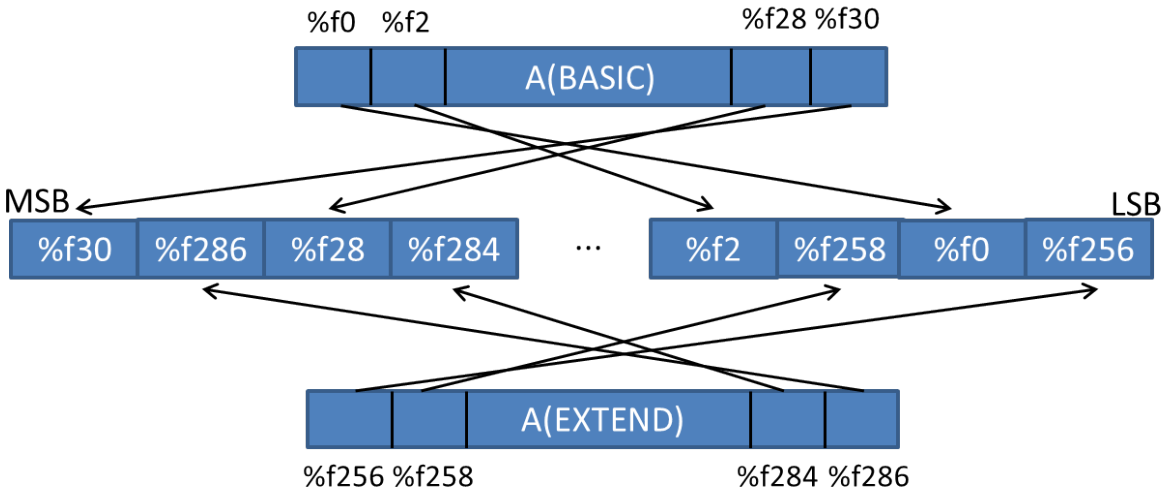


Figure 7-22 Combinations of basic and extended registers for data A

The number of registers used for FMONTMUL (as data A, B, and N) is specified by the length. Therefore each number of registers in use can be specified with a range of 1 – 32.

For example, if the length is 30 (that means the field of length is set to “0x1D”), 30 registers from LSB are used for data A, B, and N but not the two registers from MSB as shown in Figure 7-23. In addition, 30 registers from LSB that correspond to data A are updated as a result of the calculation, but not the two registers from MSB (%f30, %f286).

Figure 7-23 Example of registers specified for `FMONTMUL`

If A, B, N, and N' satisfy all the conditions stated in Table 7-26, the result of `FMONTMUL` is the same as the value calculated by the following formula (Montgomery multiplication).

$A \times B \times R^{-1} \bmod N$ ($NN' \bmod R = -1$)

Table 7-26 The conditions for the Montgomery Multiplication

data	The condition
A	A is a number that satisfies "A < N".
B	B is a number that satisfies "B < N".
N	N is an odd number.
N'	N' is a number that satisfies "NN' mod R= −1". (R is a number that satisfies "$R = 2^k > N$, $k = 64 \times$ length".) (Only the lower 64 bits of N' is used for calculations)

- **`FMONTSQR`**

The input data A, N, and N' are stated in Table 7-27.

Table 7-27 Data A, N, and N' for `FMONTSQR` (the length is 32)

Data	Corresponding Registers
A	%f30::%f286::%f28::%f284:: … ::%f0::%f256

N	%f94::%f350::%f92::%f348:: … ::%f64::%f320
N'	%f352

The registers corresponding to data A, N, and N' must be set before FMONTSQR is executed. The result of FMONTSQR is overwritten to the registers corresponding to data A.

The register assignments used for FMONTSQR are shown in Figure 7-24.

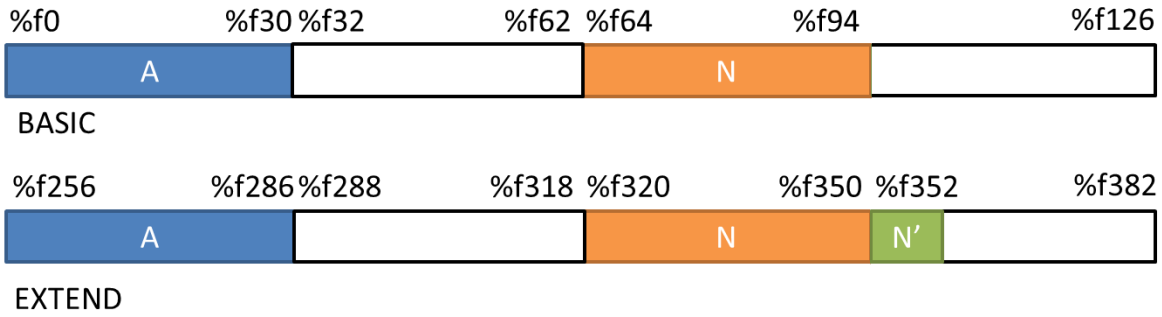


Figure 7-24 Register Assignments for FMONTSQR

Data A and N are composed of the corresponding basic and extended registers.

The number of registers used for FMONTSQR (as data A and N) is specified by the length. Therefore each number of registers in use can be specified with a range of 1 – 32.

For example, if the length is 30 (that means the field of length is set to "0x1D"), 30 registers from LSB are used for data A and N but not the two registers from MSB as shown in Figure 7-25. In addition, 30 registers from LSB that correspond to data A are updated as a result of the calculation, but not the two registers from MSB (%f30, %f286).

Figure 7-25 Example of registers specified for `FMONTSQR`

If A, N, and N' satisfy all the conditions stated in Table 7-28, the result of `FMONTSQR` is the same as the value calculated by the following formula (Montgomery multiplication).

$A \times A \times R^{-1} \bmod N$ ($NN' \bmod R = -1$)

Table 7-28 The conditions for the Montgomery Multiplication

data	the condition
A	A is a number that satisfies "$A < N$".
N	N is an odd number.
N'	N' is a number that satisfies "$NN' \bmod R = -1$". (R is a number that satisfies "$R = 2^k > N$, $k = 64 \times$ length".) (Only the lower 64 bits of N' is used for calculations)

・**Pseudo-code example used with FMONTMUL and FMONTSQR**

A pseudo-code example used with FMONTMUL and FMONTSQR in multiplication and modular arithmetic (A^D mod N) for RSA encryption is shown below.

```
/* (A^D mod N) for RSA encryption */
/* MONTMUL: (OP1) × (OP2) × R^-1 mod N  → OP1 (overwritten) */
/* MONTSQR: (OP1) × (OP1) × R^-1 mod N  → OP1 (overwritten) */
/* D = (1, d_k-2, … , d_1, d_0): binary notation (k bit)  */
/* ⊗ : Montgomery Multiplication */

load A to OP1                  ! OP1 = A
load (R^2 mod N) to OP2        ! OP2 = R^2 mod N
load N to OP3                  ! OP3 = N
load n'0 to OP4                ! OP4 = n'0
fmontmul                       ! A × R^2 × R^-1 mod N
                               ! = A × R mod N = FR(A)
copy OP1 to OP2                ! OP1, OP2 = FR(A)

for (i = k-2; i >= 0; i --) {  ! OP1 = FR(X)
    fmontsqr                   ! FR(X) ⊗ FR(X) = FR(X^2) → OP1
    if (d_i == 1) {
        fmontmul               ! FR(X^2) ⊗ FR(A) = FR(X^2 × A) → OP1
    }
}
                               ! OP1 = FR(A^D) (temporary result)
load 1 to OP2                  ! OP2 = 1
fmontmul                       ! FR(A^D) × 1 × R^-1 mod N = A^D mod N
                               ! (This result is written to OP1.)
```

Exception	Target instruction	Detection condition
fp_disabled	All	PSTATE.pef = 0 or FPRS.fef = 0
illegal_instruction	All	One of the following is true: • rs1 ≠ 0 and rs1 ≠ 32 • rd ≠ 0
illegal_action	All	• XAR.v = 0 • If XAR.v = 1 and one of the following is true: ・XAR.simd = 0 ・XAR.urs1 ≠ 0 ・XAR.urs2 ≠ 0 ・XAR.urs3 ≠ 0 ・XAR.urd ≠ 0

8. IEEE Std. 754-1985 Requirements for SPARC-V9

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™ XII, the behavior of all floating-point instructions is changed according to the value of FSR.ns.

9. Memory Models

Refer to the SPARC64 X/X+ specification.

10. Address Space Identifiers

10.3. ASI Assignment

10.3.1. Supported ASIs

ASIs supported in SPARC64™ XII 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 the MMU settings.
	Real	The address is treated as a real address (RA).
	non-T	Not translated by the MMU. VA watchpoint is not detected.
Sharing(non-T only)	Chip	The register is shared by the entire CPU.
	Core	The 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}	—	—	—	—	—	

ASI	VA	ASI name	Access	Type	Sharing	Page
$C0_{16}$	—	ASI_PST8_PRIMARY (ASI_PST8_P)	WO	Trans.	—	
$C1_{16}$	—	ASI_PST8_SECONDARY (ASI_PST8_S)	WO	Trans.	—	
$C2_{16}$	—	ASI_PST16_PRIMARY (ASI_PST16_P)	WO	Trans.	—	
$C3_{16}$	—	ASI_PST16_SECONDARY (ASI_PST16_S)	WO	Trans.	—	
$C4_{16}$	—	ASI_PST32_PRIMARY (ASI_PST32_P)	WO	Trans.	—	
$C5_{16}$	—	ASI_PST32_SECONDARY (ASI_PST32_S)	WO	Trans.	—	
$C6_{16}$ – $C7_{16}$	—	—	—	—	—	
$C8_{16}$	—	ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)	WO	Trans.	—	
$C9_{16}$	—	ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)	WO	Trans.	—	
CA_{16}	—	ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)	WO	Trans.	—	
CB_{16}	—	ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)	WO	Trans.	—	
CC_{16}	—	ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)	WO	Trans.	—	
CD_{16}	—	ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)	WO	Trans.	—	
CE_{16} – CF_{16}	—	—	—	—	—	
$D0_{16}$	—	ASI_FL8_PRIMARY (ASI_FL8_P)	RW	Trans.	—	
$D1_{16}$	—	ASI_FL8_SECONDARY (ASI_FL8_S)	RW	Trans.	—	
$D2_{16}$	—	ASI_FL16_PRIMARY (ASI_FL16_P)	RW	Trans.	—	
$D3_{16}$	—	ASI_FL16_SECONDARY (ASI_FL16_S)	RW	Trans.	—	
$D4_{16}$ – $D7_{16}$	—	—	—	—	—	
$D8_{16}$	—	ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL)	RW	Trans.	—	
$D9_{16}$	—	ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)	RW	Trans.	—	
DA_{16}	—	ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)	RW	Trans.	—	
DB_{16}	—	ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)	RW	Trans.	—	
DC_{16} – DF_{16}	—	—	—	—	—	
$E0_{16}$	—	ASI_BLOCK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)	WO	Trans.	—	
$E1_{16}$	—	ASI_BLOCK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)	WO	Trans.	—	
$E2_{16}$	—	ASI_TWINX_P/ASI_STBI_P	RW	Trans.	—	
$E3_{16}$	—	ASI_TWINX_S/ASI_STBI_S	RW	Trans.	—	
$E6_{16}$	—	—	—	—	—	
$E7_{16}$	210_{16}	ASI_RANDOM_NUMBER	RO	non-T	Chip	113
EA_{16}	—	ASI_TWINX_PL/ASI_STBI_PL	RW	Trans.	—	

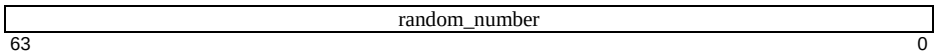
ASI	VA	ASI name	Access	Type	Sharing	Page
EB_{16}	—	ASI_TWINX_SL/ASI_STBI_SL	RW	Trans.	—	
EC_{16} – EF_{16}	—	—	—	—	—	
$F0_{16}$	—	ASI_BLOCK_PRIMARY (ASI_BLK_P)	RW	Trans.	—	
$F1_{16}$	—	ASI_BLOCK_SECONDARY (ASI_BLK_S)	RW	Trans.	—	
$F2_{16}$	any	ASI_STBI_MRU_P	WO	Trans.	—	
$F3_{16}$	any	ASI_STBI_MRU_S	WO	Trans.	—	
$F4_{16}$	—	ASI_XFILL_P	WO	Trans.	—	
$F5_{16}$	—	ASI_XFILL_S	WO	Trans.	—	
$F6_{16}$ – $F7_{16}$	—	—	—	—	—	
$F8_{16}$	—	ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)	RW	Trans.	—	
$F9_{16}$	—	ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)	RW	Trans.	—	
FA_{16}	any	ASI_STBI_MRU_P_LITTLE	WO	Trans.	—	
FB_{16}	any	ASI_STBI_MRU_S_LITTLE	WO	Trans.	—	
FC_{16} – FF_{16}	—	—	—	—	—	

10.5. ASI-Accessible Registers

10.5.5. ASI_RANDOM_NUMBER

Register name ASI_RANDOM_NUMBER
ASI number $E7_{16}$
VA 210_{16}
Range of sharing Chip
Access

	read	write
user	OK	*DAE_invalid_asi*



Bit	Field	Access	Description
63:0	random_number	RO	The value (64-bit) generated by Onchip Random Number Generator

The value (64-bit) generated by Onchip Random Number Generator can be read from the random_number field in ASI_RANDOM_NUMBER. LDXA, LDDFA, and LDTWA can be used to access this ASI (LDTWA is deprecated).

When the value read from ASI_RANDOM_NUMBER is valid, XASR.rng_stat is set to 1. If invalid, XASR.rng_stat is set to 0. If XAR.rng_stat is 0, the value of R[rd] or F[rd] is updated by an undefined value and must not be used.

The factors for the invalid value are stated below.

a) the temporary read value does not have sufficient precision

b) read failure based on the continuous hardware error

In case of a), the value can become valid with a retry, but in case of b), the value will remain invalid even with a retry. Therefore a retry process and a retry timeout process (after several retry processes) must be implemented in the software.

```
/* pseudo asm code */
init_rnd_num:
  wr    0xe7, %asi
  orcc  %g0, 0x8, %l7 /* set retry counter */
rd_rnd_num:
  bne    rd_rnd_fail /* retry out */
  nop
  ldda   [%g0+0x210] %asi, %f0
  membar #Sync
  rd     %xasr, %l0
  srlx   %l0, 40, %l0
  xorcc  %l0, 1, %l0
  bne,a   rd_rnd_num
  subcc  %l7,1,%l7
```

11. Performance Instrumentation

11.1 Overview

Performance counters are comprised of one “Performance Control Register (PCR) (ASR 16)” and multiple instances of “Performance Instrumentation Counter Register (PIC) (ASR 17)”.

SPARC64™ XII implements 4 PIC registers, which are selected by PCR.SC, and are accessed via ASR 17. Each PIC register contains two counters.

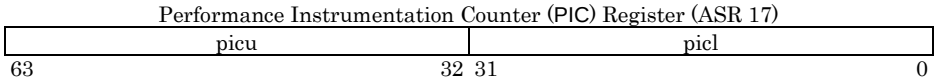
Performance Control Register (PCR) (ASR 16)

—	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 the 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. 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. A read of the PCR.ovro field returns 0. The PCR.ovro field allows PCR to be updated without changing the overflow status. The hardware maintains the most recent state of PCR.ovf so 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. 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 the PIC counter pair that 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	RW	Hyperprivileged mode. If PCR.ht = 1, events that occur while in hyperprivileged mode are counted. If PCR.ut, PCR.st, and PCR.ht are all 1, all events are counted. If PCR.ut, PCR.st, and PCR.ht are all 0, counting is disabled. PCR.ht is a global field and applies to all PICs.
2	ut	RW	User mode. If PCR.ut = 1, events that occur while in non-provileged mode are counted. If PCR.ut, PCR.st, and PCR.ht are all 1, all events are counted. If PCR.ut, PCR.st, and PCR.ht are all 0, counting is disabled. PCR.ut is a global field and applies to all PICs.
1	st	RW	System mode. If PCR.st = 1, events that occur while in privileged mode are counted. If PCR.ut, PCR.st, and PCR.ht are all 1, all events are counted. If PCR.ut, PCR.st, and PCR.ht are all 0, counting is disabled. PCR.st is a global field and applies to all PICs.
0	priv	RW	Privileged. If PCR.priv = 1, executing an `RDPCR`, `WRPCR`, `RDPIC`, or `WRPIC` instruction in non-privileged mode causes a *privileged_action* exception. If PCR.priv = 0, an attempt to update PCR.priv (writing a value of 1) in non-privileged mode via a `WRPCR` instruction causes a *privileged_action* exception. PCR.priv is a global field and applies to all PICs.

Performance Instrumentation Counter (PIC) Register (ASR 17)



Bits	Field	Access	Description
63:32	picu	RW	32bits counter selected by PCR.su for the event
31:0	picl	RW	32bits counter selected by PCR.sl for the event

11.1.1 Pseudo-code Examples

11.1.1.1 Counter Clear/Set

The counter fields in the PIC registers are read/write fields. Writing zero clears a counter and writing any other value sets the counter to that value. The following pseudo-code clears all PIC registers (privileged access is assumed).

```
/* Clear PICs without updating SL/SU 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 PIC[i]        */
}
```

11.1.1.2 Counter Event Selection and Start

Counter events are selected using the PCR.sc and PCR.su/PCR.sl fields. The following pseudo-code selects events and enables the counters (privileged access is assumed).

```
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 Stop Counter and Read

The following pseudo-code disables the counters and reads the value (privileged access is assumed).

```
pcr.ut = 0x0;       /* Disable user counts        */
pcr.st = 0x0;       /* Disable system counts, too */
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. LL cache events

5. Bus transaction events

There are 2 types of PA events, standard and supplemental, that can be measured in SPARC64™ XII.

Standard events in SPARC64™ XII 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 be changed without notice. Compatibility with future processors is not guaranteed.

Table 11-1 shows the PA events defined in SPARC64™ XII.

Shaded events are supplemental events.

For details on each event, refer to the descriptions in the following sections. Unless otherwise indicated, speculative instructions are also counted by the PA events.

Table 11-1 PA Events and Encodings

Encoding (bin)	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	*only_this_ thread_active*	*single_mode_ cycle_counts*	*single_mode_ instruction_counts*	instruction_ flow_counts	*d_move_wait*	*cse_priority_wait*	xma_inst
0000_0011	iwr_empty	*w_cse_window_ empty*	*w_eu_comp_wait*	*w_branch_comp _wait*	iwr_empty	*w_op_stv_wait*	*w_d_move*	*w_0endop*
0000_0100	*Reserved*	*w_op_stv_wait_ nc_pend*	*w_op_stv_ wait_ll_miss*	*w_op_stv_wait_ ll_miss_ex*	*Reserved*	*w_fl_comp_wait*	*w_cse_window_ empty_sp_full*	*w_op_stv_ wait_ex*
0000_0101	op_stv_wait							
0000_0110	effective_instruction_counts							
0000_0111	SIMD_load_sto re_instructions	SIMD_floating_ instructions	SIMD_fma_ instructions	sxar1_ instructions	sxar2_ instructions	unpack_sxar1	unpack_sxar2	*Reserved*
0000_1000	load_store_instructions							
0000_1001	branch_instructions							
0000_1010	floating_instructions							
0000_1011	fma_instructions							
0000_1100	prefetch_instructions							
0000_1101	fixed_point_ins tructions	ex_load_ instructions	ex_store_ instructions	fl_load_ instructions	fl_store_ instructions	SIMD_fl_load_ instructions	SIMD_fl_store_ instructions	SIMD_fixed_point_instr uctions
0000_1110	op_stv_wait_l2 _miss	op_stv_wait_l2_ miss_ex	w_op_stv_wait_l2 _miss	w_op_stv_wait_l2_mi ss_ex	op_stv_wait_l1d_ miss	op_stv_wait_l1d_m iss_ex	w_op_stv_wait_l1d _miss	w_op_stv_wait_l1d_mis s_ex
0000_1111	x_move_instruc tions	w_op_stv_wait_p fp_busy	w_op_stv_wait_pf p_busy_ex	w_op_stv_wait_pfp_b usy_swpf	load_DSP_instruc tions	SIMD_load_DSP_i nstructions	store_DSP_instruci tons	SIMD_store_DSP_instr uctions
0001_0000	*Reserved*							
0001_0001	*Reserved*							
0001_0010	rs1	flush_rs	*Reserved*					
0001_0011	1iid_use	2iid_use	3iid_use	4iid_use	*Reserved*	sync_intlk	regwin_intlk	*Reserved*
0001_0100	*Reserved*							
0001_0101	*Reserved*	toq_rsbr_phanto m	*Reserved*	flush_rs	*Reserved*		rs1	*Reserved*
0001_0110	trap_all		trap_int_level	trap_spill	trap_fill	trap_trap_inst		
0001_0111	*Reserved*		*Reserved*				other_thread_com mit	w_strand_id_not_empty
0001_1000	*only_this_ thread _active*	*both_ threads _active*	*both_ threads _empty*	*Reserved*			*op_stv_wait_ pfp_busy_swpf*	*op_stv_ wait_ll_miss*
0001_1001	*Reserved*							
0001_1010	*Reserved*		*single_sxar_comm it*	*Reserved*				*suspend_cycle*
0001_1011	rsf_pmmi	*Reserved*	op_stv_wait_ nc_pend	0iid_use	flush_rs	*Reserved*		decode_all_intlk

0001_1100	*Reserved*							
0001_1101	*op_stv_wait_ pfp_busy_ex*	*Reserved*	op_stv_wait_ ll_miss_ex	op_stv_wait_ nc_pend	cse_window_ empty_sp_full	op_stv_wait_ pfp_busy	*both_ threads _ suspended*	*Reserved*
0001_1110	cse_window_ empty	eu_comp_wait	branch_comp_ wait	0endop	op_stv_wait_ex	fl_comp_wait	1endop	2endop
0001_1111	single_uop_com mit	*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_a ll	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*		LL_miss_dm	LL_miss_pf	LL_read_dm	LL_read_pf	LL_wb_dm	LL_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	LL_miss_wait_ dm_bank0	LL_miss_wait_ pf_bank0	LL_miss_counts_ dm_bank0	LL_miss_counts_ pf_bank0	LL_miss_wait_ dm_bank1	LL_miss_wait_ pf_bank1	LL_miss_counts_ dm_bank1	LL_miss_counts_ pf_bank1
0011_0011	LL_miss_counts_ dm_bank2	LL_miss_counts_ pf_bank2	LL_miss_wait_ dm_bank2	LL_miss_wait_ pf_bank2	LL_miss_counts_ dm_bank3	LL_miss_counts_ pf_bank3	LL_miss_wait_ dm_bank3	LL_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								
0011_1110	*Reserved*							

0011_1111	*Reserved*							
0101_0000	l2_sy_miss_dm	l2_sy_read_dm	*Reserved*	l2_wb_dm	*Reserved*	l2_sy_miss_wait_dm_part1	*Reserved*	l2_sy_miss_wait_dm_part2
0101_0001	*Reserved*				l2_bi_counts	l2_cpi_counts	l2_cpb_counts	l2_cpd_counts
0101_0010	*Reserved*							
0101_0011	*Reserved*							
0101_0100	*Reserved*							
0101_0101	*Reserved*							
0101_0110	*Reserved*							
0101_0111	*Reserved*							
0101_1000	*Reserved*							
0101_1001	*Reserved*							
0101_1010	*Reserved*							
0101_1011	*Reserved*							
0101_1100	*Reserved*							
0101_1101	*Reserved*							
0101_1110	*Reserved*							
0101_1111	*Reserved*							
1111_1111	*Disabled(No PIC is counted up)*							

※Encodings not shown are Reserved.

11.2.1 Instruction and Trap Statistics

Standard PA Events

1 *cycle_counts*

Counts the number of cycles when the performance counter is enabled. Based on the settings of PCR.ut and PCR.st, this counter which is similar to the TICK register can count user cycles and system cycles separately.

2 *instruction_counts* (Non-Speculative)

Counts the number of committed instructions, including SXAR1 and SXAR2. SPARC64™ XII commits up to 4 non-SXAR instructions per cycle and up to 2 SXAR instructions. Thus, *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 with *cycle_counts*.

IPC = *effective_instruction_counts* / *cycle_counts*

If *effective_Instruction_counts* and *cycle_counts* are collected for the user or the system modes, the IPC can be calculated in either user or system mode.

4 *load_store_instructions* (Non-Speculative)

Counts the number of committed non-SIMD load/store instructions. Also counts the number of atomic load-store instructions.

5 *branch_instructions* (Non-Speculative)

Counts the number of committed branch instructions. Also counts the number of 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> = $0A_{16}$, $0B_{16}$, 16_{16}, or 17_{16}.

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 and 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 and 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 and 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 number of all trap event occurrences. The number of counted occurrences equals the sum of the occurrences that are counted by all trap PA events.

16 *trap_int_level* (Non-Speculative)

Counts the number of *interrupt_level_n* occurrences.

17 *trap_spill* (Non-Speculative)

Counts the number of *spill_n_normal* and *spill_n_other* occurrences.

18 *trap_fill* (Non-Speculative)

Counts the number of *fill_n_normal* and *fill_n_other* occurrences.

19 *trap_trap_inst* (Non-Speculative)

Counts the number of *trap_instruction* occurrences.

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. In SPARC64™ XII, some instructions are processed internally as several separate instructions and are called as instruction flows. This event does not count packed `SXAR1` and `SXAR2` instructions.

27 *single_uop_commit* (Non-Speculative)

Counts the number of committed instruction flows except for the last flow.

28 *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.

29 *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.

30 *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}`.

31 *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}.

32 *SIMD_fl_load_instructions* (Non-Speculative)

Counts the number of committed SIMD floating-point load instructions. Counts the LDF{A} and LDDF{A} instructions.

33 *SIMD_fl_store_instructions* (Non-Speculative)

Counts the number of committed SIMD floating-point store instructions. Counts the STF{A}, STDF{A}, STFR, and STDFR instructions.

34 *x_move_instructions* (Non-Speculative)

Counts the number of commited move instructions. Counts the MOVdTOx, MOVsTOuw, MOVfwTOuw, MOVsTOsw, MOVfwTOsw, MOVxTOd, MOVwTOs, MOVwTOfuw, and MOVwTOfsw instructions.

35 *fixed_point_instructions* (Non-Speculative)

Counts the number of commited integer instructions. Counts the FSLL32, FSRL32, FSRA32, FPSLL64x, FPSRL64x, FPSRA64x, FPADD{8|64}, FPSUB{8|64}, FPMUL64, FPMUL32, FPADD128XHI, FPADD{16|32}{|S}, FPSUB{16|32}{|S}, FZERO{|S}, FNOR{|S}, FANDNOT{1|2}{|S}, FNOT{1|2}{|S}, FXOR{|S}, FNAND{|S}, FAND{|S}, FXNOR{|S}, FSRC{1|2}{|S}, FORNOT{1|2}{|S}, FOR{|S}, FONE{|S}, FPMADDX, and FPMADDXHI instructions.

36 *SIMD_fixed_point_instructions* (Non-Speculative)

Counts the number of commited SIMD integer instructions. Counts the SIMD version of the FSLL32, FSRL32, FSRA32, FPSLL64x, FPSRL64x, FPSRA64x, FPADD{8|64}, FPSUB{8|64}, FPMUL64, FPMUL32, FPADD128XHI, FPADD{16|32}{|S}, FPSUB{16|32}{|S}, FZERO{|S}, FNOR{|S}, FANDNOT{1|2}{|S}, FNOT{1|2}{|S}, FXOR{|S}, FNAND{|S}, FAND{|S}, FXNOR{|S}, FSRC{1|2}{|S}, FORNOT{1|2}{|S}, FOR{|S}, FONE{|S}, FPMADDX, and FPMADDXHI instructions.

37 *load_DSP_instructions* (Non-Speculative)

Counts the number of commited load_DSP instructions. Counts the LDDFDS instructions.

38 *store_DSP_instructions* (Non-Speculative)

Counts the number of commited store_DSP instructions. Counts the STDFDS, STDFRDS, and STDFRDW instructions.

39 *SIMD_load_DSP_instructions* (Non-Speculative)

Counts the number of commited SIMD load_DSP instructions. Counts the SIMD version of the `LDDFDS` instructions.

40 *SIMD_store_DSP_instructions* (Non-Speculative)

Counts the number of commited SIMD store_DSP instructions. Counts the SIMD version of the `STDFDS`, `STDFRDS`, and `STDFRDW` instructions.

41 *iwr_empty*

Counts the number of cycles when the Issue Word Register (IWR) is empty. The IWR is a four entry register that holds instructions during a decoding and the IWR may be empty if an instruction cache miss prevents an instruction fetch.

42 *rs1* (Non-Speculative)

Counts the number of cycles in which a normal execution is halted due to one of the following:

- a trap or interrupt
- update of privileged registers
- guarantee of memory ordering
- RAS-initiated hardware retry

43 *flush_rs* (Non-Speculative)

Counts the number of pipeline flushes due to a branch misprediction. Since SPARC64™ XII supports 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 = *flush_rs* / *branch_instructions*

44 *0iid_use*

Counts the number of cycles when no instruction is issued. SPARC64™ XII issues up to four non-SXAR instructions per cycle. When no instruction is issued, *0iid_use* is incremented. In SPARC64™ XII, some instructions are processed internally as several separate instructions and are called as instruction flows. Each of these instruction flows is counted. `SXAR` instructions are also counted.

45 *1iid_use*

Counts the number of cycles when one instruction is issued.

46 *2iid_use*

Counts the number of cycles when two instructions are issued.

47 *3iid_use*

Counts the number of cycles when three instructions are issued.

48 *4iid_use*

Counts the number of cycles when four instructions are issued.

49 *sync_intlk*

Counts the number of cycles when the instructions that are issued are blocked by a pipeline sync.

50 *regwin_intlk*

Counts the number of cycles when the instructions that are issued are blocked by a register window switch.

51 *decode_all_intlk*

Counts the number of cycles when the instructions that are issued are blocked by a static interlock condition during 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.

52 *rsf_pmmi* (Non-Speculative)

Counts the number of cycles when mixing single-precision and double-precision floating-point operations prevents instructions from being issued.

53 *toq_rsbr_phantom*

Counts the number of instructions that are not branch instructions but are predicted as branch instructions to be taken. Branch prediction in SPARC64™ XII is done prior to instruction decode. In other words, branch prediction occurs regardless of whether the instruction is actually a branch instruction. Instructions that are not branch instructions may be incorrectly predicted as branch instructions to be taken.

54 *op_stv_wait* (Non-Speculative)

Counts the number of cycles when no instructions are committed because the oldest, uncommitted instruction is a memory access waiting for data. *op_stv_wait* does not count cycles when 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 counted.

55 *op_stv_wait_nc_pend* (Non-Speculative)

Counts the number of *op_stv_wait* for noncacheable accesses.

56 *op_stv_wait_ex* (Non-Speculative)

Counts the number of *op_stv_wait* for integer memory access instructions. Does not distinguish between L1 cache and L2 cache misses.

57 *op_stv_wait_ll_miss* (Non-Speculative)

Counts the number of *op_stv_wait* caused by a Last Level cache (LL cache) miss. Does not distinguish between integer and floating-point loads.

58 *op_stv_wait_ll_miss_ex* (Non-Speculative)

Counts the number of *op_stv_wait* caused by an integer-load Last Level cache (LL cache) miss.

59 *op_stv_wait_pfp_busy* (Non-Speculative)

Counts the number of *op_stv_wait* caused by a memory access instruction that cannot be executed due to the lack of an available prefetch port.

60 *op_stv_wait_pfp_busy_ex* (Non-Speculative)

Counts the number of *op_stv_wait* caused by an integer memory access instruction that cannot be executed due to the lack of an available prefetch port.

61 *op_stv_wait_swpf* (Non-Speculative)

Counts the number of *op_stv_wait* caused by a prefetch instruction.

62 *op_stv_wait_pfp_busy_swpf* (Non-Speculative)

Counts the number of op_stv_wait caused by a prefetch instruction that cannot be executed due to the lack of an available prefetch port.

63 *op_stv_wait_l2_miss* (Non-Speculative)

Counts the number of *op_stv_wait* caused by an L2 cache miss. Does not distinguish between integer and floating-point loads.

64 *op_stv_wait_l2_miss_ex* (Non-Speculative)

Counts the number of *op_stv_wait* caused by an integer-load L2 cache miss.

65 *op_stv_wait_l1d_miss* (Non-Speculative)

Counts the number of *op_stv_wait* caused by an L1D cache miss. Does not distinguish between integer and floating-point loads.

66 *op_stv_wait_l1d_miss_ex* (Non-Speculative)

Counts the number of *op_stv_wait* caused by an integer-load L1D cache miss.

67 *cse_window_empty_sp_full* (Non-Speculative)

Counts the number of cycles when no instructions are committed because the CSE is empty and the store ports are full.

68 *cse_window_empty* (Non-Speculative)

Counts the number of cycles when no instructions are committed because the CSE is empty.

69 *branch_comp_wait* (Non-Speculative)

Counts the number of cycles when 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*.

70 *eu_comp_wait* (Non-Speculative)

Counts the number of cycles when 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*.

71 *fl_comp_wait* (Non-Speculative)

Counts the number of cycles when no instructions are committed and the oldest, uncommitted instruction is a floating-point instruction.

72 *0endop* (Non-Speculative)

Counts the number of cycles when no instructions are committed. *0endop* also counts cycles where the only instruction committed is an SXAR instruction.

73 *1endop* (Non-Speculative)

Counts the number of cycles when one instruction is committed.

74 *2endop* (Non-Speculative)

Counts the number of cycles when two instructions are committed.

75 *3endop* (Non-Speculative)

Counts the number of cycles when three instructions are committed.

77 *sleep_cycle* (Non-Speculative)

Counts the number of cycles when the instruction unit is halted by a SLEEP instruction.

78 *single_sxar_commit* (Non-Speculative)

Counts the number of cycles when the only instruction committed is an unpacked SXAR instruction. These cycles are also counted by *0endop*.

79 *d_move_wait* (non-speculative)

Counts the number of cycles when no instructions are committed while waiting for the register window to be updated.

80 *cse_priority_wait*

Counts the number of cycles when no instructions are committed because the SMT thread is waiting for the commit priority. In SPARC64™ XII, 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 the thread.

81 *w_cse_window_empty* (non-speculative)

Counts the number of cycles when *cse_window_empty* is observed for the thread that has the commit priority.

82 *w_eu_comp_wait* (non-speculative)

Counts the number of cycles when *eu_comp_wait* is observed for the thread that has the commit priority.

83 *w_branch_comp_wait* (non-speculative)

Counts the number of cycles when *branch_comp_wait* is observed for the thread that has the commit priority.

84 *w_op_stv_wait* (non-speculative)

Counts the number of cycles when *op_stv_wait* is observed for the thread that has the commit priority.

85 *w_d_move_wait*

Counts the number of cycles when *d_move_wait* is observed for the thread that has the commit priority.

86 *w_0endop* (non-speculative)

Counts the number of cycles when *0endop* is observed for the thread that has the commit priority.

87 *w_op_stv_wait_nc_pend* (non-speculative)

Counts the number of cycles when *op_stv_wait_nc_pend* is observed for the thread that has the commit priority.

88 *w_op_stv_wait_ll_miss* (non-speculative)

Counts the number of cycles when *op_stv_wait_ll_miss* is observed for the thread that has the commit priority.

89 *w_op_stv_wait_ll_miss_ex* (non-speculative)

Counts the number of cycles when *op_stv_wait_ll_miss_ex* is observed for the thread that has the commit priority.

90 *w_fl_comp_wait* (non-speculative)

Counts the number of cycles when *fl_comp_wait* is observed for the thread that has the commit priority.

91 *w_cse_window_empty_sp_full* (non-speculative)

Counts the number of cycles when *cse_window_empty_sp_full* is observed for the thread that has the commit priority.

92 *w_op_stv_wait_ex* (non-speculative)

Counts the number of cycles when *op_stv_wait_ex* is observed for the thread that has the commit priority.

93 *w_op_stv_wait_pfp_busy* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_pfp_busy* is observed for the thread that has the commit priority.

94 *w_op_stv_wait_pfp_busy_ex* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_pfp_busy_ex* is observed for the thread that has the commit priority.

95 *w_op_stv_wait_pfp_busy_swpf* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_pfp_busy_swpf* is observed for the thread that has the commit priority.

96 *w_op_stv_wait_l2_miss* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_l2_miss* is observed for the thread that has the commit priority.

97 *w_op_stv_wait_l2_miss_ex* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_l2_miss_ex* is observed for the thread that has the commit priority.

98 *w_op_stv_wait_l1d_miss* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_l1d_miss* is observed for the thread that has the commit priority.

99 *w_op_stv_wait_l1d_miss_ex* (Non-Speculative)

Counts the number of cycles when *op_stv_wait_l1d_miss_ex* is observed for the thread that has the commit priority.

100 *only_this_thread_active*

Counts the number of cycles when SMT is enabled, the CSE of this thread is not empty, and the CSEs of the other threads are empty.

101 *single_mode_cycle_counts*

Counts the number of cycles when the thread is active in the single-threaded mode (SMT disabled).

102 *single_mode_instructions*

Counts the number of committed instructions in the single-threaded mode (SMT disabled).

103 *both_threads_active*

Counts the number of cycles when SMT is enabled and the CSEs of all threads are active.

104 *both_threads_empty*

Counts the number of cycles when SMT is enabled and the CSEs of all threads are empty.

105 *both_threads_suspended*

Counts the number of cycles when all threads in a core are in the suspended state.

106 *other_thread_commit*

Counts the number of cycles when no instructions are committed because the instructions in the other threads are commited.

107 *w_strand_id_not_empty*

Counts the number of cycles when CSE is not empty for the thread that has the commit priority.

11.2.2 MMU and L1 cache Events

Standard PA Events

1 *uITLB_miss*

Counts the number of instruction uTLB misses.

2 *uDTLB_miss*

Counts the number of data uTLB misses.

3 *L1I_miss*

Counts the number of L1 instruction cache misses.

4 *L1D_miss*

Counts the number of L1 data cache misses.

5 *L1I_wait_all*

Counts the total time spent on processing L1 instruction cache misses (that is, the total miss latency). In SPARC64™ XII, the L1 cache is a non-blocking cache that can process multiple cache misses simultaneously. *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 on processing L1 data cache misses (that is, the total miss latency). In SPARC64™ XII, the L1 cache is a non-blocking cache that can process multiple cache misses simultaneously. *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 L1 cache and are sent to the LL cache .

10 *swpf_fail_all*

Counts the number of prefetch instructions that are lost in the L1 cache.

11 *swpf_lbs_hit*

Counts the number of prefetch instructions that hit in the L1 cache.

Prefetch instructions sent to the L1 cache

= *swpf_success_all* + *swpf_fail_all* + *swpf_lbs_hit*

12 *L1I_thrashing*

Counts the number of L2 read requests being issued twice during the period between acquiring and releasing a store port. When an instruction fetch causes an L1 instruction cache miss, the requested data is updated in L1I cache. This counter is incremented if the updated data is evicted before it can be read.

13 *L1D_thrashing*

Counts the number of L2 read requests being issued twice during 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 L1D cache. This counter is incremented if the updated data is evicted before it can be read.

14 L1D_miss_dm

Counts the number of L1 data cache misses for the load/store instructions.

15 L1D_miss_pf

Counts the number of L1 data cache misses for the prefetch instructions.

16 L1D_miss_qpf

Counts the number of L1 data cache misses for the hardware prefetch requests.

11.2.3 L2 cache Events

L2 cache events may be due to the actions of VCPUs, I/Os or external requests. Events caused by VCPUs are counted separately for each VCPU. Those caused by I/Os or external requests are counted for all VCPUs.

In the L2 cache, the demand (dm) events are counted, but the prefetch (pf) events are not checked. The prefetch (pf) events are counted in the LL cache. For more information about prefetch (pf) events, refer to 11.2.4.

Standard PA Events

1 *l2_sy_read_dm*

Counts the number of L2 cache references in the demand requests. References in the external requests are not counted.

2 *l2_sy_miss_dm*

Counts the number of L2 cache misses caused by demand requests.

3 *l2_sy_miss_wait_dm_part{1,2}*

Counts the total time spent on processing L2 cache misses caused by demand requests, that is, the total miss latency. The latency of each memory access request is counted. The total time is the sum of *L2_sy_miss_wait_dm_part{1,2}*.

4 *l2_wb_dm*

Counts the number of writebacks to the memory caused by L2 cache misses for the demand requests.

5 *l2_bi_counts*

Counts the number of external cache-invalidate requests. Cache-invalidate requests caused by IO-FST/PST requests are also counted as this event. These requests do not check the cache data before invalidating.

6 *l2_cpi_counts*

Counts the number of external cache-copy-and-invalidate requests received. These requests copy the updated cache data to memory before invalidating for inter CPU-chip copies. Cache data that is consistent with the memory does not need to be copied and is invalidated.

7 *l2_cpb_counts*

Counts the number of external cache-copyback requests received. These requests copy updated cache data to memory.

8 *l2_cpd_counts*

Counts the number of internal or external IO cache-read requests (DMA read requests).

11.2.4 LL cache Events

LL cache events may be due to the actions of VCPUs, I/Os or external requests. Events caused by VCPUs are counted separately for each VCPU. Those caused by I/Os or external requests are counted for all VCPUs.

Most LL cache events are categorized as either demand (dm) or prefetch (pf) events.

LL demand requests are basically due to an instruction fetch, a load/store instruction, or an L1 prefetch (by software and hardware) instruction that misses the L1 cache and the L2 cache.

LL prefetch requests are basically due to a LL prefetch (by software and hardware) that misses the L1 cache. LL prefetch requests are directly sent from the L1 cache to the LL cache without referencing the L2 cache.

Due to lack of CPU resources to access the L2 cache, however, an instruction fetch, a load/store instruction, and an L1 prefetch instruction (that misses the L1 cache and the L2 cache) can be processed as LL prefetch requests at first, and then processed as LL demand requests. In this case, these requests are double counted as LL prefetch requests and LL demand requests.

For example, when a load/store instruction cannot be executed due to lack of resources needed to move data into the L1 cache, the data is first moved into the LL cache by the prefetch request generated by hardware. Once the L1 cache resources become available, the load/store instruction is executed by the demand request.

Standard PA Events

1 *LL_read_dm*

Counts the number of LL cache references in the demand requests. References by external requests are not counted.

Compatibility Note For compatibility with previous versions of the CPU, *L2_read_dm* can be specified but is handled as *LL_read_dm* by the software in SPARC64™ XII (such as cpustat).

2 *LL_read_pf*

Counts the number of LL cache references in the prefetch requests.

Compatibility Note For compatibility with previous versions of the CPU, *L2_read_pf* can be specified but is handled as *LL_read_pf* by the software in SPARC64™ XII (such as cpustat).

3 *LL_miss_dm*

Counts the number of LL cache misses caused by demand requests. This counter is the sum of *LL_miss_counts_dm_bank{0,1,2,3}*.

Compatibility Note For compatibility with previous versions of the CPU, *L2_miss_dm* can be specified but is handled as *LL_miss_dm* by the software in SPARC64™ XII (such as cpustat).

4 *LL_miss_pf*

Counts the number of LL cache misses caused by prefetch requests. This counter is the sum of *LL_miss_counts_pf_bank {0, 1, 2, 3}*.

Compatibility Note For compatibility with previous versions of the CPU, *L2_miss_pf* can be specified but is handled as *LL_miss_pf* by the software in SPARC64™ XII (such as cpustat).

5 *LL_miss_counts_dm_bank {0, 1, 2, 3}*

Counts the number of LL cache misses for each bank caused by demand requests.

When an LL 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 the memory, an external LCU, or an external CPU, the demand request is not counted in *LL_miss_counts_dm_bank{0,1,2,3}*.

Compatibility Note For compatibility with previous versions of the CPU, *L2_miss_counts_dm_bank{0, 1, 2, 3}* can be specified but is handled as *LL_miss_counts_dm_bank{0, 1, 2, 3}* by the software in SPARC64™ XII (such as cpustat).

6 *LL_miss_counts_pf_bank {0, 1, 2, 3}*

Counts the number of LL cache misses for each bank caused by prefetch requests.

Compatibility Note For compatibility with previous versions of the CPU, *L2_miss_count_pf_bank{0, 1, 2, 3}* can be specified but is handled as *LL_miss_count_pf_bank{0, 1, 2, 3}* by the software in SPARC64™ XII (such as cpustat).

7 *LL_miss_wait_dm_bank {0, 1, 2, 3}*

Counts the total time spent on processing LL cache misses for each bank caused by demand requests (that is, the total miss latency for each bank). The latency of each memory access request is counted.

When an LL 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 the memory, an external LCU, or an external CPU, the cycles are counted in *LL_miss_wait_dm_bank{0,1,2,3}* after the demand request but before the data is received.

Compatibility Note For compatibility with previous versions of the CPU, *L2_miss_wait_dm_bank{0, 1, 2, 3}* can be specified but is handled as *LL_miss_wait_dm_bank{0, 1, 2, 3}* by the software in SPARC64™ XII (such as cpustat).

8 *LL_miss_wait_pf_bank {0, 1, 2, 3}*

Counts the total time spent on processing LL cache misses for each bank caused by prefetch requests, (that is, the total miss latency for each bank). The latency of each memory access request is counted.

The LL cache miss latencies can be derived by summing *LL_miss_wait_** and then dividing by the sum of *LL_miss_counts_**.

If individual LL cache-miss latencies are calculated for pf/dm requests, the value obtained for the miss latency of dm requests may be higher than expected.

Compatibility Note For compatibility with previous versions of the CPU, *L2_miss_wait_pf_bank{0, 1, 2, 3}* can be specified but is handled as *LL_miss_wait_pf_bank{0, 1, 2, 3}* by the software in SPARC64™ XII (such as cpustat).

9 *LL_wb_dm*

Counts the number of writebacks to memory caused by LL cache misses for the demand requests.

Compatibility Note For compatibility with previous versions of the CPU, *L2_wb_dm* can be specified but is handled as *LL_wb_dm* by the software in SPARC64™ XII (such as cpustat).

10 *LL_wb_pf*

Counts the number of writebacks to memory caused by LL cache misses for the prefetch requests.

Compatibility Note For compatibility with previous versions of the CPU, *L2_wb_pf* can be specified but is handled as *LL_wb_pf* by the software in SPARC64™ XII (such as cpustat).

Supplemental PA Events

11 *lost_pf_pfp_full*

Counts the number of weak prefetch requests that are lost due to LL-PF port full.

12 *lost_pf_by_abort*

Counts the number of weak prefetch requests that are lost due to LL-pipe abort.

11.2.5 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. 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 LCU. 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 in the LCU.

6 *cpi_counts*

Counts the number of external cache-copy-and-invalidate requests received by the LCU. These requests copy updated cache data to the memory before invalidating for inter CPU-chip copies. Cache data that is consistent with the memory does not need to be copied and is invalidated. For this event, the same value is counted by all VCPUs in the LCU.

7 *cpb_counts*

Counts the number of external cache-copyback requests received by the LCU. These requests copy updated cache data to the memory for inter CPU-chip copies. For this event, the same value is counted by all VCPUs in the LCU.

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 in the LCU.

Supplemental PA Events

9 *IO_pst_counts*

Counts the number of memory write requests (IO-PST) issued by I/Os.

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 (such as executing instructions, waiting for memory access, and waiting for an execution to be completed).

SPARC64™ XII defines 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™ XII executes instructions out-of-order and has multiple execution units. The CPU is generally in a mixed state where instructions are being executed or waiting. One instruction may be waiting for data from memory, another executing a floating-point multiplication, 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™ XII 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 an instruction execution to be completed.
- An instruction fetch is unable to supply the pipeline with instructions.

Table 11-2 highlights some useful PA events and descibes how they can be used to analyze the execution efficiency.

Figure 11-1 shows the relationship between the various *op_stv_wait_** events. The PA events marked with a † in the figure are synthetic events calculated from other PA events.

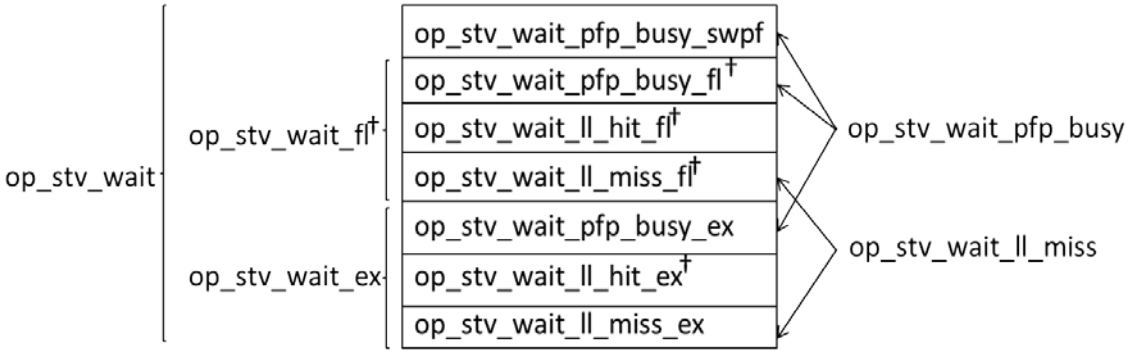


Figure 11-1 Breakdown of *op_stv_wait*

Table 11-2 Useful Performance Events for Cycle Accounting

Instructions Committed per Cycle	Cycles	Remarks
4	*cycle_counts* - *3endop* - *2endop* - *1endop* - *0endop*	N/A (maximum number of instructions are committed)
3	*3endop*	
2	*2endop*	
1	*1endop*	
0	Execution: *eu_comp_wait* + *branch_comp_wait* + *d_move_wait*	*eu_comp_wait* = *ex_comp_wait*+ *fl_comp_wait*
	Instruction Fetch: *cse_window_empy*	
	L1D cache miss: *op_stv_wait* -*op_stv_wait_l2_miss* -*op_stv_wait_ll_miss*	
	L2 cache miss: *op_stv_wait_l2_miss*	
	LL cache miss: *op_stv_wait_ll_miss*	
	Waiting Other Thread: *other_thread_commit*	
	Others: *0endop* - *op_stv_wait* - *cse_window_empy* - *eu_comp_wait* - *branch_comp_wait* - *d_move_wait* - *other_thread_commit* -(*instruction_flow_counts* - *instruction_counts*)	

12. Traps

12.5. Trap list and priorities

Symbol	Description
-x-	Traps will not occur in this mode.
P	Change to privileged mode.
P(ie)	Change to privileged mode if PSTATE.ie = 1.
H	Change to hyperprivileged mode.

Table 12-1 Trap list, by TT value

TT	Trap name	Type	Priority	Privilege level after the traps occur	Definition
000_{16}	*reserved*	—	—	—	—
006_{16}	*reserved*	—	—	—	—
007_{16}	*reserved*	—	—	—	—
008_{16}	*IAE_privilege_violation*	precise	3.1	H	
$00B_{16}$	*IAE_unauth_access*	precise	2.7	H	
$00C_{16}$	*IAE_nfo_page*	precise	3.3	H	
$00D_{16}$	*reserved*	—	—	—	—
$00E_{16}$	*reserved*	—	—	—	—
$00F_{16}$	*reserved*	—	—	—	—
010_{16}	*illegal_instruction*	precise	6.2	H	
011_{16}	*privileged_opcode*	precise	7	P	
012_{16}	*reserved*	—	—	—	—
013_{16}	*reserved*	—	—	—	—
014_{16}	*DAE_invalid_asi*	precise	12.1	H	
015_{16}	*DAE_privilege_violation*	precise	12.5	H	
016_{16}	*DAE_nc_page*	precise	12.6	H	
017_{16}	*DAE_nfo_page*	precise	12.7	H	
018_{16}-$01F_{16}$	*reserved*	—	—	—	—

TT	Trap name	Type	Priority	Privilege level after the traps occur	Definition
020_{16}	*fp_disabled*	precise	8	P	
021_{16}	*fp_exception_ieee_754*	precise	11.1	P	
022_{16}	*fp_exception_other*	precise	11.1	P	
023_{16}	*tag_overflow*	precise	14	P	
024_{16}	*clean_window*	precise	10.1	P	
025_{16}-027_{16}	*reserved*	—	—	—	—
028_{16}	*division_by_zero*	precise	15	P	
029_{16}	*reserved*	—	—	—	—
$02C_{16}$	*reserved*				—
$02D_{16}$	*reserved*	—	—	—	—
$02E_{16}$	*reserved*	—	—	—	—
$02F_{16}$	*reserved*	—	—	—	—
030_{16}	*DAE_side_effect_page*	precise	12.7	H	
033_{16}	*reserved*	—	—	—	—
034_{16}	*mem_address_not_aligned*	precise	10.2	H	
035_{16}	*LDDF_mem_address_not_aligned*	precise	10.1	H	
036_{16}	*STDF_mem_address_not_aligned*	precise	10.1	H	
037_{16}	*privileged_action*	precise	11.1	H	
038_{16}	*reserved*	—	—	—	—
039_{16}	*reserved*	—	—	—	—
$03C_{16}$	*reserved*	—	—	—	—
$03D_{16}$	*reserved*	—	—	—	—
041_{16}-$04F_{16}$	*interrupt_level_n* (n = 1 – 15) (*Interrupt_level*_15 is written as *pic_overflow*.)	disrupting	32-n[i]	P(ie)	
050_{16}-$05D_{16}$	*reserved*	—	—	—	—
061_{16}	*PA_watchpoint* (*RA_watchpoint*)	precise	12.9	H	
062_{16}	*VA_watchpoint*	precise	11.2	H	
065_{16}-067_{16}	*reserved*	—	—	—	—
069_{16}-$06B_{16}$	*reserved*	—	—	—	—
$06D_{16}$-070_{16}	*reserved*	—	—	—	—
073_{16}	*illegal_action*	precise	8.5	H	
074_{16}	*control_transfer_instruction*	precise	11.1	P	
075_{16}	*reserved*	—	—	—	—

[i] In UA2011, the priorities of *interrupt_level_15* and *pic_overflow* are different. In SPARC64™ XII, both have a priority of 17.

TT	Trap name	Type	Priority	Privilege level after the traps occur	Definition
078_{16}-$07B_{16}$	*reserved*	—	—	—	—
$07C_{16}$	*cpu_mondo*	disrupting	16.8	P(ie)	
$07D_{16}$	*dev_mondo*	disrupting	16.11	P(ie)	
$07E_{16}$	*resumable_error*	disrupting	33.3	P(ie)	
$07F_{16}$	*nonresumable_error* (not by hardware)		—	—	
080_{16}-$09C_{16}$	*spill_n_normal* (n = 0 – 7)	precise	9	P	
$0A0_{16}$-$0BC_{16}$	*spill_n_other* (n = 0 – 7)	precise	9	P	
$0C0_{16}$-$0DC_{16}$	*fill_n_normal* (n = 0 – 7)	precise	9	P	
$0E0_{16}$-$0FC_{16}$	*fill_n_other* (n = 0 – 7)	precise	9	P	
100_{16}-$17F_{16}$	*trap_instruction*	precise	16.2	P	

Table 12-2 Trap list, by priority

TT	Trap name	Type	Priority	Privilege level after the trap occur	Definition
$00B_{16}$	*IAE_unauth_access*	precise	2.7	H	
008_{16}	*IAE_privilege_violation*	precise	3.1	H	
$00C_{16}$	*IAE_nfo_page*	precise	3.3	H	
010_{16}	*illegal_instruction*	precise	6.2	H	
011_{16}	*privileged_opcode*	precise	7	P	
020_{16}	*fp_disabled*	precise	8	P	
073_{16}	*illegal_action*	precise	8.5	H	
080_{16}-$09C_{16}$	*spill_n_normal* (n = 0 – 7)	precise	9	P	
$0A0_{16}$-$0BC_{16}$	*spill_n_other* (n = 0 – 7)	precise	9	P	
$0C0_{16}$-$0DC_{16}$	*fill_n_normal* (n = 0 – 7)	precise	9	P	
$0E0_{16}$-$0FC_{16}$	*fill_n_other* (n = 0 – 7)	precise	9	P	
024_{16}	*clean_window*	precise	10.1	P	
035_{16}	*LDDF_mem_address_not_aligned*	precise	10.1	H	
036_{16}	*STDF_mem_address_not_aligned*	precise	10.1	H	
034_{16}	*mem_address_not_aligned*	precise	10.2	H	
021_{16}	*fp_exception_ieee_754*	precise	11.1	P	
022_{16}	*fp_exception_other*	precise	11.1	P	

TT	Trap name	Type	Priority	Privilege level after the trap occur	Definition
037_{16}	*privileged_action*	precise	11.1	H	
074_{16}	*control_transfer_instruction*	precise	11.1	P	
062_{16}	*VA_watchpoint*	precise	11.2	H	
014_{16}	*DAE_invalid_asi*	precise	12.1	H	
015_{16}	*DAE_privilege_violation*	precise	12.5	H	
016_{16}	*DAE_nc_page*	precise	12.6	H	
017_{16}	*DAE_nfo_page*	precise	12.7	H	
030_{16}	*DAE_side_effect_page*	precise	12.7	H	
061_{16}	*PA_watchpoint* (*RA_watchpoint*)	precise	12.9	H	
023_{16}	*tag_overflow*	precise	14	P	
028_{16}	*division_by_zero*	precise	15	P	
100_{16}-$17F_{16}$	*trap_instruction*	precise	16.2	P	
$07C_{16}$	*cpu_mondo*	disrupting	16.8	P(ie)	
$07D_{16}$	*dev_mondo*	disrupting	16.11	P(ie)	
041_{16}-$04F_{16}$	*interrupt_level_n* (n = 1 – 15) (*Interrupt_level*_15 is written as *pic_overflow*.)	disrupting	32-n[ii]	P(ie)	
$07E_{16}$	*resumable_error*	disrupting	33.3	P(ie)	
$07F_{16}$	*nonresumable_error* (not by hardware)		—	—	

[ii] In UA2011, the priorities of *interrupt_level_15* and *pic_overflow* are different. In SPARC64™ XII, both have a priority of 17.

13. Memory Management Unit

This chapter provides information about the SPARC64™ XII Memory Management Unit. It describes the internal architecture of the MMU and how to program it.

13.1. Address types

The SPARC64™ XII MMUs support a 64-bit virtual address (VA) space (no VA hole) and a 48-bit real address (RA) 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 in SPARC64™ XII (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.

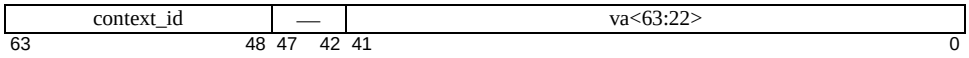
Table 13-1 the SPARC64™ XII address width

	VA	RA
Address width	64 bits	64 bits
Legal address width	64 bits (No VA hole)	48 bits

13.4. TSB Translation Table (TTE)

A TSB TTE contains the VA to RA translation for a single page mapping.

TTE Tag



TTE Data

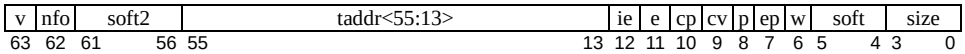


Table 13-2 TSB TTE

Bit		Field	Description
Tag	63:48	context_id	
Tag	41:0	va<63:22>	
Data	63	v	
Data	62	nfo	

<table>
<tr><td>Data 61:56</td><td>soft2</td><td></td></tr>
<tr><td>Data 55:13</td><td>taddr<55:13></td><td>Target address (RA).
In SPARC64™ XII, if the bits taadr<55:48> are not zero, an invalid_TSB_entry exception is generated.</td></tr>
<tr><td>Data 12</td><td>ie</td><td>This ie bit in the IMMU is ignored.</td></tr>
<tr><td>Data 11</td><td>e</td><td></td></tr>
<tr><td>Data 10</td><td>cp</td><td>This cp bit is ignored in SPARC64™ XII.</td></tr>
<tr><td>Data 9</td><td>cv</td><td>This cv bit is ignored in SPARC64™ XII.</td></tr>
<tr><td>Data 8</td><td>p</td><td></td></tr>
<tr><td>Data 7</td><td>ep</td><td></td></tr>
<tr><td>Data 6</td><td>w</td><td></td></tr>
<tr><td>Data 5:4</td><td>soft</td><td></td></tr>
<tr><td>Data 3:0</td><td>size</td><td>The page size of this entry is encoded as shown in the table below.
<table>
<tr><th>Size<3:0></th><th>Page size</th></tr>
<tr><td>0000</td><td>8KB</td></tr>
<tr><td>0001</td><td>64KB</td></tr>
<tr><td>0010</td><td>reserved</td></tr>
<tr><td>0011</td><td>4MB</td></tr>
<tr><td>0100</td><td>reserved</td></tr>
<tr><td>0101</td><td>256MB</td></tr>
<tr><td>0110</td><td>2GB</td></tr>
<tr><td>0111</td><td>16GB</td></tr>
<tr><td>1000-1111</td><td>reserved</td></tr>
</table></td></tr>
</table>

13.8. Page sizes

SPARC64™ XII supports six page sizes : 8 KB, 64 KB, 4 MB, 256 MB, 2GB, and 16GB. The TLBs can hold translations of all six sizes concurrently.

Table 13-3 Page types supported by SPARC64™ XII

Page type	Virtual page number	Page offset	Encode
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
2GB page	33 bits	31 bits	110_2
16GB	30 bits	34 bits	111_2

14. Opcode Maps

This chapter contains the opcode maps for the SPARC64™ XII instructions.

Opcodes marked with an em dash '—' are reserved. An attempt to execute a reserved opcode causes 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 26).

Table 14-1 op<1:0>

op<1:0>			
0	1	2	3
Branch instruction 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[PPCR] (rd = 16) WRPIC[PPIC] (rd = 17) WRGSR (rd = 19) 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 and 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[PNPT] (rs1 = 4, i = 0) RDPC (rs1 = 5, i = 0) RDFPRS (rs1 = 6, i = 0) MEMBAR (rs1 = 15, rd = 0, i = 1) RDPCR[PPCR] (rs1 = 16, i = 0) RDPIC[PPIC] (rs1 = 17, i = 0) RDGSR (rs1 = 19, i = 0) RDSTICK[PNPT] (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 instruction (op<1:0> = 3)

op3<3:0>	op3<5:4>			
	0	1	2	3
0	LDUW	LDUWA$^{P_{ASI}}$	LDF (urs2<2:1> = 00_2) LDFUW^{XII} (urs2<2:1> = 01_2) LDFSW^{XII} (urs2<2:1> = 11_2)	LDFA$^{P_{ASI}}$
1	LDUB	LDUBA$^{P_{ASI}}$	LDFSR^{D} (rd = 0) LDXFSR (rd = 1) LDXEFSR (rd = 3)	—
2	LDUH	LDUHA$^{P_{ASI}}$	LDQF	LDQFA$^{P_{ASI}}$
3	LDTW^{D} (rd even)	LDTWA$^{D,P_{ASI}}$ (rd even) LDTXA (rd even)	LDDF (urs2<1> = 0) LDDFDS^{XII} (urs2<1> = 1)	LDDFA$^{P_{ASI}}$ LDBLOCKF LDSHORTF
4	STW	STWA$^{P_{ASI}}$	STF (urs2<1> = 0) STFUW^{XII} (urs2<1> = 1)	STFA$^{P_{ASI}}$
5	STB	STBA$^{P_{ASI}}$	STFSR^{D} (rd = 0) STXFSR (rd = 1)	—
6	STH	STHA$^{P_{ASI}}$	STQF	STQFA$^{P_{ASI}}$
7	STTW^{D} (rd even)	STTWA$^{D,P_{ASI}}$ (rd even) STBI^{N} XFILL^{N}	STDF (urs2<1> = 0) STDFDS^{XII} (urs2<1> = 1)	STDFA$^{P_{ASI}}$ STBLOCKF STPARTIALF STSHORTF XFILL^{N}
8	LDSW	LDSWA$^{P_{ASI}}$	—	—
9	LDSB	LDSBA$^{P_{ASI}}$	—	—
A_{16}	LDSH	LDSHA$^{P_{ASI}}$	—	—
B_{16}	LDX	LDXA$^{P_{ASI}}$	—	—
C_{16}	—	—	STFR (type = 0 or i = 1) STFRUW^{XII} (type = 1)	CASA$^{P_{ASI}}$
D_{16}	LDSTUB	LDSTUBA$^{P_{ASI}}$	PREFETCH	PREFETCHA$^{P_{ASI}}$
E_{16}	STX	STXA$^{P_{ASI}}$ STBI^{N} XFILL^{N}	—	CASXA$^{P_{ASI}}$
F_{16}	SWAP^{D}	SWAPA$^{D,P_{ASI}}$	STDFR (type = 0 or i = 1) STDFRDS^{XII}(type = 1 and m = 0) STDFRDW^{XII}(type = 1 and m = 1)	—

Table 14-5 FPop1 (op<1:0> = 2, op3 = 34_{16}) (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}	—	FsTOx	FdTOx	FqTOx	FxTOs	—	—	—
09_{16}	—	—	—	—	—	—	—	—
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	—	—	—	—	FiTOs	—	FdTOs	FqTOs
$0D_{16}$	—	FsTOi	FdTOi	FqTOi	—	—	—	—
$0E_{16} – 1F_{16}$	—	—	—	—	—	—	—	—

Table 14-6 FPop1 (op<1:0> = 2, op3 = 34_{16}) (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}	FxTOd	—	—	—	FxTOq	—	—	—
09_{16}	—	—	—	—	—	—	—	—
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	FiTOd	FsTOd	—	FqTOd	FiTOq	FsTOq	FdTOq	—
$0D_{16}$	—	—	—	—	—	—	—	—
$0E_{16} – 1F_{16}$	—	—	—	—	—	—	—	—

Table 14-7 FPop2 (op<1:0> = 2, op3 = 35_{16})

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)	—	(Reserve for FMOVR enhance)			—
01_{16}	—	—	—	—	—	—	—	—	—
02_{16}	—	—	—	—	—	FMOVRsZ[iii]	FMOVRdZ[iii]	FMOVRqZ[iii]	—
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)	—	(Reserve for FMOVR enhance)			—
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}-$1F_{16}$	—	—	—	—	—	—	—	—	—

[iii] iw<13> = 0

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}	BPN	BN{D}	FBPN	FBN{D}	TN
1_{16}	BPE	BE{D}	FBPNE	FBNE{D}	TE
2_{16}	BPLE	BLE{D}	FBPLG	FBLG{D}	TLE
3_{16}	BPL	BL{D}	FBPUL	FBUL{D}	TL
4_{16}	BPLEU	BLEU{D}	FBPL	FBL{D}	TLEU
5_{16}	BPCS	BCS{D}	FBPUG	FBUG{D}	TCS
6_{16}	BPNEG	BNEG{D}	FBPG	FBG{D}	TNEG
7_{16}	BPVS	BVS{D}	FBPU	FBU{D}	TVS
8_{16}	BPA	BA{D}	FBPA	FBA{D}	TA
9_{16}	BPNE	BNE{D}	FBPE	FBE{D}	TNE
A_{16}	BPG	BG{D}	FBPUG	FBUG{D}	TG
B_{16}	BPGE	BGE{D}	FBPGE	FBGE{D}	TGE
C_{16}	BPGU	BGU{D}	FBPUGE	FBUGE{D}	TGU
D_{16}	BPCC	BCC{D}	FBPLE	FBLE{D}	TCC
E_{16}	BPPOS	BPOS{D}	FBPULE	FBULE{D}	TPOS
F_{16}	BPVC	BVC{D}	FBPO	FBO{D}	TVC

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 instruction (op<1 :0> = 2, op3 = 36_{16}) (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	FCMPLE16	—	—	FPADD16	FZERO	FAND
1_{16}	EDGE8N	—	—	FMUL8x16	—	FPADD16S	FZEROS	FANDS
2_{16}	EDGE8L	ARRAY16	FCMPNE16	—	FPADD64	FPADD32	FNOR	FXNOR
3_{16}	EDGE8LN	—	—	FMUL8x16AU	—	FPADD32S	FNORS	FXNORS
4_{16}	EDGE16	ARRAY32	FCMPLE32	—	—	FPSUB16	FANDNOT2	FSRC1
5_{16}	EDGE16N	—	FSLL32[XII]	FMUL8x16AL	—	FPSUB16S	FANDNOT2S	FSRC1S
6_{16}	EDGE16L	—	FCMPNE32	FMUL8sUx16	FPSUB64	FPSUB32	FNOT2	FORNOT2
7_{16}	EDGE16LN	LZD	FSRL32[XII]	FMUL8uLx16	—	FPSUB32S	FNOT2S	FORNOT2S
8_{16}	EDGE32	ALIGNADDRES	FCMPGT16	FMULD8sUx16	FALIGNDATA	—	FANDNOT1	FSRC2
9_{16}	EDGE32N	BMASK	—	FMULD8uLx16	—	—	FANDNOT1S	FSRC2S
A_{16}	EDGE32L	ALIGNADDRES_LITTLE	FCMPEQ16	FPACK32	—	—	FNOT1	FORNOT1
B_{16}	EDGE32LN	—	—	FPACK16	FPMERGE[XII]	—	FNOT1S	FORNOT1S
C_{16}	—	—	FCMPGT32	—	BSHUFFLE	—	FXOR	FOR
D_{16}	—	—	—	FPACKFIX	FEXPAND	—	FXORS	FORS

opf<3:0	opf<8:4>							
E_{16}	—	—	FCMPEQ32	PDIST	FPMUL64[XII]	—	FNAND	FONE
F_{16}	—	—	FSRA32[XII]	—	FPMUL32[XII]	—	FNANDS	FONES

Table 14-14 IMPDEP1 : VIS instruction (op<1 :0> = 2, op3 = 36_{16}) (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	FAESENCX	FADDtd	FADDod	FCMPLE16X (urs3<1:0> = 00_2) FPCMPLE16X (urs3<1:0> = 00_2) FPCMPLE16FX[XII] (urs3<1:0> = 10_2) FPCMPLE16XACC[XII] (urs3<1:0> = 11_2)	FCMPLE8X (urs3<1:0> = 00_2) FPCMPLE8X (urs3<1:0> = 00_2) FPCMPLE8FX[XII] (urs3<1:0> = 10_2) FPCMPLE8XACC[XII] (urs3<1:0> = 11_2)	—	—
1_{16}	SIAM	FAESDECX	FSUBtd	FSUBod	FUCMPLE16X (urs3<1:0> = 00_2) FPCMPULE16X (urs3<1:0> = 00_2) FPCMPULE16FX [XII] (urs3<1:0> = 10_2) FPCMPULE16XACC[XII] (urs3<1:0> = 11_2)	FUCMPLE8X (urs3<1:0> = 00_2) FPCMPULE8X (urs3<1:0> = 00_2) FPCMPULE8FX[XII] (urs3<1:0> = 10_2) FPCMPULE8XACC[XII] (urs3<1:0> = 11_2)	—	—
2_{16}		FAESENCLX	FMULtd	FMULod	FPCMPLE4X[XII] (urs3<1:0> = 00_2) FPCMPLE4FX[XII] (urs3<1:0> = 10_2) FPCMPLE4XACC[XII] (urs3<1:0> = 11_2)	FPCMPGT4X[XII] (urs3<1:0> = 00_2) FPCMPGT4FX[XII] (urs3<1:0> = 10_2) FPCMPGT4XACC[XII] (urs3<1:0> = 11_2)	—	—
3_{16}	SLEEP	FAESDECLX	FDIVtd	FDIVod	FUCMPNE16X (urs3<1:0> = 00_2) FPCMPUNE16X (urs3<1:0> = 00_2) FPCMPUNE16FX[XII] (urs3<1:0> = 10_2) FPCMPUNE16XACC[XII] (urs3<1:0> = 11_2)	FUCMPNE8X (urs3<1:0> = 00_2) FPCMPUNE8X (urs3<1:0> = 00_2) FPCMPUNE8FX[XII] (urs3<1:0> = 10_2) FPCMPUNE8XACC[XII] (urs3<1:0> = 11_2)	—	—

4_{16}	—	FAESKEYX	FCMPtd	FCMPod	FCMPLE32X (urs3<1:0> = 00_2) FPCMPLE32X (urs3<1:0> = 00_2) FPCMPLE32FX[XII] (urs3<1:0> = 10_2) FPCMPLE32XACC[XII] (urs3<1:0> = 11_2)	FCMPLE64X (urs3<1:0> = 00_2) FPCMPLE64X (urs3<1:0> = 00_2) FPCMPLE64FX[XII] (urs3<1:0> = 10_2) FPCMPLE64XACC[XII] (urs3<1:0> = 11_2)	FPMAX32x	—
5_{16}	SDIAM	FPSELMOV8X (urs3<1> = 0) FPSELMOV8FX[XII] (urs3<1> = 1)	FCMPEtd	—	FUCMPLE32X (urs3<1:0> = 00_2) FPCMPULE32X (urs3<1:0> = 00_2) FPCMPULE32FX[XII] (urs3<1:0> = 10_2) FPCMPULE32XACC[XII] (urs3<1:0> = 11_2)	FUCMPLE64X (urs3<1:0> = 00_2) FPCMPULE64X (urs3<1:0> = 00_2) FPCMPULE64FX[XII] (urs3<1:0> = 10_2) FPCMPULE64XACC[XII] (urs3<1:0> = 11_2)	FPMAXu32x	—
6_{16}	—	FPSELMOV16X (urs3<1> = 0) FPSELMOV16FX[XII] (urs3<1> = 1)	FQUAtd	FQUAod	FPCMPULE4X[XII] (urs3<1:0> = 00_2) FPCMPULE4FX[XII] (urs3<1:0> = 10_2) FPCMPULE4XACC[XII] (urs3<1:0> = 11_2)	FPCMPUGT4X[XII] (urs3<1:0> = 00_2) FPCMPUGT4FX[XII] (urs3<1:0> = 10_2) FPCMPUGT4XACC[XII] (urs3<1:0> = 11_2)	FPMIN32x	—
7_{16}	—	FPSELMOV32X (urs3<1> = 0) FPSELMOV32FX[XII] (urs3<1> = 1)	—	FRQUAod	FUCMPNE32X (urs3<1:0> = 00_2) FPCMPUNE32X (urs3<1:0> = 00_2) FPCMPUNE32FX[XII] (urs3<1:0> = 10_2) FPCMPUNE32XACC[XII] (urs3<1:0> = 11_2)	FUCMPNE64X (urs3<1:0> = 00_2) FPCMPUNE64X (urs3<1:0> = 00_2) FPCMPUNE64FX[XII] (urs3<1:0> = 10_2) FPCMPUNE64XACC[XII] (urs3<1:0> = 11_2)	FPMINu32X	—
8_{16}	—	FDESENCX	—	FXADDodLO	FCMPGT16X (urs3<1:0> = 00_2) FPCMPGT16X (urs3<1:0> = 00_2) FPCMPGT16FX[XII] (urs3<1:0> = 10_2) FPCMPGT16XACC[XII] (urs3<1:0> = 11_2)	FCMPGT8X (urs3<1:0> = 00_2) FPCMPGT8X (urs3<1:0> = 00_2) FPCMPGT8FX[XII] (urs3<1:0> = 10_2) FPCMPGT8XACC[XII] (urs3<1:0> = 11_2)	—	—
9_{16}	PADD32	FDESPC1X	—	FXADDodHI	FUCMPGT16X (urs3<1:0> = 00_2) FPCMPUGT16X (urs3<1:0> = 00_2) FPCMPUGT16FX[XII] (urs3<1:0> = 10_2) FPCMPUGT16XACC[XII] (urs3<1:0> = 11_2)	FUCMPGT8X (urs3<1:0> = 00_2) FPCMPUGT8X (urs3<1:0> = 00_2) FPCMPUGT8FX[XII] (urs3<1:0> = 10_2) FPCMPUGT8XACC[XII] (urs3<1:0> = 11_2)	—	—
A_{16}	—	FDESIPX	—	FXMULodLO	—	—	—	—
B_{16}	—	FDESIIPX	—	—	FUCMPEQ16X (urs3<1:0> = 00_2) FPCMPUEQ16X (urs3<1:0> = 00_2) FPCMPUEQ16FX[XII] (urs3<1:0> = 10_2) FPCMPUEQ16XACC[XII] (urs3<1:0> = 11_2)	FUCMPEQ8X (urs3<1:0> = 00_2) FPCMPUEQ8X (urs3<1:0> = 00_2) FPCMPUEQ8FX[XII] (urs3<1:0> = 10_2) FPCMPUEQ8XACC[XII] (urs3<1:0> = 11_2)	—	—

C_{16}	—	FDESKEYX	FbuxTOtd	—	FCMPGT32X (urs3<1:0> = 00_2) FPCMPGT32X (urs3<1:0> = 00_2) FPCMPGT32FX[XII] (urs3<1:0> = 10_2) FPCMPGT32XACC[XII] (urs3<1:0> = 11_2)	FCMPGT64X (urs3<1:0> = 00_2) FPCMPGT64X (urs3<1:0> = 00_2) FPCMPGT64FX[XII] (urs3<1:0> = 10_2) FPCMPGT64XACC[XII] (urs3<1:0> = 11_2)	FPMAX64x	—
D_{16}	—	—	FtdTObux	—	FUCMPGT32X (urs3<1:0> = 00_2) FPCMPUGT32X (urs3<1:0> = 00_2) FPCMPUGT32FX[XII] (urs3<1:0> = 10_2) FPCMPUGT32XACC[XII] (urs3<1:0> = 11_2)	FUCMPGT64X (urs3<1:0> = 00_2) FPCMPUGT64X (urs3<1:0> = 00_2) FPCMPUGT64FX[XII] (urs3<1:0> = 10_2) FPCMPUGT64XACC[XII] (urs3<1:0> = 11_2)	FPMAXu64x	—
E_{16}	FMONTMUL[XII] FMONTSQR[XII]	FPCSL8X[XII]	FbsxTOtd	FodTOtd	FPCMPUNE4X[XII] (urs3<1:0> = 00_2) FPCMPUNE4FX[XII] (urs3<1:0> = 10_2) FPCMPUNE4XACC[XII] (urs3<1:0> = 11_2)	FPCMPUEQ4X[XII] (urs3<1:0> = 00_2) FPCMPUEQ4FX[XII] (urs3<1:0> = 10_2) FPCMPUEQ4XACC[XII] (urs3<1:0> = 11_2)	FPMIN64x	—
F_{16}	—	FPADD128XHI	FtdTObsx	FtdTOod	FUCMPEQ32X (urs3<1:0> = 00_2) FPCMPUEQ32X (urs3<1:0> = 00_2) FPCMPUEQ32FX[XII] (urs3<1:0> = 10_2) FPCMPUEQ32XACC[XII] (urs3<1:0> = 11_2)	FUCMPEQ64X (urs3<1:0> = 00_2) FPCMPUEQ64X (urs3<1:0> = 00_2) FPCMPUEQ64FX[XII] (urs3<1:0> = 10_2) FPCMPUEQ64XACC[XII] (urs3<1:0> = 11_2)	FPMINu64x	—

Table 14-15 IMPDEP1 : VIS instruction (op<1 :0> = 2, op3 = 36_{16}) (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}	19_{16}-$1F_{16}$
0_{16}	FSEXTW[XII]	MOVdTOx[XII]	FPCMPULE8	—	—	—	FCMPEQd	FMAXd	—	—
1_{16}	FZEXTW[XII]	MOVsTOuw[XII] (urs3<0> = 0) MOVfwTOuw[XII] (urs3<0> = 1)	—	—	—	FLCMPs	FCMPEQs	FMAXs	—	—
2_{16}	—	—	FPCMPUNE8	—	—	FLCMPd	FCMPEQEd	FMINd	—	—
3_{16}	—	MOVsTOsw[XII] (urs3<0> = 0) MOVfwTOsw[XII] (urs3<0> = 1)	—	—	—	—	FCMPEQEs	FMINs	—	—
4_{16}	FPCMP64X	—	FPADD8[XII]	—	—	FPSUB8[XII]	FCMPLEEd	FRCPAd	FEPERM32X[XII]	—
5_{16}	FPCMPU64X	—	—	—	—	—	FCMPLEEs	FRCPAs	FEPERM64X[XII]	—
6_{16}	FPSLL64X	—	—	—	—	—	FCMPLTEd	FRSQRTAd	—	—
7_{16}	FPSRL64X	—	—	—	—	—	FCMPLTEs	FRSQRTAs	—	—

8_{16}	—	MOVxTOd	FPCMPUGT8	—	—	—	FCMPNEd	FTRISSELd	—	—
9_{16}	—	MOVwTOs (urs3<1:0> = $0\ 0_2$) MOVwTOfuw[XII] (urs3<1:0> = $0\ 1_2$) MOVwTOfsw[XII] (urs3<1:0> = $1\ 1_2$)	—	—	—	—	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 = 37_{16})

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

Refer to the SPARC64 X/X+ specification.