4

Registers

A SPARC processor includes two types of registers: general-purpose or "working" data registers and control/status registers. The IU's general-purpose registers are called ***r*** registers, and the FPU's general-purpose registers are called ***f*** registers. Coprocessor working registers are coprocessor-implementation dependent.

IU control/status registers include:

- Processor State Register (PSR)
- Window Invalid Mask (WIM)
- Trap Base Register (TBR)
- Multiply/Divide Register (Y)
- Program Counters (PC, nPC)
- implementation-dependent Ancillary State Registers (ASRs)
- implementation-dependent IU Deferred-Trap Queue

FPU control/status registers include:

- Floating-Point State Register (FSR)
- implementation-dependent Floating-Point Deferred-Trap Queue (FQ)

Coprocessor (CP) control/status registers, if present, may include:

- implementation-dependent Coprocessor State Register (CSR)
- implementation-dependent Coprocessor Deferred-Trap Queue (CQ)

4.1. IU *r* Registers

An implementation of the IU may contain from 40 through 520 general-purpose 32-bit ***r*** registers. They are partitioned into 8 ***global*** registers, plus an implementation-dependent number of 16-register ***sets***. A register set is further partitioned into 8 ***in*** registers and 8 ***local*** registers. See Table 4-1.

Windowed *r* Registers

At a given time, an instruction can access the 8 *globals* and a 24-register **window** into the *r* registers. A register window comprises the 8 ***in*** and 8 ***local*** registers of a particular register set, together with the 8 *in* registers of an adjacent register set, which are addressable from the current window as ***out*** registers. See Figure 4-1.

The number of windows or register sets, NWINDOWS, ranges from 2 to 32, depending on the implementation. The total number of *r* registers in a given implementation is 8 (for the *globals*), plus the number of sets 16 registers/set. Thus, the minimum number of *r* registers is 40 (2 sets), and the maximum number is 520 (32 sets).

Table 4-1 *Window Addressing*

Windowed Register Address	*r* Register Address
in[0] – in[7]	r[24] – r[31]
local[0] – local[7]	r[16] – r[23]
out[0] – out[7]	r[8] – r[15]
global[0] – global[7]	r[0] – r[7]

The current window into the *r* registers is given by the current window pointer (CWP), a 5-bit counter field in the Processor State Register (PSR). The CWP is incremented by a RESTORE (or RETT) instruction and decremented by a SAVE instruction or a trap. Window overflow and underflow are detected via the window invalid mask (WIM) register, which is controlled by supervisor software.

Overlapping of Windows

Each window shares its *ins* and *outs* with the two adjacent windows. The *outs* of the CWP+1 window are addressable as the *ins* of the current window, and the *outs* in the current window are the *ins* of the CWP−1 window. The *locals* are unique to each window.

An *r* register with address *o*, where $8 \le o \le 15$, refers to exactly the same register as $(o + 16)$ does after the CWP is decremented by 1 (modulo NWINDOWS). Likewise, a register with address *i*, where $24 \le i \le 31$, refers to exactly the same register as address $(i - 16)$ does after the CWP is incremented by 1 (modulo NWINDOWS). See Figure 4-2.

Since CWP arithmetic is performed modulo NWINDOWS, the highest numbered implemented window overlaps with window 0. The *outs* of window 0 are the *ins* of window NWINDOWS−1. Implemented windows must be contiguously numbered from 0 through NWINDOWS−1.

Programming Note

Since the procedure call instructions (CALL and JMPL) do not change the CWP, a procedure can be called without changing the window. See Appendix D, “Software Considerations.”

Because the windows overlap, the number of windows available to software is 1 less than the number of implemented windows, or NWINDOWS−1. When the register file is full, the *outs* of the newest window are the *ins* of the oldest window — which still contains valid program data.

No assumptions can be made regarding the values contained in the “local” and “out” regsiters of a register window upon re-entering the window through a SAVE instruction. If, with traps enabled, a program executes a RESTORE followed by a SAVE, the resulting window’s *locals* and *outs* may not be valid after the SAVE, since a trap may have occurred between the RESTORE and the

SAVE. However, with traps disabled, the *locals* and *outs* remain valid.

Doubleword Operands

Instructions that access a doubleword in the *r* registers assume even-odd register alignment. The least-significant bit of an *r* register address in these instructions is reserved, and for future compatibility should be supplied as zero by software.

An attempt to execute a doubleword load or store instruction that refers to a misaligned (odd) destination register number may cause an illegal_instruction trap.

Special *r* Registers

The utilization of four *r* registers is fixed, in whole or in part, by the architecture:

- If r[0] is addressed as a source operand (*rs1* = 0 or *rs2* = 0, or *rd* = 0 for a Store) the constant value 0 is read. When r[0] is used as a destination operand (*rd* = 0, excepting Stores), the data written is discarded (no *r* register is modified).
- The CALL instruction writes its own address into register r[15] (*out* register 7).
- When a trap occurs, the program counters PC and nPC are copied into registers r[17] and r[18] (*local* registers 1 and 2) of the trap's new register window.

Register Usage

See Appendix D, "Software Considerations," for a description of conventional usage of the *r* registers.

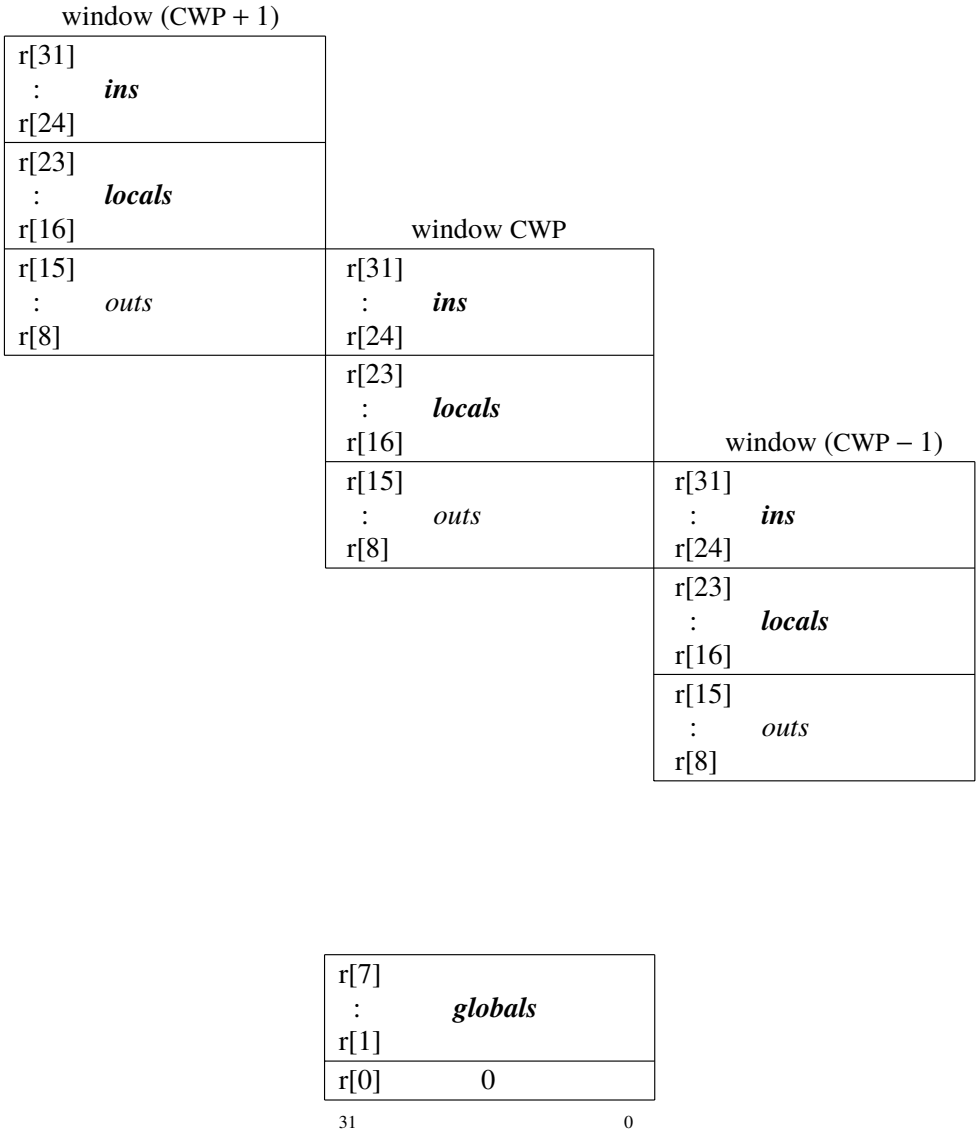


Figure 4-1 *Three Overlapping Windows and the 8 Global Registers*

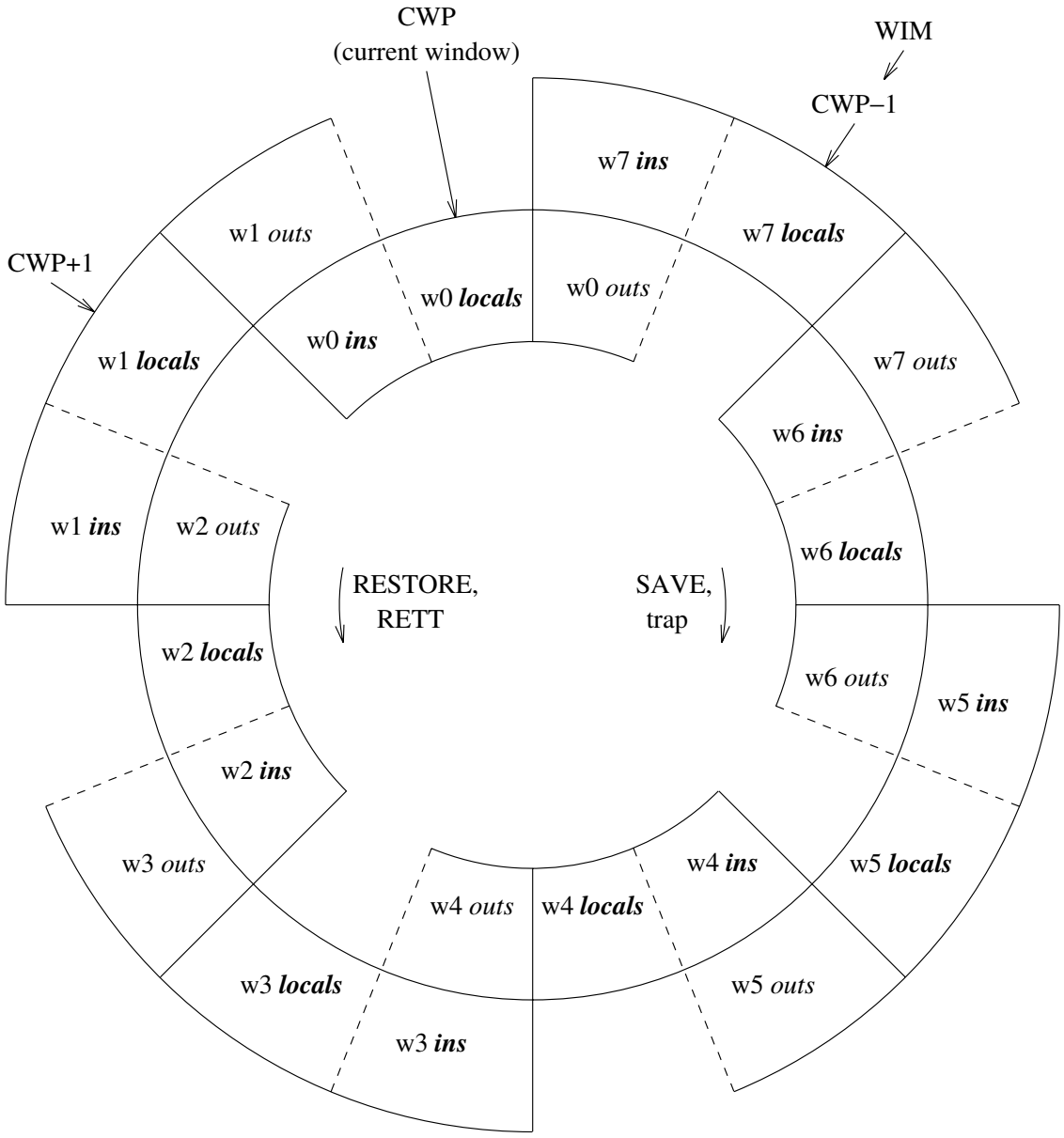


Figure 4-2 *The Windowed r Registers*

In Figure 4-1, NWINDOWS = 8. The 8 *globals* are not illustrated. The register sets are indicated in bold face. CWP = 0 and WIM[7] = 1. If the procedure using window w0 executes a RESTORE, window w1 will become the current window. If the procedure using window w0 executes a SAVE, a window_overflow trap will occur. The overflow trap handler uses the w7 *locals*.

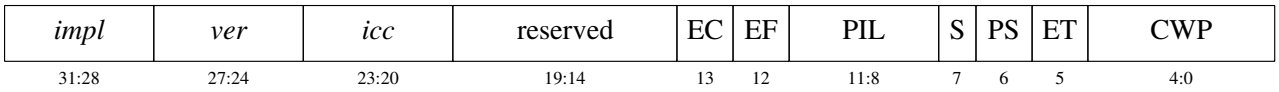
4.2. IU Control/Status Registers

The 32-bit IU control/status registers include the Processor State Register (PSR), the Window Invalid Mask register (WIM), the Trap Base Register (TBR), the multiply/divide (Y) register, the program counters (PC and nPC), and optional, implementation-dependent Ancillary State Registers (ASRs) and the IU deferred-trap queue.

Processor State Register (PSR)

The 32-bit PSR contains various fields that control the processor and hold status information. It can be modified by the SAVE, RESTORE, Ticc, and RETT instructions, and by all instructions that modify the condition codes. The privileged RDPSR and WRPSR instructions read and write the PSR directly.

Figure 4-3 *PSR Fields*



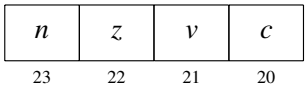
The PSR provides the following fields:

PSR_implementation (*impl*) Bits 31 through 28 are hardwired to identify an implementation or class of implementations of the architecture. The hardware should not change this field in response to a WRPSR instruction. Together, the PSR.*impl* and PSR.*ver* fields define a **unique** implementation or class of implementations of the architecture. See Appendix L, "Implementation Characteristics."

PSR_version (*ver*) Bits 27 through 24 are implementation-dependent. The *ver* field is either hardwired to identify one or more particular implementations or is a readable and writable state field whose properties are implementation-dependent. See Appendix L, "Implementation Characteristics."

PSR_integer_cond_codes (*icc*) Bits 23 through 20 are the IU's condition codes. These bits are modified by the arithmetic and logical instructions whose names end with the letters **cc** (e.g., ANDcc), and by the WRPSR instruction. The Bicc and Ticc instructions cause a transfer of control based on the value of these bits, which are defined as follows:

Figure 4-4 *Integer Condition Codes (icc) Fields of the PSR*



PSR_negative (*n*) Bit 23 indicates whether the 32-bit 2's complement ALU result was negative for the last instruction that modified the *icc* field. 1 = negative, 0 = not negative.

PSR_zero (*z*)

Bit 22 indicates whether the 32-bit ALU result was zero for the last instruction that modified the *icc* field. 1 = zero, 0 = nonzero.

PSR_overflow (*v*)

Bit 21 indicates whether the ALU result was within the range of (was representable in) 32-bit 2's complement notation for the last instruction that modified the *icc* field. 1 = overflow, 0 = no overflow.

PSR_carry (*c*)

Bit 20 indicates whether a 2's complement carry out (or borrow) occurred for the last instruction that modified the *icc* field. Carry is set on addition if there is a carry out of bit 31. Carry is set on subtraction if there is borrow into bit 31. 1 = carry, 0 = no carry.

PSR_reserved

Bits 19 through 14 are reserved. When read by a RDPSR instruction, these bits deliver zeros. For future compatibility, supervisor software should only issue WRPSR instructions with zero values in this field.

PSR_enable_coprocessor (EC)

Bit 13 determines whether the implementation-dependent coprocessor is enabled. If disabled, a coprocessor instruction will trap. 1 = enabled, 0 = disabled. If an implementation does not support a coprocessor in hardware, PSR.EC should always read as 0 and writes to it should be ignored.

PSR_enable_floating-point (EF)

Bit 12 determines whether the FPU is enabled. If disabled, a floating-point instruction will trap. 1 = enabled, 0 = disabled. If an implementation does not support a hardware FPU, PSR.EF should always read as 0 and writes to it should be ignored.

Programming Note

Software can use the EF and EC bits to determine whether a particular process uses the FPU or CP. If a process does not use the FPU/CP, its registers do not need to be saved across a context switch.

PSR_proc_interrupt_level (PIL)

Bits 11 (the most significant bit) through 8 (the least significant bit) identify the interrupt level above which the processor will accept an interrupt. See Chapter 7, "Traps."

PSR_supervisor (S)

Bit 7 determines whether the processor is in supervisor or user mode. 1 = supervisor mode, 0 = user mode.

PSR_previous_supervisor (PS)

Bit 6 contains the value of the S bit at the time of the most recent trap.

PSR_enable_traps (ET)

Bit 5 determines whether traps are enabled. A trap automatically resets ET to 0. When ET=0, an interrupt request is ignored and an exception trap causes the IU to halt execution, which typically results in a reset trap that resumes execution at address 0. 1 = traps enabled, 0 = traps disabled. See Chapter 7, "Traps."

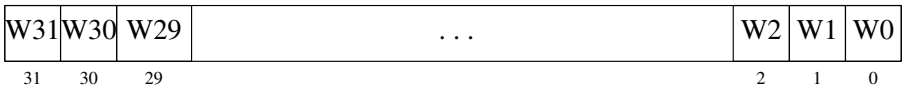
PSR_current_window_pointer (CWP)

Bits 4 (the MSB) through 0 (the LSB) comprise the current window pointer, a counter that identifies the current window into the *r* registers. The hardware decrements the CWP on traps and SAVE instructions, and increments it on RESTORE and RETT instructions (modulo NWINDOWS).

Window Invalid Mask Register (WIM)

The Window Invalid Mask register (WIM) is controlled by supervisor software and is used by hardware to determine whether a window overflow or underflow trap is to be generated by a SAVE, RESTORE, or RETT instruction.

Figure 4-5 *WIM Fields*



There is an active state bit in the WIM for each register set or window in an implementation. WIM[*n*] corresponds to the register set addressed when CWP = *n*.

When a SAVE, RESTORE, or RETT instruction executes, the current value of the CWP is compared against the WIM. If the SAVE, RESTORE, or RETT instruction would cause the CWP to point to an "invalid" register set, that is, one whose corresponding WIM bit equals 1 (WIM[CWP] = 1), a window_overflow or window_underflow trap is caused.

The WIM can be read by the privileged RDWIM instruction and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeroes and values written to unimplemented bits are unused. A WRWIM with all bits set to 1, followed by a RDWIM, yields a bit vector in which the implemented windows (and only the implemented windows) are indicated by 1's.

The WIM allows for implementations with up to 32 windows.

Trap Base Register (TBR)

The Trap Base Register (TBR) contains three fields that together equal the address to which control is transferred when a trap occurs.

Figure 4-6 *TBR Fields*

The TBR provides the following fields:

TBR_trap_base_address (TBA) Bits 31 through 12 are the trap base address, which is established by supervisor software. It contains the most-significant 20 bits of the trap table address. The TBA field is written by the WRTBR instruction.

TBR_trap_type (*tt*) Bits 11 through 4 comprise the trap type (*tt*) field. This 8-bit field is written by the hardware when a trap occurs, and retains its value until the next trap. It provides an offset into the trap table. The WRTBR instruction does not affect the *tt* field.

TBR_zero (0) Bits 3 through 0 are zeroes. The WRTBR instruction does not affect this field. For future compatibility, supervisor software should only issue a WRTBR instruction with a zero value in this field.

See Chapter 7, “Traps,” for additional information.

Multiply/Divide Register (Y)

The 32-bit Y register contains the most significant word of the double-precision product of an integer multiplication, as a result of either an integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction, or of a routine that uses the integer multiply step (MULScc) instruction. The Y register also holds the most significant word of the double-precision dividend for an integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

The Y register can be read and written with the RDY and WRY instructions.

Program Counters (PC, nPC)

The 32-bit PC contains the address of the instruction currently being executed by the IU. The nPC holds the address of the next instruction to be executed (assuming a trap does not occur).

For a delayed control transfer, the instruction that immediately follows the transfer instruction is known as the delay instruction. This delay instruction is executed (unless the control transfer instruction annuls it) before control is transferred to the target. During execution of the delay instruction, the nPC points to the target of the control transfer instruction, while the PC points to the delay instruction. See Chapter 5, “Instructions.”

The PC is read by a CALL or JMPL instruction. The PC and nPC are written to two *local* registers during a trap. See Chapter 7, “Traps,” for details.

Ancillary State Registers (ASR)

SPARC provides for up to 31 Ancillary State Registers (ASR’s), numbered from 1 to 31.

ASR’s numbered 1-15 are reserved for future use by the architecture and should not be referenced by software.

ASR’s numbered 16-31 are available for implementation-dependent uses, such as timers, counters, diagnostic registers, self-test registers, and trap-control registers. A particular IU may choose to implement from zero to sixteen of these ASR’s. The semantics of accessing any of these ASR’s is implementation-dependent. Whether a particular Ancillary State Register is privileged or not is implementation-dependent.

An ASR is read and written with the RDASR and WRASR instructions. A read/write ASR instruction is privileged if the accessed register is privileged.

IU Deferred-Trap Queue

An implementation may contain zero or more deferred-trap queues. Such a queue contains sufficient state to implement resumable deferred traps caused by the IU. Note that fp_exception and cp_exception deferred traps are handled by the floating-point and coprocessor deferred-trap queues.

An IU deferred-trap queue can be read and written via privileged load/store alternate or read/write ancillary state register instructions.

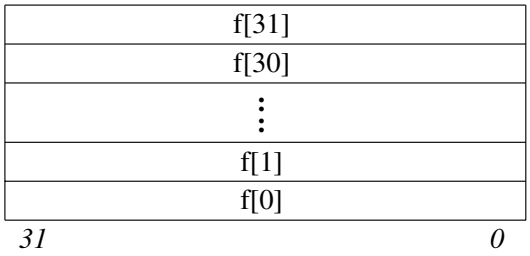
The contents and operation of an IU deferred-trap queue are implementation-dependent and are not visible to user application programs.

See Appendix L, "Implementation Characteristics," for a discussion of implemented queues.

4.3. FPU *f* Registers

The FPU contains 32 32-bit floating-point ***f*** registers, which are numbered from f[0] to f[31]. Unlike the windowed *r* registers, at a given time an instruction has access to any of the 32 *f* registers. The *f* registers can be read and written by FPop (FPop1/FPop2 format) instructions, and by load/store single/double floating-point instructions (LDF, LDDF, STF, STDF). See Figure 4-7.

Figure 4-7 *The f Registers*



Double and Quad Operands

A single *f* register can hold one single-precision operand. A double-precision operand requires an aligned pair of *f* registers, and a quad-precision operand requires an aligned quadruple of *f* registers. Thus, at a given time, the *f* registers can hold a maximum of 32 single-precision, 16 double-precision, or 8 quad-precision operands.

Instructions that access a floating-point double in the *f* registers assume double alignment. The least-significant bit of a doubleword *f* register address specifier is reserved and should be set to zero by software. Similarly, the least-significant two bits of a quadword *f* register address are reserved and should be set to zero by software. See Table 4-2.

Table 4-2 *Floating-Point Doubles and Quads in f Registers*

sub-format name	format fields	*f* register address
FD-0	s:exp[10:0]:fraction[51:32]	0 mod 2
FD-1	fraction[31:0]	1 mod 2
FQ-0	s:exp[14:0]:fraction[111:96]	0 mod 4
FQ-1	fraction[95:64]	1 mod 4
FQ-2	fraction[63:32]	2 mod 4
FQ-3	fraction[31:0]	3 mod 4

It is recommended (but not required) that an attempt to execute an instruction that refers to a mis-aligned floating-point register operand (double-precision operand in a register whose number is not 0 mod 2, or quadruple-precision operand in a register whose number is not 0 mod 4) cause an fp_exception trap with FSR.*ftt* = 6 (invalid_fp_register).

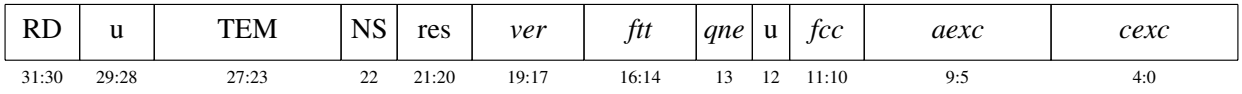
4.4. FPU Control/Status Registers

The 32-bit FPU control/status registers include a Floating-point State Register (FSR) that contains mode and status information about the FPU, and an optional, implementation-dependent, floating-point deferred-trap queue (FQ).

Floating-Point State Register (FSR)

The FSR register fields contain FPU mode and status information. The FSR is read and written by the STFSR and LDFSR instructions.

Figure 4-8 *FSR Fields*



The FSR provides the following fields:

FSR_rounding_direction (RD) Bits 31 and 30 select the rounding direction for floating-point results according to ANSI/IEEE Standard 754-1985.

Table 4-3 *Rounding Direction (RD) Field of FSR*

RD	Round Toward:
0	Nearest (even, if tie)
1	0
2	+ ∞
3	− ∞

FSR_unused (u)

Bits 29, 28, and 12 are unused. For future compatibility, software should only issue a LDFSR instruction with zero values in these bits.

FSR_trap_enable_mask (TEM)

Bits 27 through 23 are enable bits for each of the five floating-point exceptions that can be indicated in the current_exception field (*cexc*). See Figure 4-9. If a floating-point operate instruction generates one or more exceptions and the TEM bit corresponding to one or more of the exceptions is 1, an fp_exception trap is caused. A TEM value of 0 prevents that exception type from generating a trap.

FSR_nonstandard_fp (NS)

Bit 22, when set to 1, causes the FPU to produce implementation-defined results that may not correspond to ANSI/IEEE Standard 754-1985. For instance, to obtain higher performance, implementations may convert a subnormal floating-point operand or result to zero when NS is set. See Appendix L, "Implementation Characteristics," for a description of how this field has been used in existing implementations.

FSR_reserved (res)

Bits 21 and 20 are reserved. When read by an STFSR instruction, these bits deliver zeroes. For future compatibility, software should only issue LDFSR instructions with zero values in these bits.

FSR_version (*ver*)

Bits 19 through 17 identify one or more particular implementations of the FPU architecture. For each SPARC IU implementation (as identified by its PSR.*impl* and PSR.*vers* fields), there may be one or more FPU implementations, or none. This field identifies the particular FPU implementation present. Version number 7 is reserved to indicate that no hardware floating-point controller is present. See Appendix L, "Implementation Characteristics," for a description of how this field has been used in existing implementations.

FSR_floating-point_trap_type (*ftt*)

Bits 16 through 14 identify floating-point exception trap types. After a floating-point exception occurs, the *ftt* field encodes the type of floating-point exception until an STFSR or another FPop is executed.

The *ftt* field can be read by the STFSR instruction. An LDFSR instruction does not affect *ftt*.

Supervisor-mode software which handles floating-point traps must execute an STFSR to determine the floating-point trap type. Whether STFSR explicitly zeroes *ftt* is implementation-dependent; if STFSR does not zero *ftt*, then the trap software must ensure that a subsequent STFSR from user mode shows a value of zero for *ftt*.

Programming Note — LDFSR cannot be used for this purpose since it leaves *ftt* unchanged, although executing a non-trapping FPop such as "`fmovs %f0,%f0`" prior to returning to user mode will zero *ftt*. *ftt* remains valid until the next FPop instruction completes execution.

This field encodes the exception type according to Table 4-4. Note that value 7 is reserved for future expansion.

Table 4-4 *Floating-point Trap Type (ftt) Field of FSR*

ftt	Trap Type
0	None
1	IEEE_754_exception
2	unfinished_FPop
3	unimplemented_FPop
4	sequence_error
5	hardware_error
6	invalid_fp_register
7	*reserved*

The sequence_error and hardware_error trap types are not expected to arise in the normal course of computation. They are essentially unrecoverable, from the point of view of user applications.

In contrast, IEEE_754_exception, unfinished_FPop, and unimplemented_FPop are expected to arise occasionally in the normal course of computation and must be recoverable by supervisor software. When a floating-point trap occurs (as observed by a user signal (trap) handler):

1) The value of *aexc* is unchanged.
2) The value of *cexc* is unchanged, except that on an IEEE_754_exception exactly one bit corresponding to the trapping exception will be set. Unfinished_FPop, unimplemented_FPop, and sequence_error floating point exceptions do not affect *cexc*.
3) The source *f* registers are unchanged (usually implemented by leaving the destination *f* register unchanged).
4) The value of *fcc* is unchanged.

The foregoing describes the result seen by a user signal handler if an IEEE exception is signaled, either immediately from an IEEE_754_exception or after recovery from an unfinished_FPop or unimplemented_FPop. In either case, *cexc* as seen by the trap handler will reflect the exception causing the trap.

In the cases of unfinished_FPop and unimplemented_FPop traps that don't subsequently generate IEEE exceptions, the recovery software is expected to define *cexc*, *aexc*, and either the destination *f* register or *fcc*, as appropriate.

ftt = IEEE_754_exception

An IEEE_754_exception floating-point trap type indicates that a floating-point exception occurred that conforms to the ANSI/IEEE Standard 754-1985. The exception type is encoded in the *cexc* field. Note that *aexc*, *fcc*, and the destination *f* register are not affected by an IEEE_754_exception trap.

ftt = unfinished_FPop

An unfinished_FPop indicates that an implementation's FPU was unable to generate correct results or exceptions as defined by ANSI/IEEE Standard 754-1985. In this case, the *cexc* field is unchanged.

ftt = unimplemented_FPop

An unimplemented_FPop indicates that an implementation's FPU decoded an FPop that it does not implement. In this case, the *cexc* field is unchanged.

Programming Note

In the case of an unfinished_FPop or unimplemented_FPop floating-point trap type, software should emulate or re-execute the exception-causing instruction, and update the FSR, destination *f* register(s), and *fcc*.

ftt = sequence_error

A sequence_error indicates one of three abnormal error conditions in the FPU, all caused by erroneous supervisor software:

— An attempt was made to execute a STDFQ instruction on an implementation without a floating-point deferred-trap queue (FQ).

— An attempt was made to execute a floating-point instruction when the FPU was not able to accept one. This type of sequence_error arises from a logic error in supervisor software that has caused a previous floating-point trap to be incompletely serviced (for example, the floating-point queue was not emptied after a previous floating-point exception).

— An attempt was made to execute a STDFQ instruction when the floating-point deferred-trap queue (FQ) was empty, that is, when FSR.*qne* = 0. (Note that generation of sequence_error is recommended, but not required in this case)

Programming Note

If a sequence_error fp_exception occurs during execution of user code (due to either of the above conditions), it may not be possible to recover sufficient state to continue execution of the user application.

ftt = hardware_error

A hardware_error indicates that the FPU detected a catastrophic internal error, such as an illegal state or a parity error on an *f* register access.

If a hardware_error occurs during execution of user code, it may not be possible to recover sufficient state to continue execution of the user application.

ftt = invalid_fp_register

An invalid_fp_register trap type indicates that one (or more) operands of an FPop are misaligned, that is, a double-precision register number is not 0 mod 2, or a quadruple-precision register number is not 0 mod 4. It is recommended that implementations generate an fp_exception trap with FSR.*ftt* = invalid_fp_register in this case, but an implementation may choose not to generate a trap.

FSR_FQ_not_empty (*qne*)

Bit 13 indicates whether the optional floating-point deferred-trap queue (FQ) is empty after a deferred fp_exception trap or after a store double floating-point queue (STDFQ) instruction has been executed. If *qne* = 0, the queue is empty; if *qne* = 1, the queue is not empty.

The *qne* bit can be read by the STFSR instruction. The LDFSR instruction does not affect *qne*. However, executing successive STDFQ instructions will (eventually) cause the FQ to become empty (*qne* = 0). If an implementation does not provide an FQ, this bit reads as zero. Supervisor software must arrange for this bit to always read as zero to user mode software.

FSR_fp_condition_codes (*fcc*)

Bits 11 and 10 contain the FPU condition codes. These bits are updated by floating-point compare instructions (FCMP and FCMPE). They are read and written by the STFSR and LDFSR instructions, respectively. FBfcc bases its control transfer on this field.

In the following table, f_{rs1} and f_{rs2} correspond to the single, double, or quad values in the *f* registers specified by an instruction's *rs1* and *rs2* fields. The question mark (?) indicates an unordered relation, which is true if either f_{rs1} or f_{rs2} is a signaling NaN or quiet NaN. Note that *fcc* is unchanged if FCMP or FCMPE generates an IEEE_754_exception trap.

Table 4-5 *Floating-point Condition Codes (fcc) Field of FSR*

fcc	Relation
0	$f_{rs1} = f_{rs2}$
1	$f_{rs1} < f_{rs2}$
2	$f_{rs1} > f_{rs2}$
3	f_{rs1} ? f_{rs2} (unordered)

FSR_accrued_exception (*aexc*)

Bits 9 through 5 accumulate IEEE_754 floating-point exceptions while fp_exception traps are disabled using the TEM field. See Figure 4-10. After an FPop completes, the TEM and *cexc* fields are logically *and*'d together. If the result is nonzero, an fp_exception trap is generated; otherwise, the new *cexc* field is *or*'d into the *aexc* field. Thus, while traps are masked, exceptions are accumulated in the *aexc* field.

FSR_current_exception (*cexc*)

Bits 4 through 0 indicate that one or more IEEE_754 floating-point exceptions were generated by the most recently executed FPop instruction. The absence of an exception causes the corresponding bit to be cleared. See Figure 4-11.

The *cexc* bits are set as described in section 4.4.2 by the execution of an FPop that either does not cause a trap or causes an fp_exception trap with FSR.*ftt* = IEEE_754_exception. It is recommended that an IEEE_754_exception which traps should cause exactly one bit in FSR.*cexc* to be set, corresponding to the detected IEEE 754 exception. If the execution of an FPop causes a trap other than an fp_exception due to an IEEE 754 exception, FSR.*cexc* is left unchanged.

Floating-Point Exception Fields

The current and accrued exception fields and the trap enable mask assume the following definitions of the floating-point exception conditions (per ANSI/IEEE Standard 754-1985):

Figure 4-9 *Trap Enable Mask (TEM) Fields of FSR*

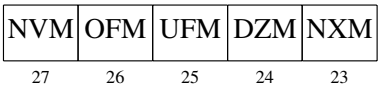


Figure 4-10 *Accrued Exception Bits (aexc) Fields of FSR*

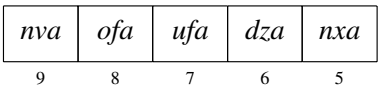
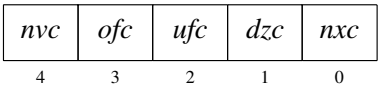


Figure 4-11 *Current Exception Bits (cexc) Fields of FSR*



FSR_invalid (*nvc, nva*)

An operand is improper for the operation to be performed. For example, 0 0, and ∞ − ∞ are invalid. 1 = invalid operand, 0 = valid operand(s).

FSR_overflow (*ofc, ofa*)

The rounded result would be larger in magnitude than the largest normalized number in the specified format. 1 = overflow, 0 = no overflow.

FSR_underflow (*ufc, ufa*)

The rounded result is inexact and would be smaller in magnitude than the smallest normalized number in the indicated format. 1 = underflow, 0 = no underflow.

Underflow is never indicated when the correct unrounded result is zero. Otherwise,

if UFM=0: The *ufc* and *ufa* bits will be set if the correct unrounded result of an operation is less in magnitude than the smallest normalized number and the correctly-rounded result is inexact. These bits will be set if the correct unrounded result is less than the smallest normalized number, but the correct rounded result is the smallest normalized number. *nxc* and *nxa* are always set as well.

if UFM=1: An IEEE-754_exception trap will occur if the correct unrounded result of an operation would be smaller than the smallest normalized number. A trap will occur if the correct unrounded result would be smaller than the smallest normalized number, but the correct rounded result would be the smallest normalized number.

FSR_division-by-zero (*dzc, dza*) X 0, where X is subnormal or normalized. Note that 0 0 does **not** set the *dzc* bit. 1 = division-by-zero, 0 = no division-by-zero.

FSR_inexact (*nxc, nxa*) The rounded result of an operation differs from the infinitely precise correct result. 1 = inexact result, 0 = exact result.

FSR Conformance

An implementation may choose to implement the TEM, *cexc*, and *aexc* fields in hardware in either of two ways:

(1) Implement all three fields conformant to ANSI/IEEE Standard 754-1985.

(2) Implement the NXM, *nxa*, and *nxc* bits of these fields conformant to ANSI/IEEE Standard 754-1985. Implement each of the remaining bits in the three fields either

(a) Conformant to the ANSI/IEEE Standard, or

(b) As a state bit that may be set by software which calculates the ANSI/IEEE value of the bit. For any bit implemented as a state bit:

- The IEEE exception corresponding to the state bit must **always** cause an exception (specifically, an unfinished_FPop exception). During exception processing in the trap handler, the bit in the state field can be written to the appropriate value by an LDFSR instruction, and
- The state bit must be implemented in such a way that if it is written to a particular value by an LDFSR instruction, it will be read back as the same value in a subsequent STFSR.

Programming Note The software must be capable of simulating the entire FPU to properly handle the unimplemented_FPop, unfinished_FPop, and IEEE_754_exception floating-point traps. Thus, a user application program always "sees" an FSR that is fully compliant with ANSI/IEEE Standard 754-1985.

Floating-Point Deferred-Trap Queue (FQ)

The floating-point deferred-trap queue (FQ), if present in an implementation, contains sufficient state information to implement resumable, deferred floating-point traps.

If floating-point instructions are to execute concurrently with (asynchronously from) integer instructions in a given implementation, the implementation must provide a floating-point queue. If floating-point instructions execute synchronously with integer instructions, provision of a floating-point queue is optional.

The FQ can be read with the privileged store double floating-point queue instruction (STDFQ). In a given implementation, it may also be readable or writable via privileged load/store double alternate (LDDA, STDA) instructions, or by read/write Ancillary State Register instructions (RDASR, WRASR).

The contents of and operations upon the FQ are implementation-dependent. However, if an FQ is present, supervisor software must be able to deduce the exception-causing instruction's opcode (*opf*), operands, and address from its FQ entry. This must also be true of any other pending floating-point operations in the queue. See Appendix L, "Implementation Characteristics," for a discussion of the formats and operation of implemented floating-point queues.

In an implementation without an FQ, the *qne* bit in the FSR is always 0, and an STDFQ instruction causes an fp_exception trap with FSR.*ftt* = 4 (sequence_error).

4.5. CP Registers

All of the coprocessor data and control/status registers are optional and implementation-dependent.

The coprocessor working registers are accessed via load/store coprocessor and CPop1/CPop2 format instructions.

The architecture also provides instruction support for reading and writing a Coprocessor State Register (CSR) and a coprocessor deferred-trap queue (CQ).

If that a higher priority trap is not pending, and the CP is not present or PSR.EC = 0, execution of a load or store to a coprocessor register or of a coprocessor operate instruction generates a cp_disabled trap.