Project
on
Microprocessor Architecture and Instruction Set

INSTITUTE OF TECHNOLOGY,
NIRMA UNIVERSITY

ultraSPARC

SPARC V9 MICROPROCESSOR FAMILY

TABLE OF CONTENTS

ultraSPARC Microprocessor Overview

- Designer : Sun Microsystem
- Bits : 64 bit Processor
- Design : RISC
- Version : v9
- Production : 1993
- Encoding : Fixed
- Registers : 32

ultraSPARC Microprocessor Overview (Cont.)

- ***SPARC*** (Scalable Processor Architecture) is a RISC Instruction set Architecture (ISA) developed by *Sun Microsystem* and introduced in mid-1987.
- **SPARC** is a registered trademark of *SPARC International, Inc.*, an organization established in 1989 to promote the SPARC architecture, manage SPARC trademarks, and provide conformance testing. Implementations of the original 32-bit SPARC architecture were initially designed and used in Sun's Sun-4 workstation and server systems, replacing their earlier Sun-3 systems based on the Motorola 68000 family of processors. Later, SPARC processors were used in SMP and CC-NUMA servers produced by Sun Microsystems, Solbourne and Fujitsu, among others, and designed for 64-bit operation.

ultraSPARC Microprocessor Overview (Cont.)

- *SPARC International* was intended to open the SPARC architecture to make a larger ecosystem for the design, which has been licensed to several manufacturers, including Texas Instruments, Atmel, Cypress Semiconductor, and Fujitsu. As a result of SPARC International, the SPARC architecture is fully open and non-proprietary.
- In March 2006 the complete design of Sun Microsystems' UltraSPARC T1 microprocessor was released in open-source form at OpenSPARC.net and named the OpenSPARC T1. In 2007 the design of Sun's UltraSPARC T2 microprocessor was also released in open-source form as OpenSPARC.

Manufacturer Company

- ***Sun Microsystems, Inc.*** was a company that sold computers, computer components, computer software, and information technology services and that created the Java programming language, and the Network File System (NFS). Sun significantly evolved several key computing technologies, among them Unix, RISC Processors, Thin Client Computing, and virtualized computing. Sun was founded on February 24, 1982. At its height, Sun headquarters were in Santa Clara, California (part of Silicon Valley), on the former west campus of the *Agnews Developmental Center*.

Microprocessor Family Features

The Oracle SPARC Architecture 2011, includes the following principal features:

- A linear 64-bit address space with 64-bit addressing.
- 32-bit wide instructions— These are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.
- Few addressing modes— A memory address is given as either “register + register” or “register + immediate”.
- Triadic register addresses— Most computational instructions operate on two register operands or one register and a constant and place the result in a third register.
- Multiprocessor synchronization instructions— Multiple variations of atomic load-store memory operations are supported.

Microprocessor Family Features (Cont.)

- A large windowed register file— At any one instant, a program sees 8 global integer registers plus a 24-register window of a larger register file. The windowed registers can be used as a cache of procedure arguments, local values, and return addresses.
- Floating point— The architecture provides an IEEE 754-compatible floating-point instruction set, operating on a separate register file that provides 32 single-precision (32-bit), 32 double-precision (64-bit), and 16 quad-precision (128-bit) overlayed registers.
- Branch elimination instructions— Several instructions can be used to eliminate branches altogether (for example, Move on Condition i.e. JX on JNX instruction in 8085). Eliminating branches increases performance in superscalar and superpipelined implementations.
- Detailed MMU architecture— Oracle SPARC Architecture 2011 provides a blueprint for the software view of the UltraSPARC MMU (TTEs and TSBs).

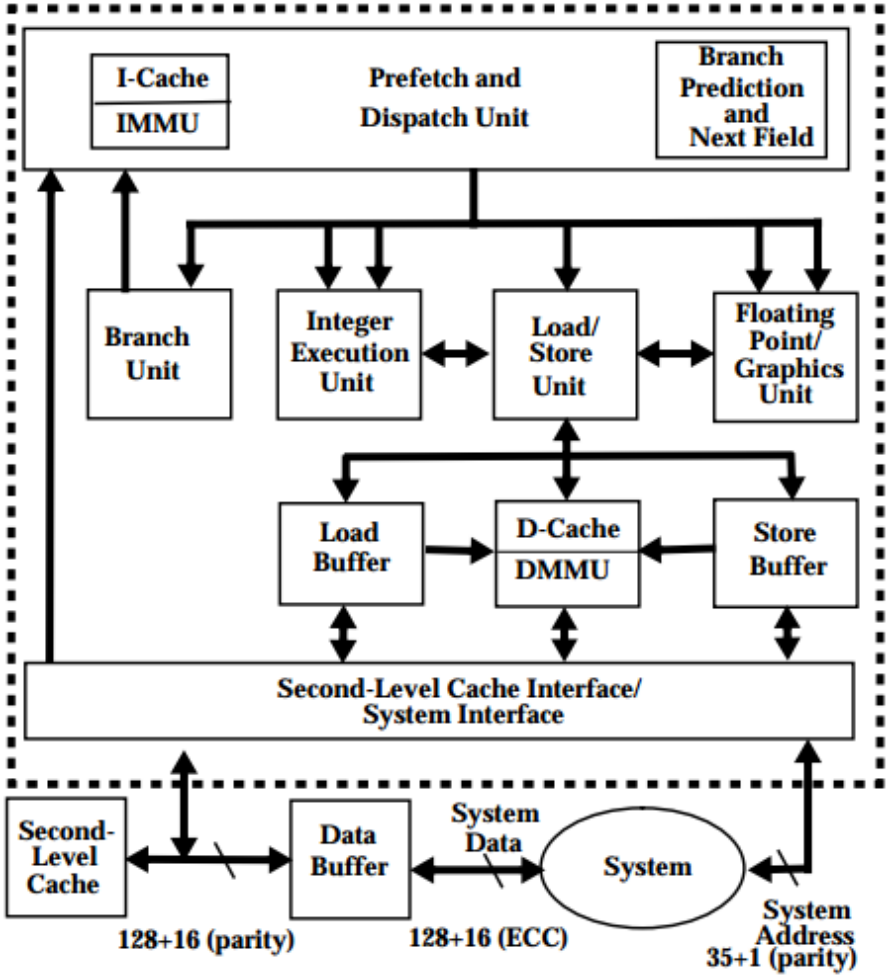
Microprocessor Family Features (Cont.)

- Fast trap handlers— Traps are vectored through a table.
- Hardware trap stack— A hardware trap stack is provided to allow nested traps. It contains all of the machine state necessary to return to the previous trap level. The trap stack makes the handling of faults and error conditions simpler, faster, and safer.
- Extended instruction set— Oracle SPARC Architecture 2011 provides many instruction set extensions, including the VIS instruction set for "vector" (SIMD) data operations.
- Hyperprivileged mode, which simplifies porting of operating systems, supports far greater portability of operating system (privileged) software, and supports the ability to run multiple simultaneous guest operating systems.

Microprocessor Architecture Overview

- An Oracle SPARC Architecture processor logically consists of an integer unit (IU) and a floating-point unit (FPU), each with its own registers. This organization allows for implementations with concurrent integer and floating-point instruction execution. Integer registers are 64 bits wide; floating-point registers are 32, 64, or 128 bits wide. Instruction operands are single registers, register pairs, register quadruples, or immediate constants.
- An Oracle SPARC Architecture virtual processor can run in non privileged mode , privileged mode, or in mode(s) of greater privilege. In privileged mode, the processor can execute non privileged and privileged instructions. In non privileged mode, the processor can only execute non privileged instructions. In non privileged or privileged mode, an attempt to execute an instruction requiring greater privilege than the current mode causes a trap.

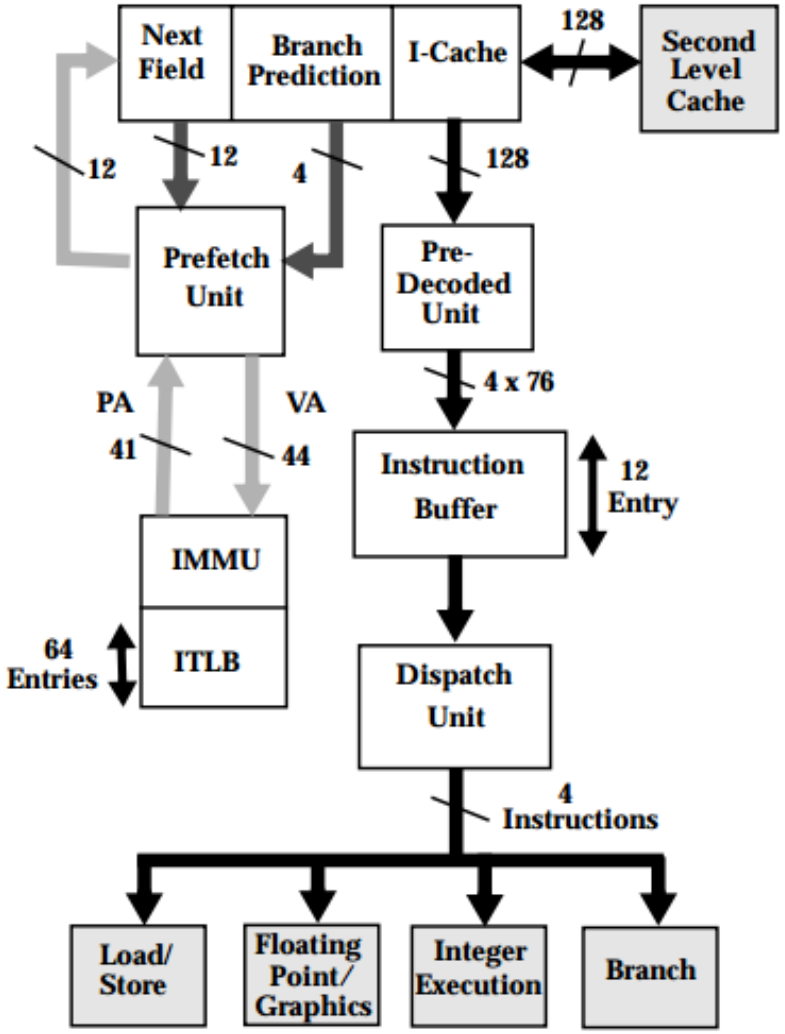
Microprocessor Architecture Overview (Cont.)



Block Diagram

ultraSPARC

Microprocessor Architecture Overview (Cont.)

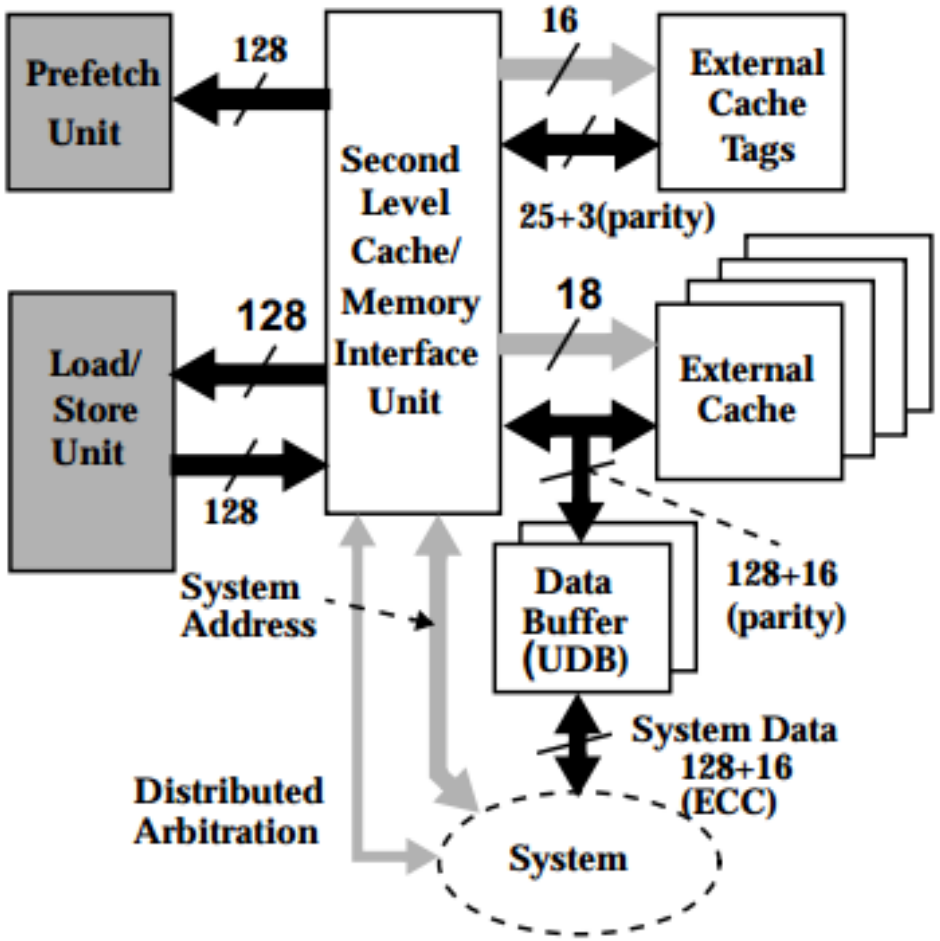


Block Diagram

ultraSPARC

front end

Microprocessor Architecture Overview (Cont.)



Block Diagram

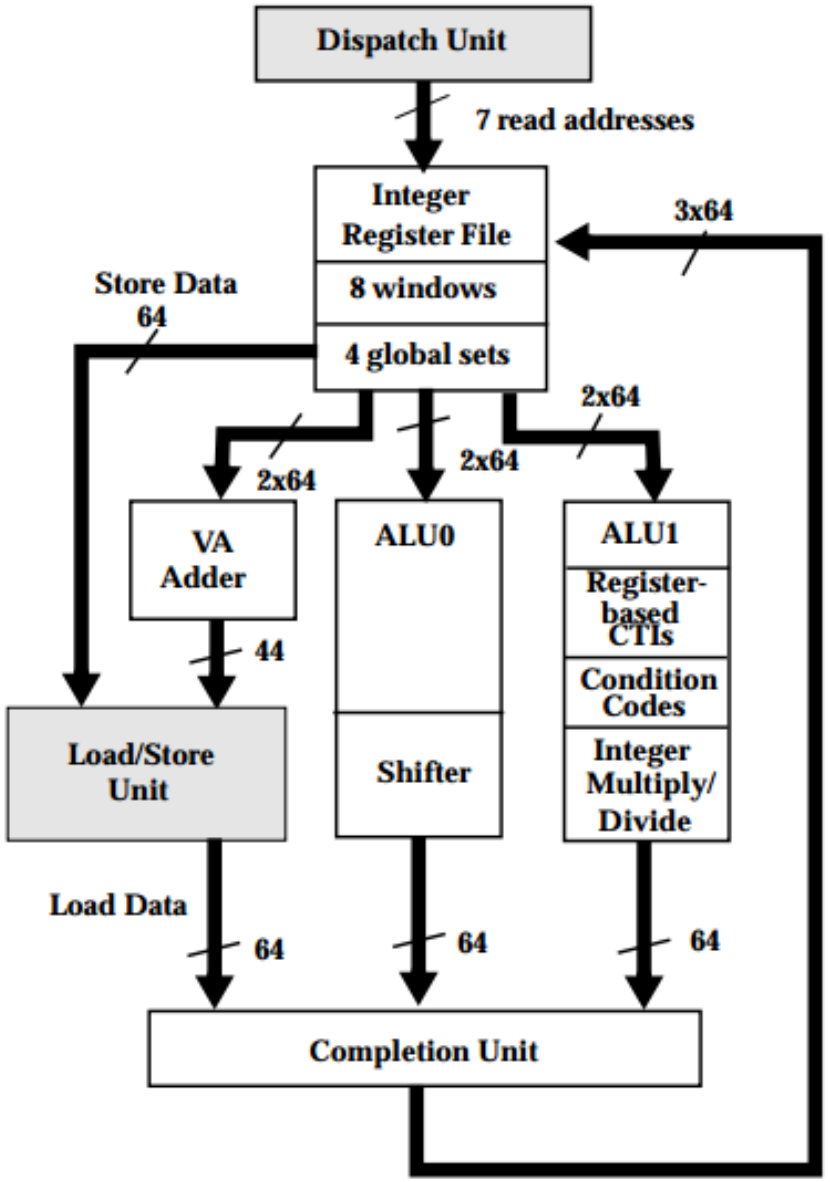
ultraSPARC

System Interface

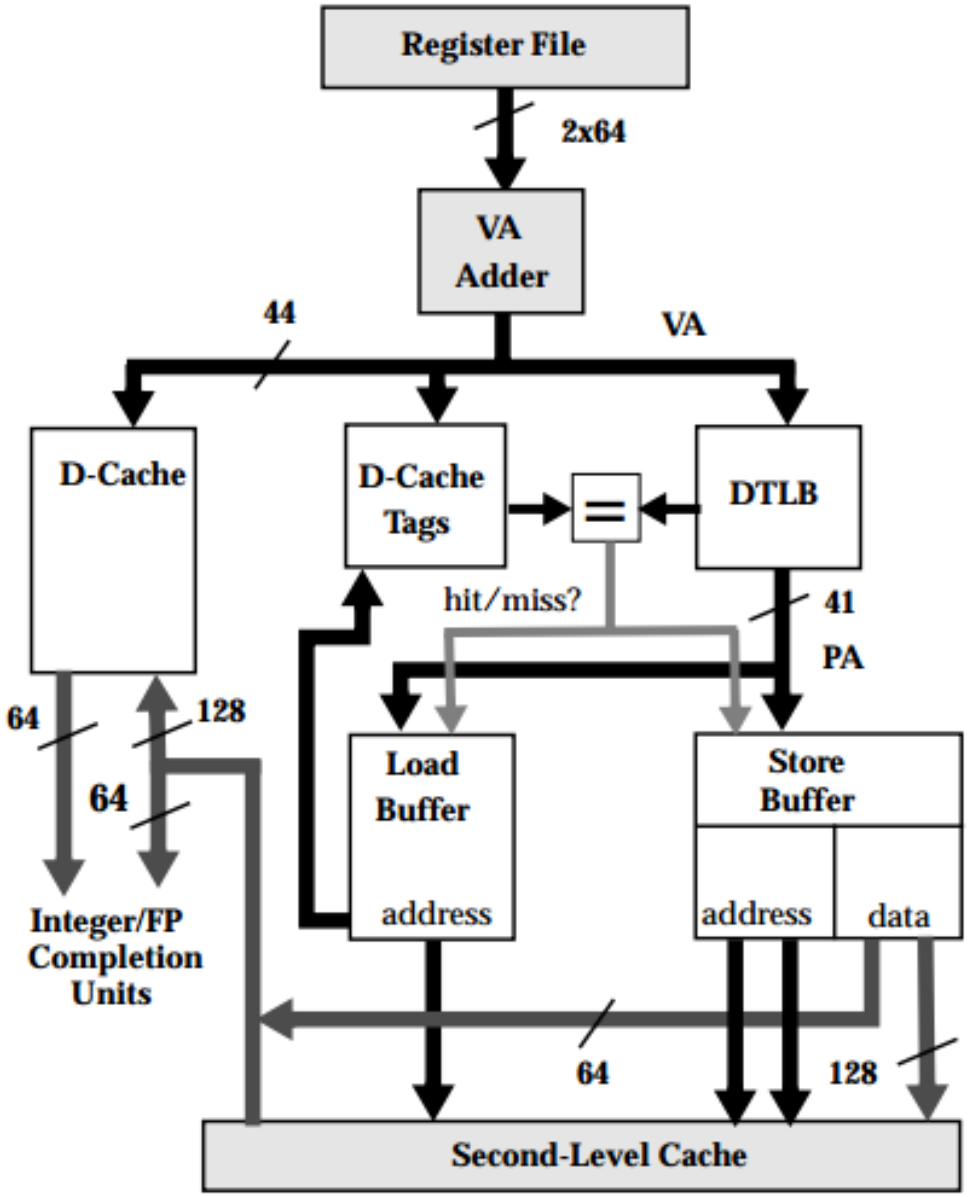
Microprocessor Architecture Overview (Cont.)

Integer Unit (IU)

- An Oracle SPARC Architecture 2011 implementation's integer unit contains the general-purpose registers and controls the overall operation of the virtual processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU.
- An Oracle SPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bitRregisters. This corresponds to a grouping of the registers intoMAXPGL+1 sets of global Registers plus a circular stack of N_REG_WINDOWS sets of 16 registers each, known as register windows. The number of register windows present (N_REG_WINDOWS) is implementation dependent, within the range of 3 to 32 (inclusive).



Integer Execution Unit

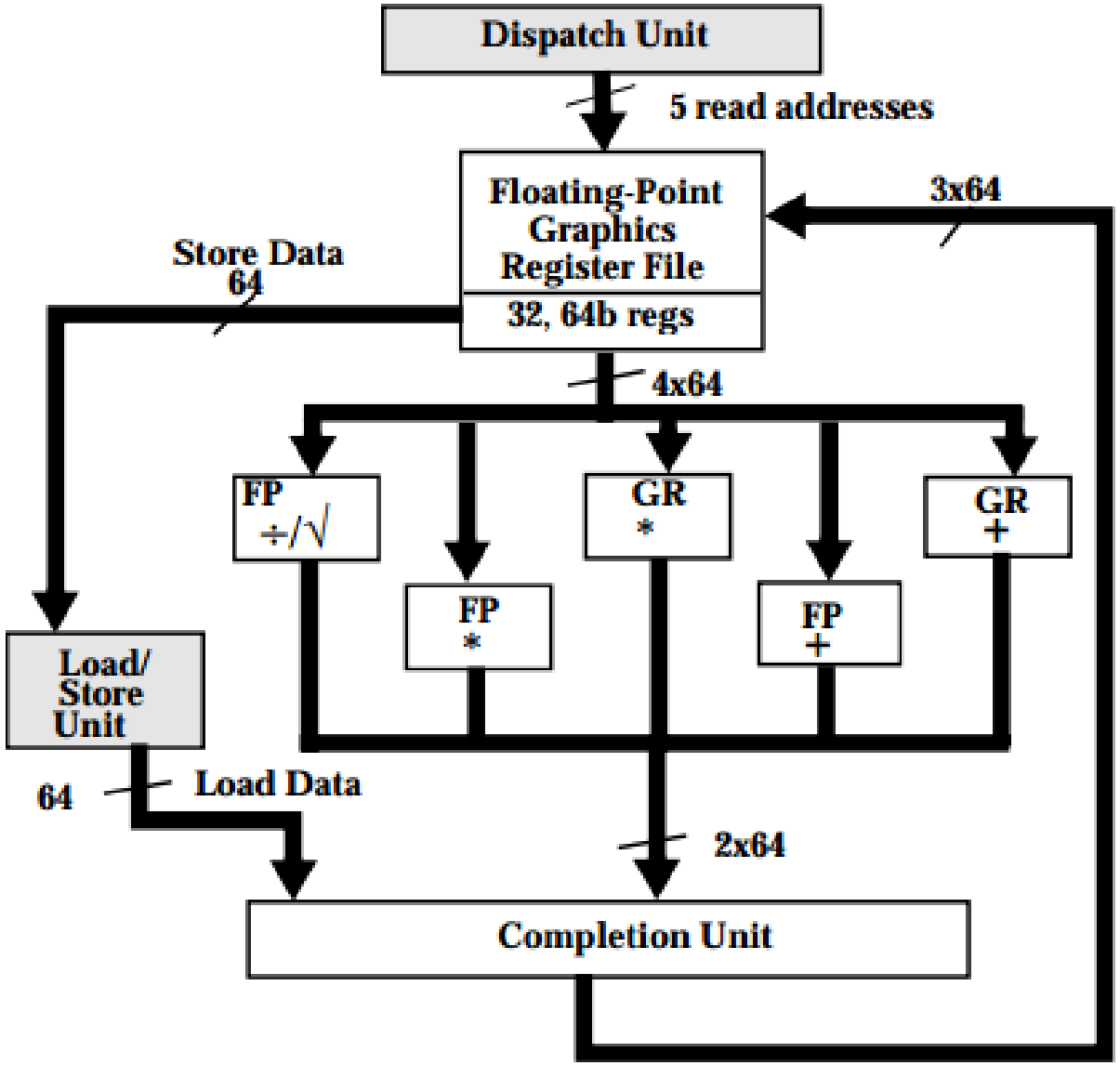


Load / Store Unit

Microprocessor Architecture Overview (Cont.)

Floating-Point Unit (FPU)

- An Oracle SPARC Architecture 2011 implementation's FPU has thirty-two 32-bit (single-precision) floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-bit (quad-precision) floating-point registers, some of which overlap.
- If no FPU is present, then it appears to software as if the FPU is permanently disabled.
- If the FPU is not enabled, then an attempt to execute a floating-point instruction generates an fp_disabled trap and the fp_disabled trap handler software must either
  - Enable the FPU (if present) and re-execute the trapping instruction, or
  - Emulate the trapping instruction in software.



Floating Point Unit

Load / Store Unit is same as the in the Integer Execution Unit.

Instruction Overview

- Instructions are fetched by the virtual processor from memory and are executed, annulled, or trapped.
- Instructions are encoded in 4 major formats and partitioned into 11 general categories.
- Instructions are described in the following sections:
  i. **Instruction Execution**
  ii. **Instruction Formats**
  iii. **Instruction Categories**

Instruction Overview (Cont.)

Instruction Execution

- The instruction at the memory location specified by the program counter is fetched and then executed. Instruction execution may change program-visible virtual processor and/or memory state. As a side effect of its execution, new values are assigned to the program counter (PC) and the next program counter (NPC).
- An instruction may generate an exception if it encounters some condition that makes it impossible to complete normal execution. Such an exception may in turn generate a precise trap. Other events may also cause traps: an exception caused by a previous instruction (a deferred trap), an interrupt or asynchronous error (a disrupting trap), or a reset request (a reset trap). If a trap occurs, control is vectored into a trap table.

Instruction Overview (Cont.)

Instruction Execution

- If a trap does not occur and the instruction is not a control transfer, the next program counter is copied into the PC, and the NPC is incremented by 4 (ignoring arithmetic overflow if any). There are two types of control-transfer instructions (CTIs): delayed and immediate. For a delayed CTI, at the end of the execution of the instruction, NPC is copied into the PC and the target address is copied into NPC. For an immediate CTI, at the end of execution, the target is copied to PC and target + 4 is copied to NPC. In the SPARC instruction set, many CTIs do not transfer control until after a delay of one instruction, hence the term "delayed CTI" (DCTI). Thus, the two program counters provide for a delayed-branch execution model.

Instruction Overview (Cont.)

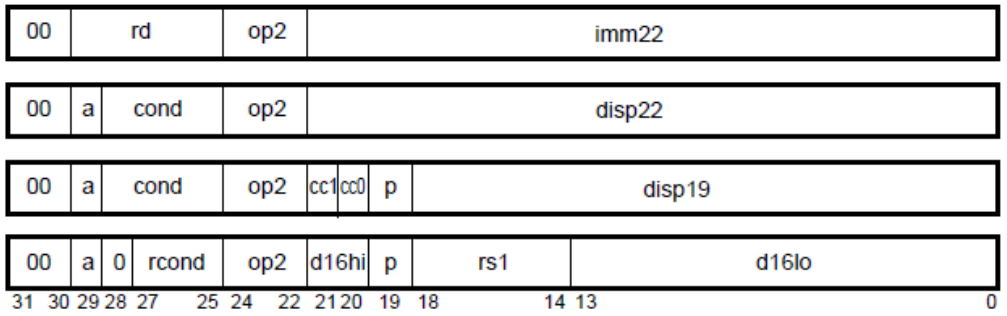
Instruction Execution

- For each instruction access and each normal data access, an 8-bit address space identifier (ASI) is appended to the 64-bit memory address. Load/store alternate instructions (*Address Space Identifiers (ASIs)*) can provide an arbitrary ASI with their data addresses or can use the ASI value currently contained in the ASI register.

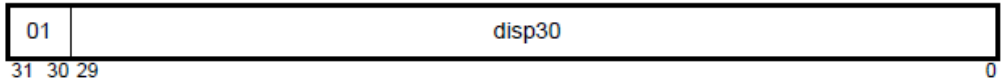
Instruction Overview (Cont.)

❖ *Instruction Formats*

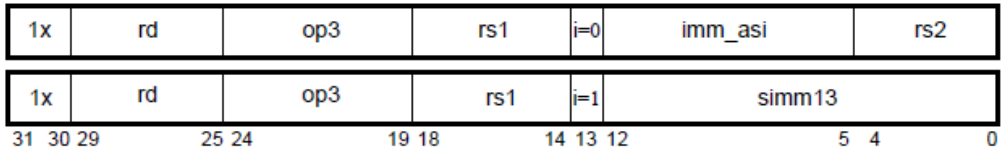
op = 00_2: *SETHI, Branches, and ILLTRAP*



op = 01_2: *CALL*



op = 10_2 *or* 11_2: *Arithmetic, Logical, Moves, Tcc, Loads, Stores, Prefetch, and Misc*



Summary of Instruction Formats

Instruction Overview (Cont.)

- *Instruction Categories*
  1) Memory access/Memory synchronization
  2) Integer arithmetic / logical / shift
  3) Control transfer/Conditional moves
  4) State register access
  5) Floating-point operate
  6) Conditional move
  7) Register window management
  8) SIMD (single instruction, multiple data) instructions

Instruction Overview (Cont.)

1. *Memory Access Instructions*

- ✓ LOAD, STORE and PREFETCH instructions are the only instructions that access memory. They use two R registers or an R register and a signed 13-bit immediate value to calculate a 64-bit, byte-aligned memory address. The Integer Unit appends an ASI(address space identifier) to this address.
- ✓ The destination field of the LOAD/STORE instruction specifies either one or two R registers or one, two or four F registers that supply the data for a store or that receive the data from a load.
- ✓ Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and extended-word(64-bit) accesses. There are versions of integer load instructions that perform either sign-extension or zero-extension on 8-bit, 16-bit, and 32-bit values as they are loaded into a 64-bit destination register. Floating-point load and store instructions support word, doubleword, and quadword memory accesses.

Addressing Conventions

Term	Definition
byte	A load/store byte instruction accesses the addressed byte in both big- and little-endian modes.
halfword	For a load/store halfword instruction, two bytes are accessed. The most significant byte (bits 15–8) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 1.
word	For a load/store word instruction, four bytes are accessed. The most significant byte (bits 31–24) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 3.
doubleword or extended word	For a load/store extended or floating-point load/store double instruction, eight bytes are accessed. The most significant byte (bits 63:56) is accessed at the address specified in the instruction; the least significant byte (bits 7:0) is accessed at the address + 7. For the deprecated integer load/store twin word instructions (LDTW, LDTWA†, STTW, STTWA), two big-endian words are accessed. The word at the address specified in the instruction corresponds to the even register specified in the instruction; the word at address + 4 corresponds to the following odd-numbered register. †Note that the LDTXA instruction, which is not an LDTWA operation but does share LDTWA's opcode, is ***not*** deprecated.
quadword	For a load/store quadword instruction, 16 bytes are accessed. The most significant byte (bits 127–120) is accessed at the address specified in the instruction; the least significant byte (bits 7–0) is accessed at the address + 15.

Big-endian Addressing Convention. Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte's significance decreases as its address increases. The big-endian addressing conventions are described in TABLE.

Byte Address

7	0

Halfword Address{0} =

0		1	
15	8	7	0

Word Address{1:0} =

00		01		10		11	
31	24	23	16	15	8	7	0

Doubleword / Extended word Address{2:0} =

000		001		010		011	
63	56	55	48	47	40	39	32

Address{2:0} =

100		101		110		111	
31	24	23	16	15	8	7	0

Quadword Address{3:0} =

0000		0001		0010		0011	
127	120	119	112	111	104	103	96

Address{3:0} =

0100		0101		0110		0111	
95	88	87	80	79	72	71	64

Address{3:0} =

1000		1001		1010		1011	
63	56	55	48	47	40	39	32

Address{3:0} =

1100		1101		1110		1111	
31	24	23	16	15	8	7	0

Big-endian Addressing Conventions

Instruction Overview (Cont.)

2. *Integer Arithmetic / Logical / Shift Instructions*

- ✓ The arithmetic/logical/shift instructions perform arithmetic, tagged arithmetic, logical and shift operations. With one exception, these instructions compute a result that is a function of two source operands; the result is either written into a destination register or discarded.
  - ‼ Exception : SETHI(Set High 22 Bits of Low Word) can be used in combination with other arithmetic and/or logical instructions to create a constant in an R register.
- ✓ Shift instructions shift the contents of an R register left or right by a given number of bits (“shift count”). The shift distance is specified by a constant in the instruction or by the contents of an R register.

Instruction Overview (Cont.)

3. *Control Transfer Instructions*

- ✓ Control-transfer instructions (CTIs) include PC-relative branches and calls, register-indirect jumps and conditional traps.
- ✓ Most of the control-transfer instructions are delayed; that is, the instruction immediately following a control-transfer instruction in logical sequence is dispatched before the control transfer to the target address is completed. Note that the next instruction in logical sequence may not be the instruction following the control-transfer instruction in memory.
- ✓ Branch and CALL instructions use PC-relative displacements. The jump and link (JMPL) and return(RETURN) instructions use a register-indirect target address. They compute their target addresses either as the sum of two R registers or as the sum of an R register and a 13-bit signed immediate value.

Control-Transfer Characteristics

Instruction Group	Address Form	Delayed?	Taken?	Annul Bit?	New PC	New NPC
Non-CTIs	—	—	—	—	NPC	NPC + 4
Bcc	PC-relative	Yes	Yes	0	NPC	EA
Bcc	PC-relative	Yes	No	0	NPC	NPC + 4
Bcc	PC-relative	Yes	Yes	1	NPC	EA
Bcc	PC-relative	Yes	No	1	NPC + 4	NPC + 8
BA	PC-relative	Yes	Yes	0	NPC	EA
BA	PC-relative	No	Yes	1	EA	EA + 4
BN	PC-relative	Yes	No	0	NPC	NPC + 4
BN	PC-relative	Yes	No	1	NPC + 4	NPC + 8
CALL	PC-relative	Yes	—	—	NPC	EA
C*Bcond	PC-relative	No	Yes	—	EA	EA + 4
C*Bcond	PC-relative	No	No	—	NPC	NPC + 4
JMPL, RETURN	Register-indirect	Yes	—	—	NPC	EA
DONE	Trap state	No	—	—	TNPC[TL]	TNPC[TL] + 4
RETRY	Trap state	No	—	—	TPC[TL]	TNPC[TL]
Tcc	Trap vector	No	Yes	—	EA	EA + 4
Tcc	Trap vector	No	No	—	NPC	NPC + 4

The effective address, “EA” in TABLE, specifies the target of the control-transfer instruction. The effective address is computed in different ways, depending on the particular instruction.

Instruction Overview (Cont.)

- ✓ **PC-relative effective address —** A PC-relative effective address is computed by sign extending the instruction's immediate field to 64-bits, left-shifting the word displacement by 2 bits to create a byte displacement, and adding the result to the contents of the PC.

- ✓ **Register-indirect effective address —** If i = 0, a register-indirect effective target address is R[rs1] + R[rs2]. If i = 1, a register-indirect effective target address is R[rs1] + **sign_ext** *(simm13*).

- ✓ **Trap vector effective address —** A trap vector effective address first computes the software trap number as the least significant 7 or 8 bits of R[rs1] + R[rs2] if i = 0, or as the least significant 7 or 8 bits of R[rs1] + *imm_trap#* if i = 1. Whether 7 or 8 bits are used depends on the privilege level — 7 bits are used in non privileged mode and 8 bits are used in privileged mode. The trap level, TL, is incremented. The hardware trap type is computed as 256 + the software trap number and stored in TT[TL]. The effective address is generated by combining the contents of the TBA register with the trap type and other data; see *Trap Processing* on page 428 for details.

- ✓ **Trap state effective address —** A trap state effective address is not computed but is taken directly from either TPC[TL] or TNPC[TL].

Instruction Overview (Cont.)

4. *State Register Access Instructions*

- ✓ Ancillary State Registers
  - The read and write ancillary state register instructions read and write the contents of ancillary state registers visible to non privileged software (Y, CCR, ASI, PC, TICK, and FPRS) and some registers visible only to privileged software (SOFTINT and STICK_CMPR).
- ✓ PR State Registers
  - The read and write privileged register instructions (RDPR and WRPR) read and write the contents of state registers visible only to privileged software (TPC, TNPC, TSTATE, TT, TICK, TBA, PSTATE, TL, PIL, CWP, CANSAVE, CANRESTORE, CLEANWIN, OTHERWIN, and WSTATE).

Instruction Overview (Cont.)

5. *Floating-Point Operate Instructions*
   - ✓ Floating-point operate (FPop) instructions perform all floating-point calculations; they are register to register instructions that operate on the floating-point registers.
   - ✓ Floating-point operate instructions (FPops) compute a result that is a function of one , two, or three source operands and place the result in one or more destination F registers.
   - ✓ With one exception: floating-point compare operations do not write to an F register but instead update one of the fcc*n* fields of the FSR.

Instruction Overview (Cont.)

6. *Conditional Move Instructions*
   - ✓ Conditional move instructions conditionally copy a value from a source register to a destination register, depending on an integer or floating-point condition code or on the contents of an integer register. These instructions can be used to reduce the number of branches in software.
   - ✓ Conditional move instructions also conditionally move a value from a source register to a destination register.

MOVr and FMOVr Test Conditions

Condition	Description
NZ	Nonzero
Z	Zero
GEZ	Greater than or equal to zero
LZ	Less than zero
LEZ	Less than or equal to zero
GZ	Greater than zero

The MOVr and FMOVr instructions allow the contents of any integer or floating-point register to be moved to a destination integer or floating-point register if the contents of a register satisfy a specified condition. The conditions to test are enumerated in TABLE.

Instruction Overview (Cont.)

7. *Register Window Management Instructions*
   - ✓ Register window instructions manage the register windows.
   - ✓ The state of the register windows is determined by the contents of the set of privileged registers. Those registers are affected by the Register Window Management Instructions.
   - ✓ SAVE and RESTORE are non privileged and cause a register window to be pushed or popped. FLUSHW is non privileged and causes all of the windows except the current one to be flushed to memory.
   - ✓ SAVED and RESTORED are used by privileged software to end a window spill or fill trap handler.

Instruction Overview (Cont.)

8. *SIMD Instructions*

- ✓ Oracle SPARC Architecture 2011 includes SIMD (single instruction, multiple data) instructions, also known as "vector" instructions, which allow a single instruction to perform the same operation on multiple data items, totalling 64 bits, such as eight 8-bit, four 16-bit, or two 32-bit data items. These operations are part of the “VIS” extensions.

Data Formats

- The Oracle SPARC Architecture recognizes these fundamental data types:
  - ✓Signed integer: 8, 16, 32, and 64 bits
  - ✓Unsigned integer: 8, 16, 32, and 64 bits
  - ✓SIMD data formats: Uint8 SIMD (32 bits), Int16 SIMD (64 bits), and Int32 SIMD (64 bits)
  - ✓Floating point: 32, 64, and 128 bits

Data Format (Cont.)

- The signed integer values are stored as two's-complement numbers with a width commensurate with their range.
- Unsigned integer values, bit vectors, Boolean values, character strings, and other values representable in binary form are stored as unsigned integers with a width commensurate with their range.
- The floating-point formats conform to the IEEE Standard for Binary Floating-point Arithmetic, IEEE STD. 754-1985.
- In tagged words, the least significant two bits are treated as a tag; the remaining 30 bits are treated as a signed integer.

Data Format (Cont.)

- Data formats are described in these sections:

1) Integer Data Formats
2) Floating-Point Data Formats
3) SIMD Data Formats

Integer Data Format

Signed Integer, Unsigned Integer, and Tagged Format Ranges

Data Type	Width (bits)	Range
Signed integer byte	8	-2^7 to $2^7 - 1$
Signed integer halfword	16	-2^{15} to $2^{15} - 1$
Signed integer word	32	-2^{31} to $2^{31} - 1$
Signed integer doubleword/extended-word	64	-2^{63} to $2^{63} - 1$
Unsigned integer byte	8	0 to $2^8 - 1$
Unsigned integer halfword	16	0 to $2^{16} - 1$
Unsigned integer word	32	0 to $2^{32} - 1$
Unsigned integer doubleword/extended-word	64	0 to $2^{64} - 1$
Integer tagged word	32	0 to $2^{30} - 1$

TABLE describes the width and ranges of the signed, unsigned, and tagged integer data formats.

Integer Data Format (Cont.)

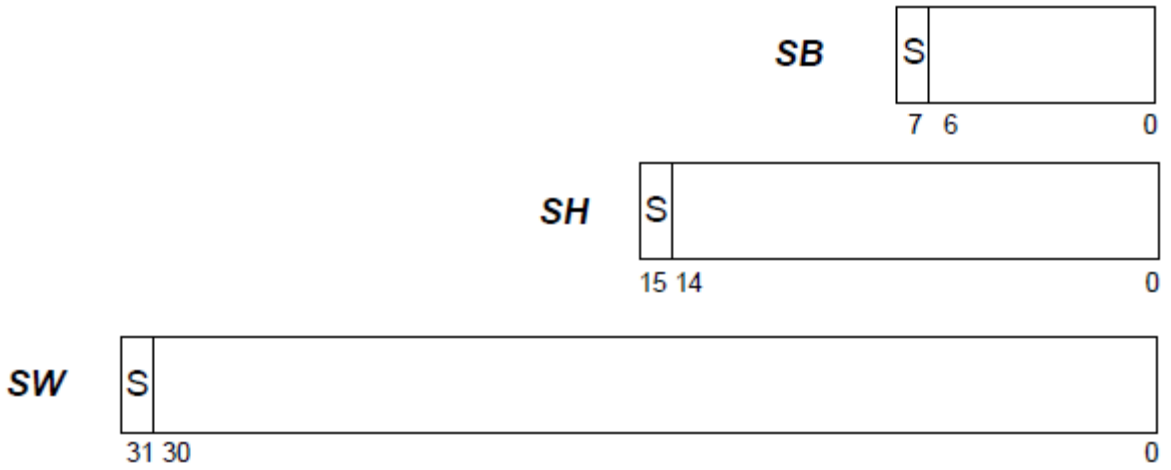
Subformat Name	Subformat Field	Memory Address		Register Number	
		Required Alignment	Address (big-endian)[1]	Required Alignment	Register Number
SD-0	signed_dbl_integer{63:32}	n mod 8 = 0	n	r mod 2 = 0	r
SD-1	signed_dbl_integer{31:0}	$(n + 4)$ mod 8 = 4	$n + 4$	$(r + 1)$ mod 2 = 1	$r + 1$
SX	signed_ext_integer{63:0}	n mod 8 = 0	n	—	r
UD-0	unsigned_dbl_integer{63:32}	n mod 8 = 0	n	r mod 2 = 0	r
UD-1	unsigned_dbl_integer{31:0}	$(n + 4)$ mod 8 = 4	$n + 4$	$(r + 1)$ mod 2 = 1	$r + 1$
UX	unsigned_ext_integer{63:0}	n mod 8 = 0	n	—	r

TABLE describes the memory and register alignment for multiword integer data. All registers in the integer register file are 64 bits wide, but can be used to contain smaller (narrower) data sizes. Note that there is no difference between integer extended-words and doublewords in memory; the only difference is how they are represented in registers.

Integer Data Format (Cont.)

1. Signed Integer Data Type

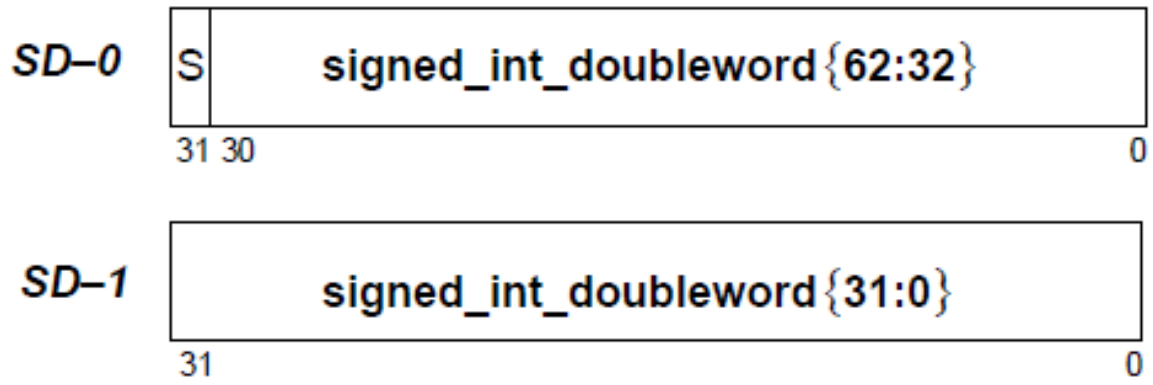
Signed Integer Byte, Halfword, and Word



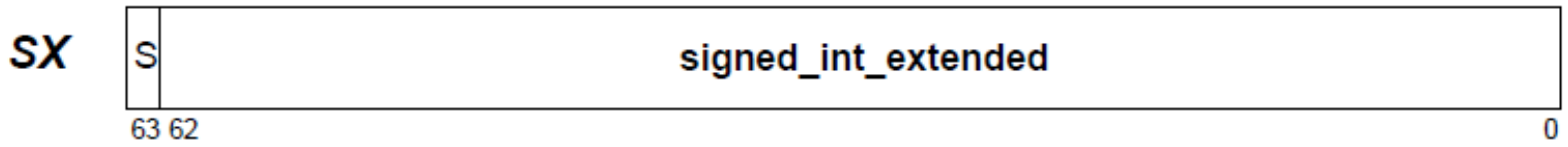
Signed Integer Byte, Halfword, and Word Data Formats

Integer Data Format (Cont.)

Signed Integer Doubleword (64 bits)



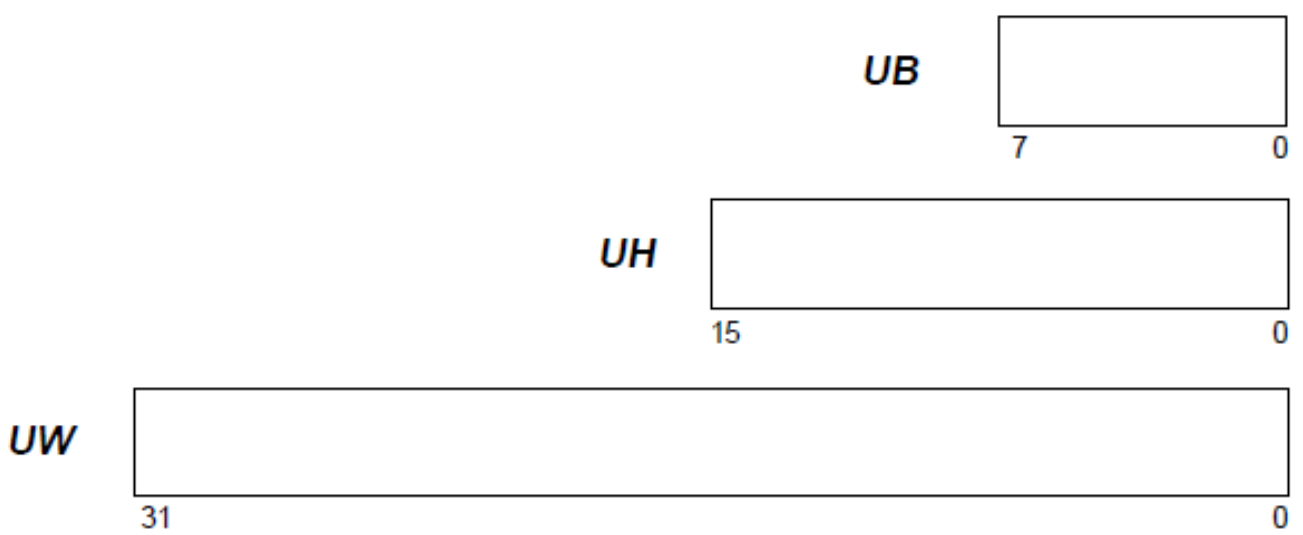
Signed Integer Extended-Word (64 bits)



Integer Data Format (Cont.)

2. Unsigned Integer Data Type

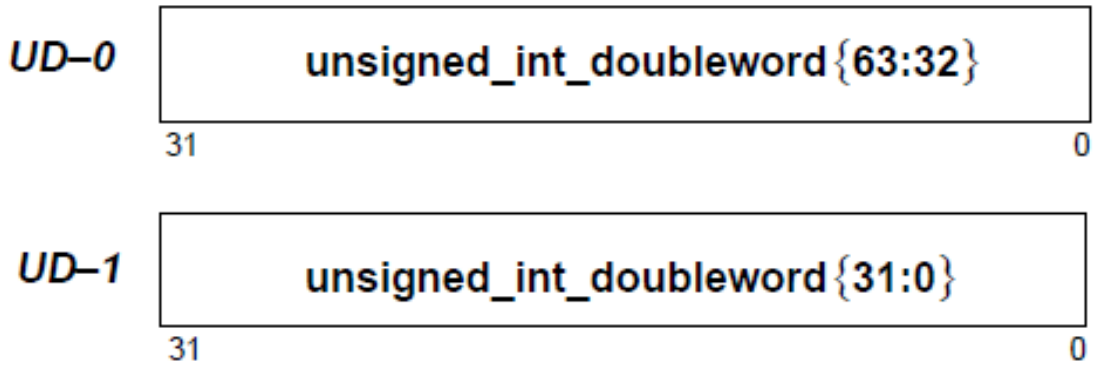
Unsigned Integer Byte, Halfword, and Word



Unsigned Integer Byte, Halfword, and Word Data Formats

Integer Data Format (Cont.)

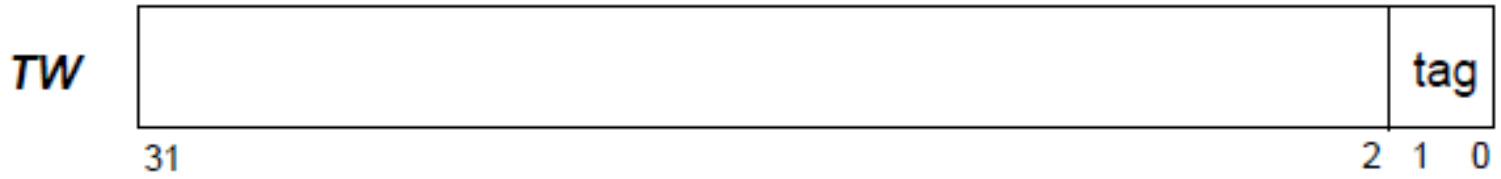
Unsigned Integer Doubleword (64 bits)



Unsigned Integer Extended-Word (64 bits)

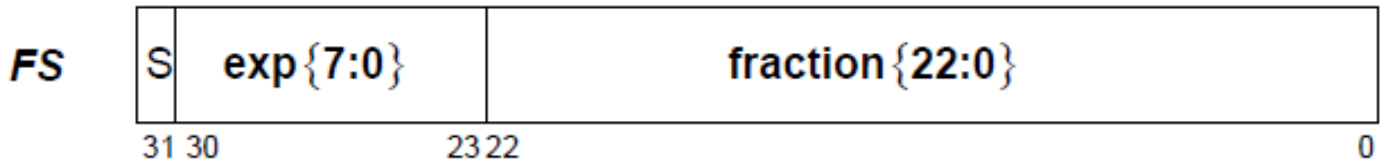
Integer Data Format (Cont.)

3. Tagged Word

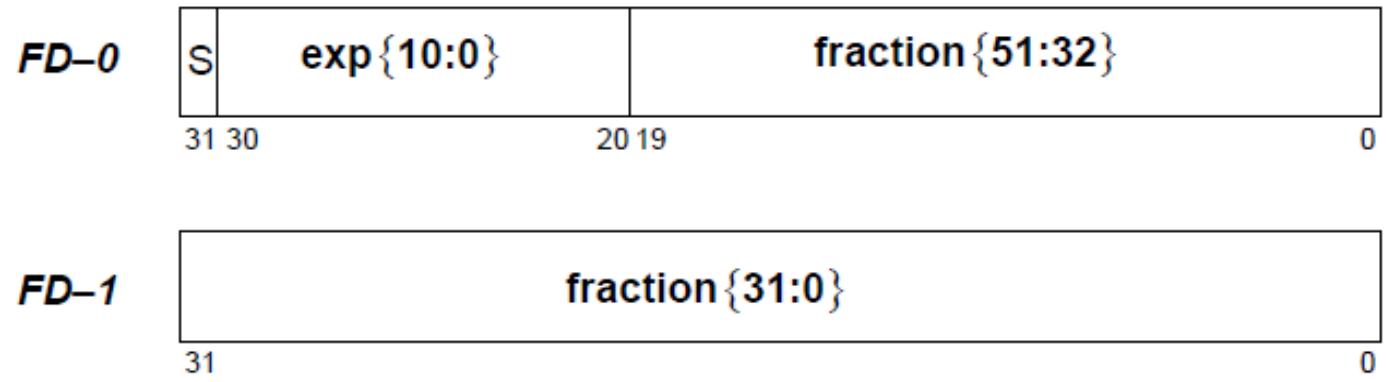


Floating-Point Data Formats

1. Floating Point, Single Precision (32 bits)

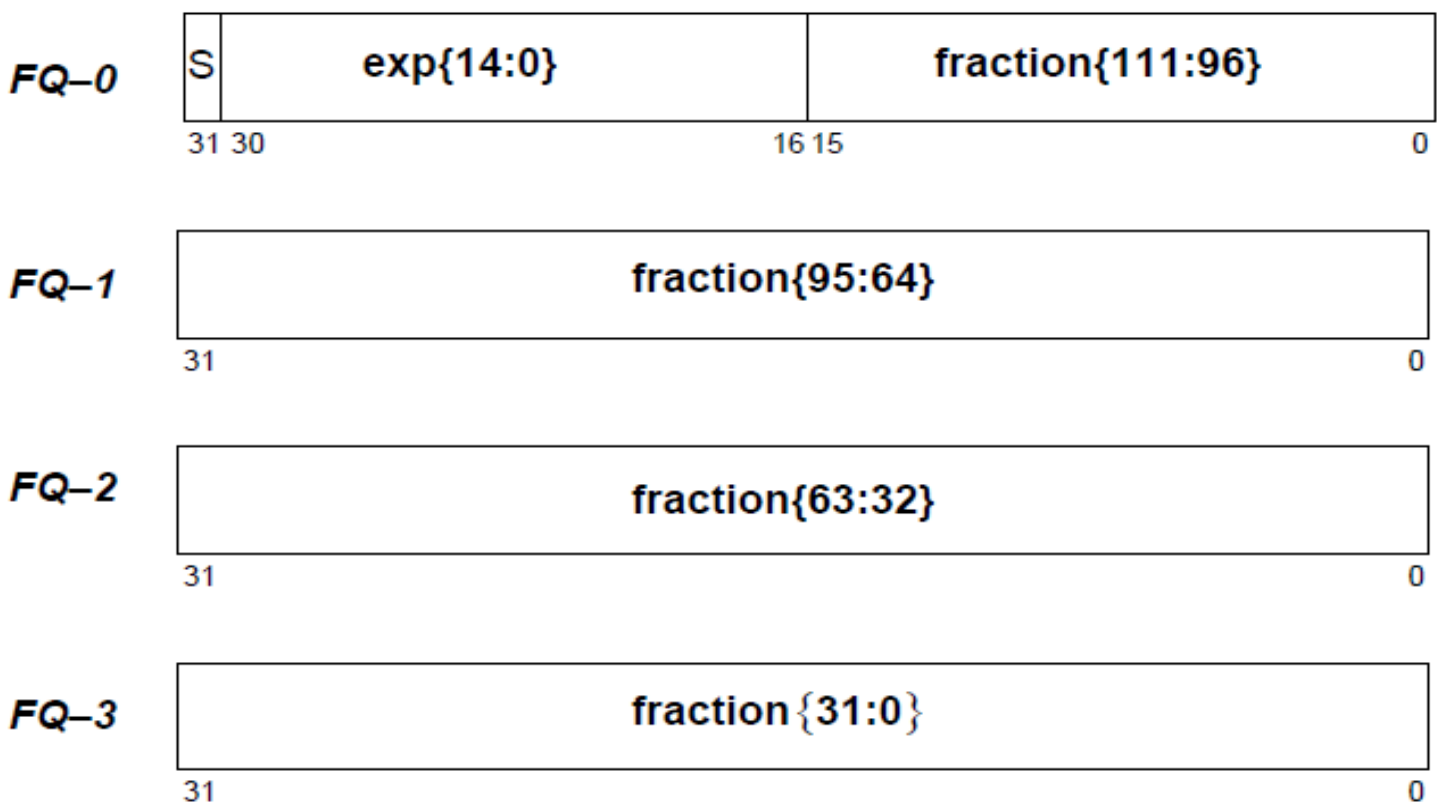


2. Floating Point, Double Precision (64 bits)



Floating-Point Data Formats (Cont.)

3. Floating Point, Quad Precision (128 bits)



Floating-Point Data Formats (Cont.)

Floating-Point Single-Precision Format Definition

s = sign (1 bit)
e = biased exponent (8 bits)
f = fraction (23 bits)
u = undefined

Normalized value ($0 < e < 255$):	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal value ($e = 0$):	$(-1)^s \times 2^{-126} \times 0.f$
Zero ($e = 0, f = 0$)	$(-1)^s \times 0$
Signalling NaN	s = u; e = 255 (max); f = .0uu--uu (At least one bit of the fraction must be nonzero)
Quiet NaN	s = u; e = 255 (max); f = .1uu--uu
$-\infty$ (negative infinity)	s = 1; e = 255 (max); f = .000--00
$+\infty$ (positive infinity)	s = 0; e = 255 (max); f = .000--00

Floating-Point Single-Precision Format Definition

Floating-Point Data Formats (Cont.)

Floating-Point Double-Precision Format Definition

s = sign (1 bit)
e = biased exponent (11 bits)
f = fraction (52 bits)
u = undefined

Normalized value ($0 < e < 2047$):	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal value ($e = 0$):	$(-1)^s \times 2^{-1022} \times 0.f$
Zero ($e = 0$, $f = 0$)	$(-1)^s \times 0$
Signalling NaN	s = u; e = 2047 (max); f = .0uu--uu (At least one bit of the fraction must be nonzero)
Quiet NaN	s = u; e = 2047 (max); f = .1uu--uu
$-\infty$ (negative infinity)	s = 1; e = 2047 (max); f = .000--00
$+\infty$ (positive infinity)	s = 0; e = 2047 (max); f = .000--00

Floating-Point Double-Precision Format Definition

Floating-Point Data Formats (Cont.)

Floating-Point Quad-Precision Format Definition

s = sign (1 bit)
e = biased exponent (15 bits)
f = fraction (112 bits)
u = undefined

Normalized value ($0 < e < 32767$):	$(-1)^s \times 2^{e-16383} \times 1.f$
Subnormal value ($e = 0$):	$(-1)^s \times 2^{-16382} \times 0.f$
Zero ($e = 0$, $f = 0$)	$(-1)^s \times 0$
Signalling NaN	s = u; e = 32767 (max); f = .0uu--uu (At least one bit of the fraction must be nonzero)
Quiet NaN	s = u; e = 32767 (max); f = .1uu--uu
$-\infty$ (negative infinity)	s = 1; e = 32767 (max); f = .000--00
$+\infty$ (positive infinity)	s = 0; e = 32767 (max); f = .000--00

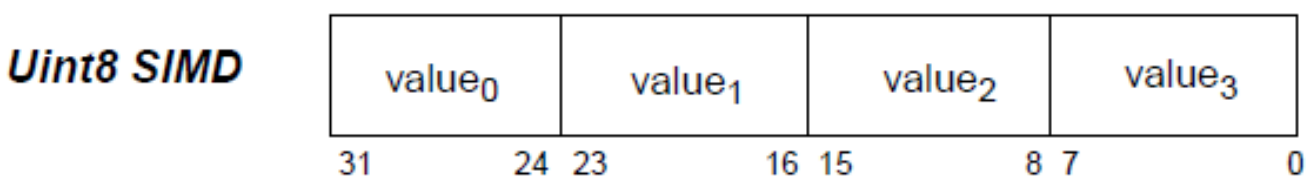
Floating-Point Quad-Precision Format Definition

SIMD Data Formats

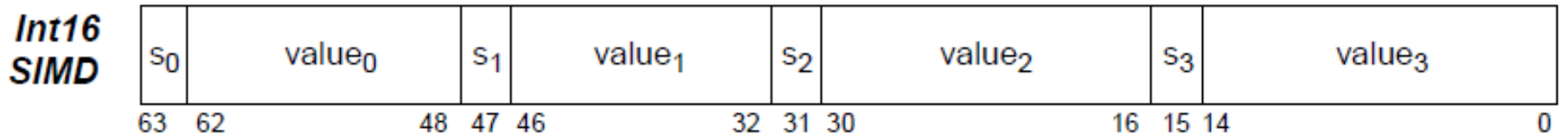
- SIMD (single instruction/multiple data) instructions perform identical operations on multiple data contained ("packed") in each source operand. This section describes the data formats used by SIMD instructions.
- Conversion between the different SIMD data formats can be achieved through SIMD multiplication or by the use of the SIMD data formatting instructions.
- *Programming Note*
  - The SIMD data formats can be used in graphics calculations to represent intensity values for an image (e.g., α, B, G, R).
  - Intensity values are typically grouped in one of two ways, when using SIMD data formats:
    - Band interleaved images, with the various color components of a point in the image stored together, and
    - Band sequential images, with all of the values for one color component stored together.

SIMD Data Formats (Cont.)

1. Uint8 SIMD Data Format
   - The Uint8 SIMD data format consists of four unsigned 8-bit integers contained in a 32-bit word



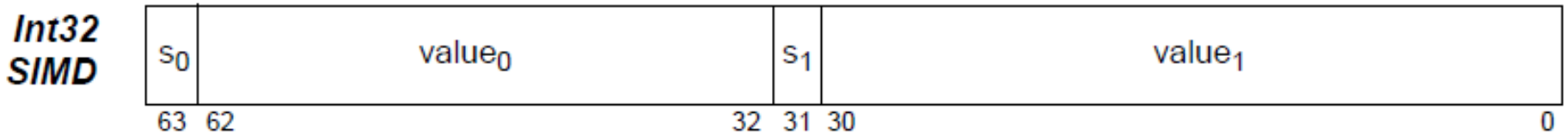
2. Int16 SIMD Data Formats
   - The Int16 SIMD data format consists of four signed 16-bit integers contained in a 64-bit word



SIMD Data Formats (Cont.)

3. Int32 SIMD Data Format

- The Int32 SIMD data format consists of two signed 32-bit integers contained in a 64-bit word



Programming Note:

The integer SIMD data formats can be used to hold fixed-point data. The position of the binary point in a SIMD datum is implied by the programmer and does not influence the computations performed by instructions that operate on that SIMD data format.

Registers

- General-Purpose (**R**)Registers
- Floating-Point Registers
- Floating-Point State Register (**FSR**)
- Ancillary State Registers
  - The following registers are included in this category:
    i. 32-bit Multiply/Divide Register (**Y**)
    ii. Integer Condition Codes Register (**CCR**)
    iii. Address Space Identifier (**ASI**) Register
    iv. Tick (**TICK**) Register
    v. Program Counters (**PC, NPC**)
    vi. Floating-Point Registers State (**FPRS**) Register
    vii. General Status Register (**GSR**)
    viii. System Tick (**STICK**) Register

Register (Cont.)

- Register-Window PR State Registers
  - The following registers are included in this subcategory:
    i. Current Window Pointer (**CWPP**) Register
    ii. Savable Windows (**CANSAVEP**) Register
    iii. Restorable Windows (**CANRESTOREP**) Register
    iv. Clean Windows (**CLEANWINP**) Register
    v. Other Windows (**OTHERWINP**) Register
    vi. Window State (**WSTATEP**) Register

Register (Cont.)

- Non-Register-Window PR State Registers
  - The following registers are included in this subcategory:
    i. Trap Program Counter (**TPCP**) Register
    ii. Trap Next PC (**TNPCP**) Register
    iii. Trap State (**TSTATEP**) Register
    iv. Trap Type (**TTP**) Register
    v. Trap Base Address (**TBAP**) Register
    vi. Processor State (**PSTATEP**) Register
    vii. Trap Level Register (**TLP**)
    viii. Processor Interrupt Level (**PILP**) Register
    ix. Global Level Register (**GLP**)

General-Purpose R Registers

- An Oracle SPARC Architecture virtual processor contains an array of general-purpose 64-bit **R** registers. The array is partitioned into MAXPGL + 1 (**Maximum Global Level)** sets of eight global registers, plus N_REG_WINDOWS groups of 16 registers each. The value of N_REG_WINDOWS in SPARC Architecture implementation falls within the range 3 to 32 (inclusive).
- One set of 8 global registers is always visible. At any given time, a group of 24 registers, known as a register window, is also visible. A register window comprises the 16 registers from the current 16-register group (referred to as 8 in registers and 8 local registers), plus half of the registers from the next 16-register group (referred to as 8 out registers).
- SPARC instructions use 5-bit fields to reference R registers. That is, 32 R registers are visible to software at any moment.

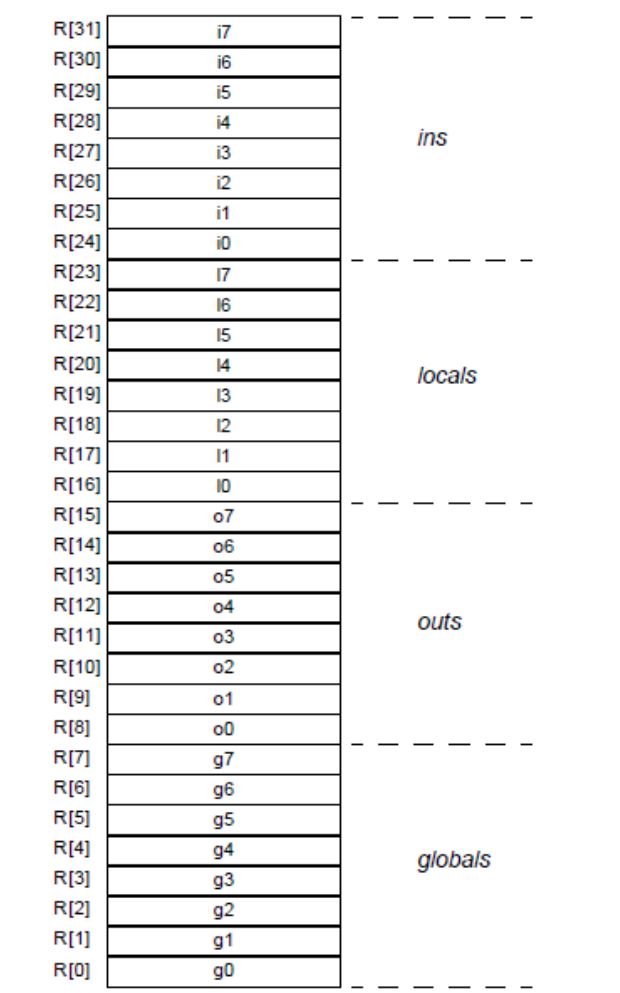
General-Purpose R Registers (Cont.)

- The following types of Registers are included in this section:
  - Global R Registers
  - Windowed R Registers
    I. IN
    II. OUT
    III. LOCAL

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]

General-Purpose R Registers (Cont.)

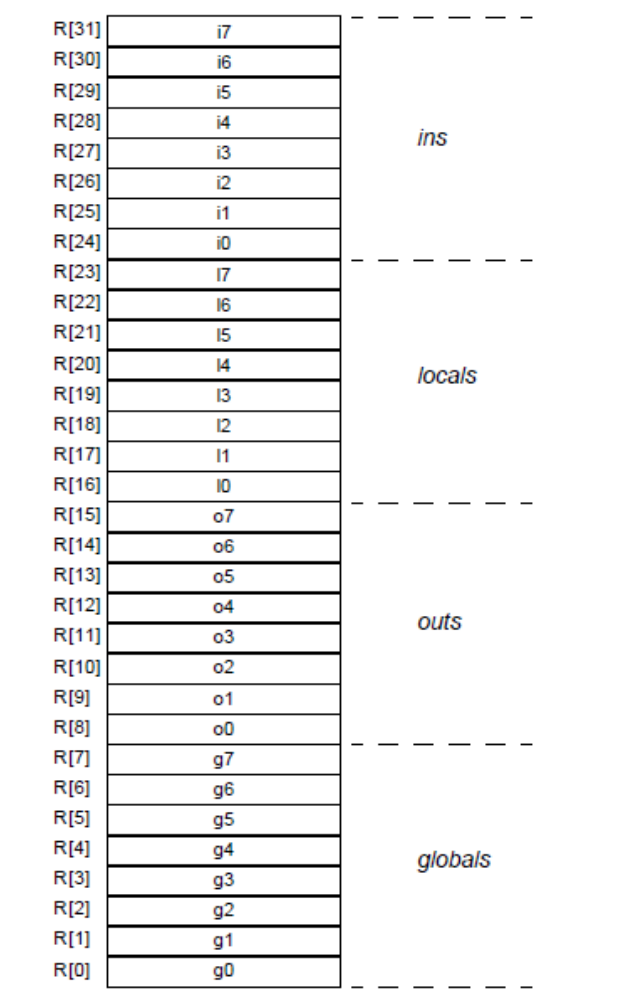


General-Purpose Registers (as Visible at Any Given Time)

Global R Registers

- ✓ Registers R[0]–R[7] refer to a set of eight registers called the *global* registers (labelled g0 through g7).
- ✓ At any time, one of *MAXPGL* +1 sets of eight registers is enabled and can be accessed as the current set of global registers.
- ✓ The currently enabled set of global registers is selected by the GL register(*Global Level Register).*

General-Purpose R Registers (Cont.)



General-Purpose Registers (as Visible at Any Given Time)

Windowed R Registers

- ✓ A set of 24 R registers that is visible as R[8]–R[31] at any given time is called a "register window".
- ✓ The registers that become R[8]–R[15] in a register window are called the *out* registers of the window.
- ✓ Note that the *in* registers of a register window become the *out* registers of an adjacent register window.
- ✓ The names *in*, *local*, and *out* originate from the fact that the *out* registers are typically used to pass parameters from (out of) a calling routine and that the called routine receives those parameters as its *in* registers.

Floating-Point Registers

- The floating-point register set consists of sixty-four 32-bit registers, which may be accessed as follows:
  i. Sixteen 128-bit quad-precision registers, referenced as FQ[0], FQ[4], …, FQ[60]
  ii. Thirty-two 64-bit double-precision registers, referenced as FD[0], FD[2], …, FD[62]
  iii. Thirty-two 32-bit single-precision registers, referenced as FS[0], FS[1], …, FS[31] (only the lower half of the floating-point register file can be accessed as single-precision registers)

- The floating-point registers are arranged so that some of them overlap, that is, are aliased. Unlike the windowed **R** registers, all of the floating-point registers are accessible at any time. The floating-point registers can be read and written by floating-point operate (FPop1/FPop2 format) instructions, by load/store single/double/quad floating-point instructions, by VIS instructions, and by block load and block store instructions.

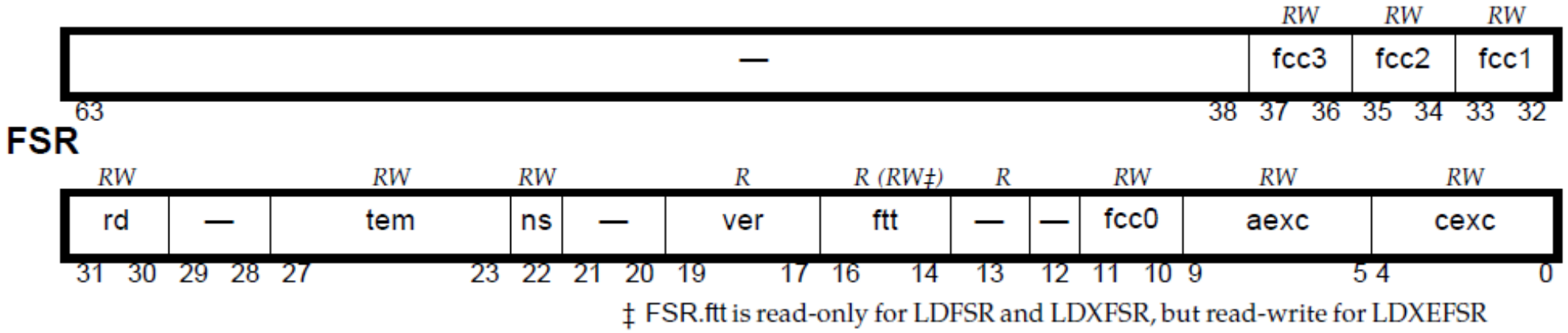
Floating-Point Registers (Cont.)

Register Operand Type	Full 6-bit Register Number						Encoding in a 5-bit Register Field in an Instruction				
Single	0	b{4}	b{3}	b{2}	b{1}	b{0}	b{4}	b{3}	b{2}	b{1}	b{0}
Double	b{5}	b{4}	b{3}	b{2}	b{1}	0	b{4}	b{3}	b{2}	b{1}	b{5}
Quad	b{5}	b{4}	b{3}	b{2}	0	0	b{4}	b{3}	b{2}	0	b{5}

Register numbers for single, double, and quad registers are encoded differently in the 5-bit register number field of a floating-point instruction. If the bits in a register number field are labelled b{4} … b{0} (where b{4} is the most significant bit of the register number), the encoding of floating-point register numbers into 5-bit instruction fields is as given in TABLE.

Floating-Point State Register (FSR)

- The Floating-Point State register (**FSR**) fields, illustrated in FIGURE, contain FPU mode and status information. The lower 32 bits of the FSR are read and written by the (deprecated) STFSR and LDFSR instructions, respectively. The 64-bit FSR register is read by the STXFSR instruction and written by the LDXFSR instruction. The ver, ftt, qne, unimplemented (for example, ns), and reserved ("—") fields of FSR are not modified by either LDFSR or LDXFSR. The LDXEFSR instruction can be used to write all implemented fields of FSR (notably ftt), but not the ver, qne, and reserved fields.



‡ FSR.ftt is read-only for LDFSR and LDXFSR, but read-write for LDXEFSR

Ancillary State Registers

- The SPARC V9 architecture defines several optional ancillary state registers (ASRs) and allows for additional ones. Access to a particular ASR may be privileged or non privileged.
- An ASR is read and written with the Read State Register and Write State Register instructions, respectively. These instructions are privileged if the accessed register is privileged.
- The SPARC V9 architecture left ASRs numbered 16–31 available for implementation-dependent uses.
- Each virtual processor contains its own set of ASRs; ASRs are not shared among virtual processors.

Ancillary State Registers (Cont.)

ASR Register Summary

ASR number	ASR name	Register	Read by Instruction(s)	Written by Instruction(s)
0	Y^{D}	Y register (deprecated)	RDY^{D}	WRY^{D}
1	—	*Reserved*	—	—
2	CCR	Condition Codes register	RDCCR	WRCCR
3	ASI	ASI register	RDASI	WRASI
4	$TICK^{P_{dis},H_{dis}}$	TICK register	$RDTICK^{P_{dis},H_{dis}}$, $RDPR^{P}$ (TICK)	$WRPR^{P}$ (TICK)
5	PC	Program Counter (PC)	RDPC	(all instructions)
6	FPRS	Floating-Point Registers Status register	RDFPRS	WRFPRS
7–13 ($7\text{-}0D_{16}$)	—	*Reserved*	—	—
14 ($0E_{16}$)	—	*Reserved*	—	—
15 ($0F_{16}$)	—	*Reserved* (used for MEMBAR; see text under MEMBAR [p. 257] or RDasr [p. 296] instructions	—	—
16–31 (10_{16}-$1F_{16}$)		non-SPARC V9 ASRs	—	—
16-18 (10_{16}- 12_{16})	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—

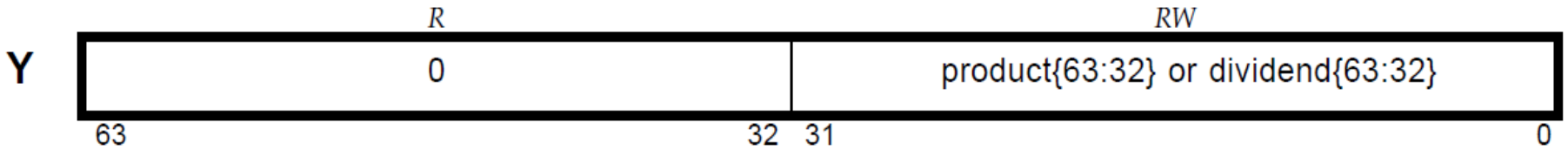
Ancillary State Registers (Cont.)

19 (13_{16})	GSR	General Status register (GSR)	RDGSR, FALIGNDATAg, many VIS and floating-point instructions	WRGSR, BMASK, SIAM
20 (14_{16})	SOFTINT_SET^{P}	(pseudo-register, for "Write 1s Set" to SOFTINT register, ASR 22)	—	WRSOFTINT_SET^{P}
21 (15_{16})	SOFTINT_CLR^{P}	(pseudo-register, for "Write 1s Clear" to SOFTINT register, ASR 22)	—	WRSOFTINT_CLR^{P}
22 (16_{16})	SOFTINT^{P}	per-virtual processor Soft Interrupt register	RDSOFTINT^{P}	WRSOFTINT^{P}
23 (17_{16})	—	*Reserved*	—	—
24 (18_{16})	STICK$^{P_{dis},H_{dis}}$	System Tick register	RDSTICK$^{P_{dis},H_{dis}}$	—
25 (19_{16})	STICK_CMPR^{P}	System Tick Compare register	RDSTICK_CMPR^{P}	WRSTICK_CMPR^{P}
26 ($1A_{16}$)	CFR	Compatibility Feature register	RDCFR	——
27 ($1B_{16}$)	PAUSE	Pause Count register	—	PAUSE (WRasr 27)
28 ($1C_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—
29 ($1D_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—
31 ($1F_{16}$)	—	Implementation dependent (impl. dep. #8-V8-Cs20, 9-V8-Cs20)	—	—

Ancillary State Registers (Cont.)

- The following registers are included in this category:

1) 32-bit Multiply/Divide Register (**Y**)
   - The low-order 32 bits of the Y register, illustrated in FIGURE, contain the more significant word of the 64-bit product of an integer multiplication, as a result of a 32-bit integer multiply (SMUL, SMULcc, UMUL, UMULcc) instruction. The Y register also holds the more significant word of the 64-bit dividend for a 32-bit integer divide (SDIV, SDIVcc, UDIV, UDIVcc) instruction.

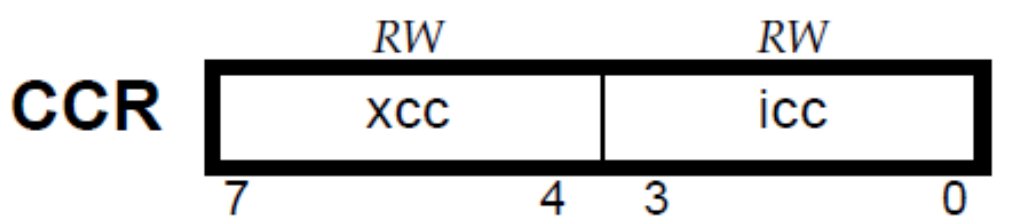


   - Although Y is a 64-bit register, its high-order 32 bits always read as 0.
   - The Y register may be explicitly read and written by the RDY and WRY instructions, respectively.

Ancillary State Registers (Cont.)

2) Integer Condition Codes Register (**CCR**)

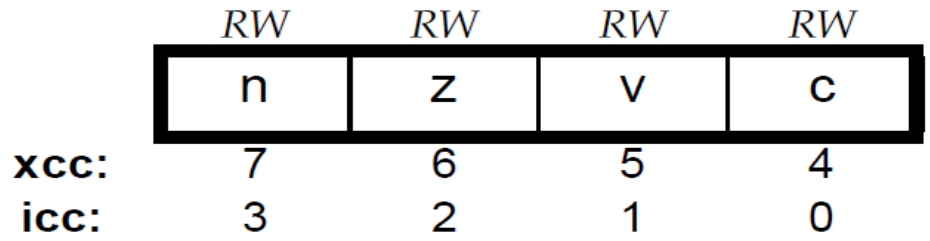
- The Condition Codes Register (CCR), shown in FIGURE, contains the integer condition codes. The CCR register may be explicitly read and written by the RDCCR and WRCCR instructions, respectively.



- All instructions that set integer condition codes set both the xcc and icc fields. The xcc condition codes indicate the result of an operation when viewed as a 64-bit operation. The icc condition codes indicate the result of an operation when viewed as a 32-bit operation. For example, if an operation results in the 64-bit value 0000 0000 FFFF FFFF16, the 32-bit result is negative (icc.n is set to 1) but the 64-bit result is nonnegative (xcc.n is set to 0).

Ancillary State Registers (Cont.)

- Each of the 4-bit condition-code fields is composed of four 1-bit subfields, as shown in FIGURE.



- The n bits indicate whether the two's-complement ALU result was negative for the last instruction that modified the integer condition codes; 1 = negative, 0 = not negative.
- The z bits indicate whether the ALU result was zero for the last instruction that modified the integer condition codes; 1 = zero, 0 = nonzero.
- The v bits signify whether the ALU result was within the range of (was representable in) 64-bit (xcc) or 32-bit (icc) two's complement notation for the last instruction that modified the integer condition codes; 1 = overflow, 0 = no overflow.

Ancillary State Registers (Cont.)

- The c bits indicate whether a 2's complement carry (or borrow) occurred during the last instruction that modified the integer condition codes. Carry is set on addition if there is a carry out of bit 63 (xcc) or bit 31 (icc). Carry is set on subtraction if there is a borrow into bit 63 (xcc) or bit 31 (icc); 1 = borrow, 0 = no borrow.

Unsigned Comparison of Operand Values	Setting of Carry bits in CCR
R[rs1]{31:0} ≥ R[rs2]{31:0}	CCR.icc.c ← 0
R[rs1]{31:0} < R[rs2]{31:0}	CCR.icc.c ← 1
R[rs1]{63:0} ≥ R[rs2]{63:0}	CCR.xcc.c ← 0
R[rs1]{63:0} < R[rs2]{63:0}	CCR.xcc.c ← 1

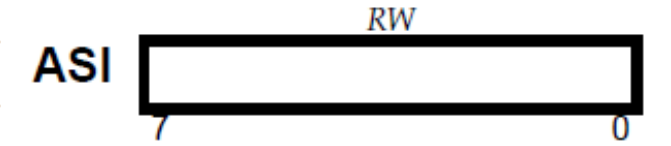
Ancillary State Registers (Cont.)

- Both fields of CCR (xcc and icc) are modified by arithmetic and logical instructions, the names of which end with the letters “cc” (for example, ANDcc), and by the WRCCR instruction. They can be modified by a DONE or RETRY instruction, which replaces these bits with the contents of TSTATE.ccr.
- The behavior of the following instructions are conditioned by the contents of CCR.icc or CCR.xcc:
  - ✓ BPcc and Tcc instructions (conditional transfer of control)
  - ✓ Bicc (conditional transfer of control, based on CCR.icc only)
  - ✓ MOVcc instruction (conditionally move the contents of an integer register)
  - ✓ FMOVcc instruction (conditionally move the contents of a floating-point register)

Ancillary State Registers (Cont.)

3) Address Space Identifier (**ASI**)

- The Address Space Identifier register (FIGURE 5-12) specifies the address space identifier to be used for load and store alternate instructions that use the "rs1 + simm13" addressing form.
- The ASI register may be explicitly read and written by the RDASI and WRASI instructions,respectively.

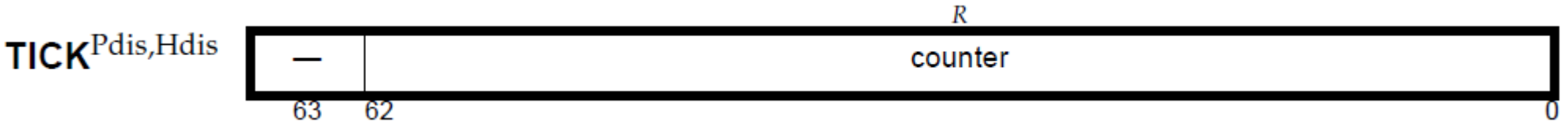


- Software (executing in any privilege mode) may write any value into the ASI register. However, values in the range 00_{16} to $7F_{16}$ are "restricted" ASIs; an attempt to perform an access using an ASI in that range is restricted to software executing in a mode with sufficient privileges for the ASI. When an instruction executing in non privileged mode attempts an access using an ASI in the range 00_{16} to $7F_{16}$ or an instruction executing in privileged mode attempts an access using an ASI the range 30_{16} to $7F_{16}$, a *privileged_action* exception is generated.

Ancillary State Registers (Cont.)

4) Tick (**TICK**) Register

- The counter field of the TICK register is a 63-bit counter that counts strand clock cycles.
- Bit 63 of the TICK register reads as 0.
- Privileged software can read the TICK register with either the RDPR or RDTICK instruction, but only when privileged access to TICK is enabled by hyper privileged software. An attempt by privileged software to read the TICK register when privileged access is disabled causes a *privileged_action* exception.
- Privileged software cannot write to the TICK register; an attempt to do so (with the WRPR instruction) results in an *illegal_instruction* exception.



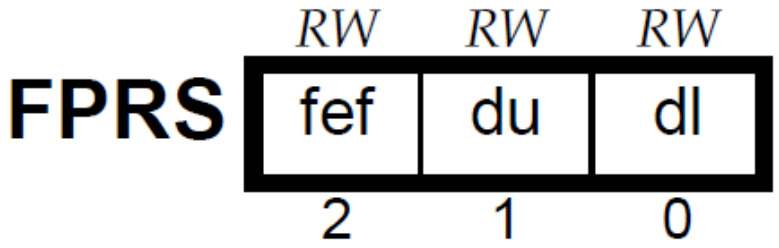
Ancillary State Registers (Cont.)

5) Program Counters (**PC, NPC**)
   - The PC contains the address of the instruction currently being executed. The least-significant two bits of PC always contain zeroes.
   - The PC can be read directly with the RDPC instruction. PC cannot be explicitly written by any instruction (including Write State Register), but is implicitly written by control transfer instructions. A WRasr to ASR 5 causes an *illegal_instruction* exception.
   - The Next Program Counter, NPC, is a pseudo-register that contains the address of the next instruction to be executed if a trap does not occur. The least-significant two bits of NPC always contain zeroes.
   - NPC is written implicitly by control transfer instructions. However, NPC cannot be read or written explicitly by any instruction.

Ancillary State Registers (Cont.)

6) Floating-Point Registers State (**FPRS**) Register

- The Floating-Point Registers State (FPRS) register, shown in FIGURE, contains control information for the floating-point register file; this information is readable and writable by non privileged software.

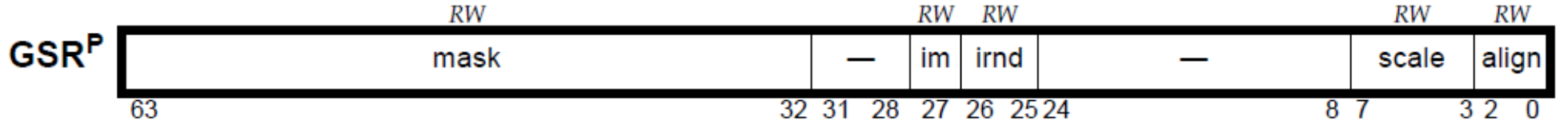


- The FPRS register may be explicitly read and written by the RDFPRS and WRFPRS instructions, respectively.

Ancillary State Registers (Cont.)

7) General Status Register (**GSR**)

- The General Status Register1 (GSR) is a non privileged read/write register that is implicitly referenced by many VIS instructions. The GSR can be read by the RDGSR instruction (*Read Ancillary State Register*) and written by the WRGSR instruction (*Write Ancillary State Register*).
- If the FPU is disabled (PSTATE.pef = 0 or FPRS.fef = 0), an attempt to access this register using an otherwise-valid RDGSR or WRGSR instruction causes an *fp_disabled* trap.



Ancillary State Registers (Cont.)

GSR Bit Description

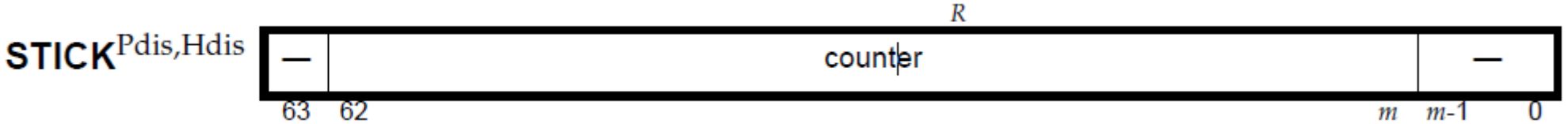
Bit	Field	Description
63:32	mask	This 32-bit field specifies the mask used by the BSHUFFLE instruction. The field contents are set by the BMASK instruction.
31:28	—	*Reserved.*
27	im	Interval Mode: If GSR.im = 0, rounding is performed according to FSR.rd; if GSR.im = 1, rounding is performed according to GSR.irnd.
26:25	irnd	IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.im = 1), as follows:
24:8	—	*Reserved.*
7:3	scale	5-bit shift count in the range 0–31, used by the FPACK instructions for formatting.
2:0	align	Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.

irnd	Round toward ...
0	Nearest (even, if tie)
1	0
2	+ ∞
3	– ∞

Ancillary State Registers (Cont.)

8) System Tick (**STICK**) Register

- The System Tick (STICK) register provides a counter that consistently measures time across all virtual processors (strands) of a system.
- The 63-bit counter field of the STICK register automatically increments at a fixed frequency of 1.0 GHz, therefore counter bit *n* will be observed to increment at a frequency of 1.0 GHz ÷ 2^n.



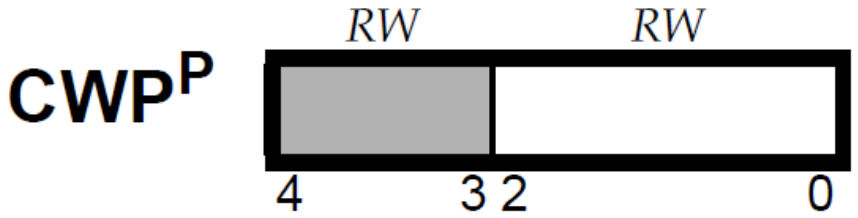
- The counter field spans bits 62:*m* of the STICK register. *m* is implementation-dependent , but *m* must be less than or equal to 4 (STICK granularity of 16ns or better); *m* can be 0.
- The counter must increment 1 billion ± 25,000 times every second (an error rate of no more than 25 parts per million). This means that the counter must not gain or lose more than 2.16 seconds per day.

Register-Window PR State Registers

- The state of the register windows is determined by the contents of a set of privileged registers. These state registers can be read/written by privileged software using the RDPR/WRPR instructions.
- An attempt by non privileged software to execute a RDPR or WRPR instruction cause a *privileged_opcode* exception.
- In addition, these registers are modified by instructions related to register windows and are used to generate traps that allow supervisor software to spill, fill, and clean register windows.

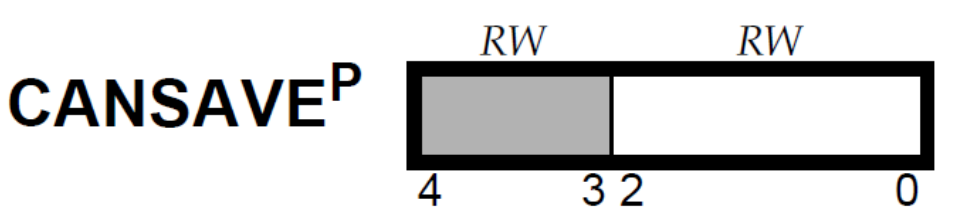
Register-Window PR State Registers (Cont.)

1) Current Window Pointer (**CWP**) Register
   - The privileged CWP register, shown in FIGURE, is a counter that identifies the current window into the array of integer registers.



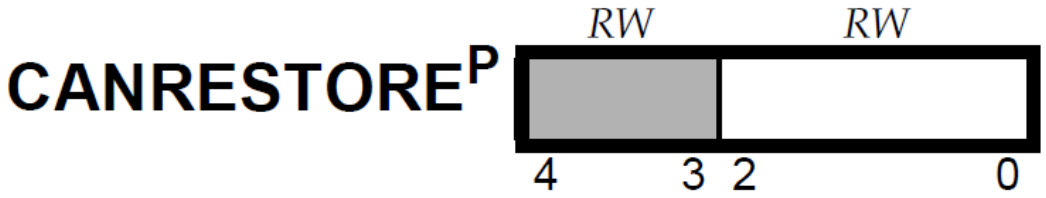
2) Savable Windows (**CANSAVE**) Register
   - The privileged CANSAVE register, shown in FIGURE, contains the number of register windows following CWP that are not in use and are, hence, available to be allocated by a SAVE instruction without generating a window spill exception.

Register-Window PR State Registers (Cont.)



3) Restorable Windows (**CANRESTORE**) Register

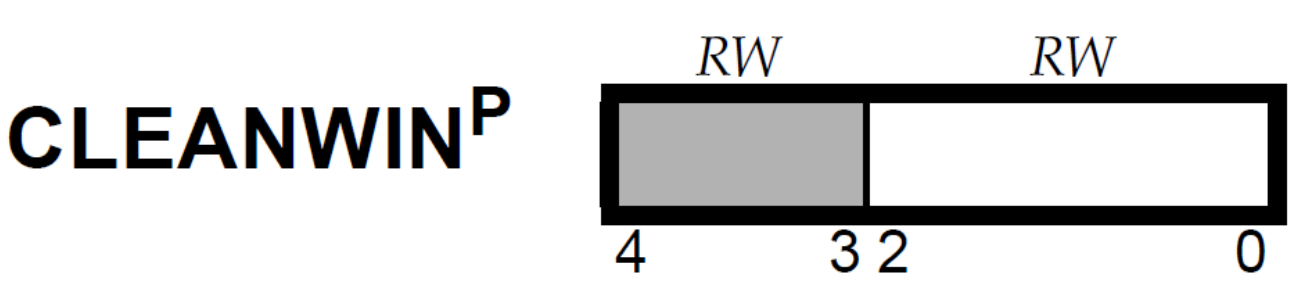
- The privileged CANRESTORE register, shown in FIGURE 5-24, contains the number of register windows preceding CWP that are in use by the current program and can be restored (by the RESTORE instruction) without generating a window fill exception.



Register-Window PR State Registers (Cont.)

4) Clean Windows (**CLEANWIN**) Register

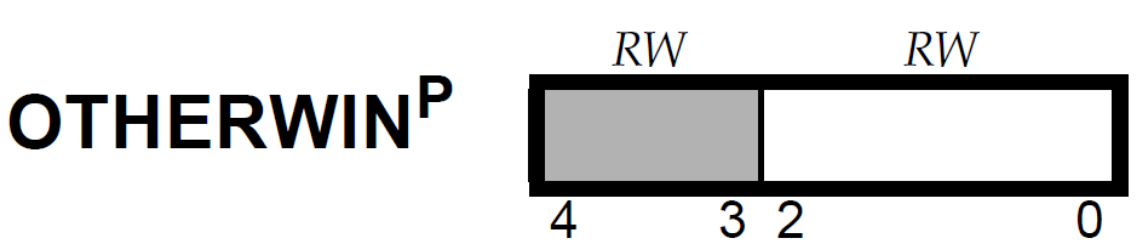
- The privileged CLEANWIN register, shown in FIGURE, contains the number of windows that can be used by the SAVE instruction without causing a *clean_window* exception.
- The CLEANWIN register counts the number of register windows that are "clean" with respect to the current program; that is, register windows that contain only zeroes, valid addresses, or valid data from that program. Registers in these windows need not be cleaned before they can be used.



Register-Window PR State Registers (Cont.)

5) Other Windows (**OTHERWIN**) Register

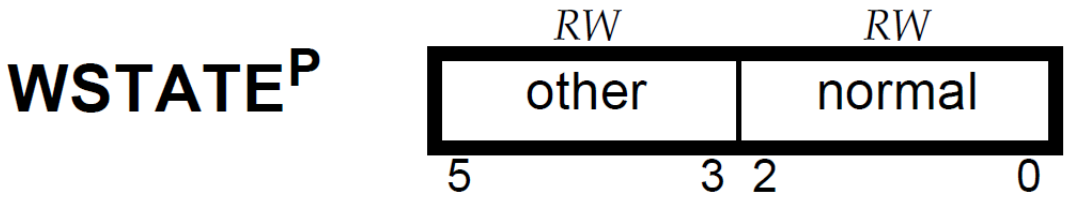
- The privileged OTHERWIN register, shown in FIGURE 5-26, contains the count of register windows that will be spilled/filled by a separate set of trap vectors based on the contents of WSTATE.other. If OTHERWIN is zero, register windows are spilled/filled by use of trap vectors based on the contents of WSTATE.normal.
- The OTHERWIN register can be used to split the register windows among different address spaces and handle spill/fill traps efficiently by use of separate spill/fill vectors.



Register-Window PR State Registers (Cont.)

6) Window State (**WSTATE**) Register

- The privileged WSTATE register, shown in FIGURE, specifies bits that are inserted into TT[TL]{4:2} on traps caused by window spill and fill exceptions. These bits are used to select one of eight different window spill and fill handlers. If OTHERWIN = 0 at the time a trap is taken because of a window spill or window fill exception, then the WSTATE.normal bits are inserted into TT[TL]. Otherwise, the WSTATE.other bits are inserted into TT[TL].



Non-Register-Window PR State Registers

- The registers described in this section are visible only to software running in privileged mode (that is, when PSTATE.priv = 1), and may be accessed with the WRPR and RDPR instructions. (An attempt to execute a WRPR or RDPR instruction in nonprivileged mode causes a *privileged_opcode* exception.)
- Each virtual processor provides a full set of these state registers.
- **Implementation Note :**
  - ✓ A write to any privileged register, including PR state registers, may drain the CPU pipeline.

Non-Register-Window PR State Registers (Cont.)

1) Trap Program Counter (**TPC**) Register
   - The privileged Trap Program Counter register contains the program counter (PC) from the previous trap level. There are *MAXPTL* instances of the TPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TPC[TL] register is accessible. An attempt to read or write the TPC register when TL = 0 causes an *illegal_instruction* exception.

Events that involve TPC, when executing with TL = n.

Event	Effect
Trap	TPC[$n+1$] ← PC
RETRY instruction	PC ← TPC[n]
RDPR (TPC)	R[rd] ← TPC[n]
WRPR (TPC)	TPC[n] ← *value*

Non-Register-Window PR State Registers (Cont.)

2) Trap Next PC (**TNPC**) Register

- The privileged Trap Next Program Counter register is the next program counter (NPC) from the previous trap level. There are *MAXPTL* instances of the TNPC, but only one is accessible at any time. The current value in the TL register determines which instance of the TNPC register is accessible. An attempt to read or write the TNPC register when TL = 0 causes an *illegal_instruction* exception.

Events that involve TNPC, when executing with TL = n.

Event	Effect
Trap	TNPC[$n+1$] ← NPC
DONE instruction	PC ← TNPC[n]; NPC ← TNPC[n] + 4
RETRY instruction	NPC ← TNPC[n]
RDPR (TNPC)	R[rd] ← TNPC[n]
WRPR (TNPC)	TNPC[n] ← *value*

Non-Register-Window PR State Registers (Cont.)

3) Trap State (**TSTATE**) Register

- The privileged Trap State register contains the state from the previous trap level, comprising the contents of the GL, CCR, ASI, CWP, and PSTATE registers from the previous trap level. There are *MAXPTL* instances of the TSTATE register, but only one is accessible at a time. The current value in the TL register determines which instance of TSTATE is accessible. An attempt to read or write the TSTATE register when TL = 0 causes an *illegal_instruction* exception.

Events That Involve TSTATE, When Executing with TL = n

Event	Effect
Trap	TSTATE[$n + 1$] ← (registers)
DONE instruction	(registers) ← TSTATE[n]
RETRY instruction	(registers) ← TSTATE[n]
RDPR (TSTATE)	R[rd] ← TSTATE[n]
WRPR (TSTATE)	TSTATE[n] ← *value*

Non-Register-Window PR State Registers (Cont.)

4) Trap Type (**TT**) Register

- The privileged Trap Type register contains the trap type of the trap that caused entry to the current trap level. There are *MAXPTL* instances of the TT register, but only one is accessible at a time. The current value in the TL register determines which instance of the TT register is accessible. An attempt to read or write the TT register when TL = 0 causes an *illegal_instruction* exception.
- During normal operation, the value of TT[*n*], where *n* is greater than the current trap level (*n* > TL), is undefined.

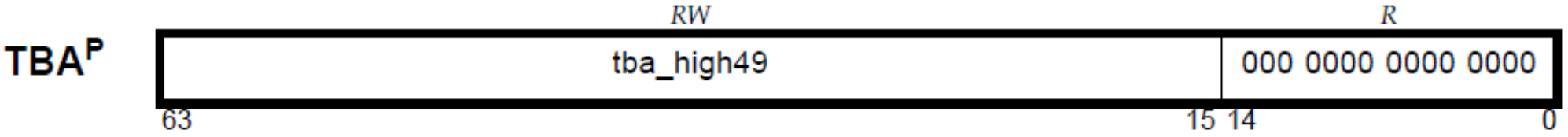
Events that involve TT, when executing with TL = *n*.

Event	Effect
Trap	TT[*n* + 1] ← (trap type)
RDPR (TT)	R[rd] ← TT[*n*]
WRPR (TT)	TT[*n*] ← *value*

Non-Register-Window PR State Registers (Cont.)

5) Trap Base Address (**TBA**) Register

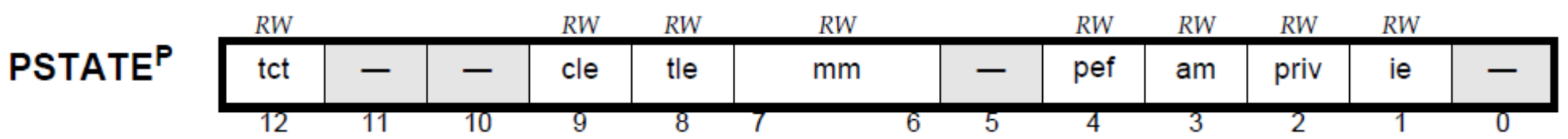
- The privileged Trap Base Address register (TBA), shown in FIGURE, provides the upper 49 bits (bits 63:15) of the virtual address used to select the trap vector for a trap that is to be delivered to privileged mode. The lower 15 bits of the TBA always read as zero, and writes to them are ignored.



Non-Register-Window PR State Registers (Cont.)

6) Processor State (**PSTATE**) Register

- The privileged Processor State register (PSTATE), shown in FIGURE , contains control fields for the current state of the virtual processor. There is only one instance of the PSTATE register per virtual processor.
- Writes to PSTATE are non delayed; that is, new machine state written to PSTATE is visible to the next instruction executed. The privileged RDPR and WRPR instructions are used to read and write PSTATE, respectively.



Non-Register-Window PR State Registers (Cont.)

7) Trap Level (**TL**) Register

- The privileged Trap Level register (TL) specifies the current trap level. TL = 0 is the normal (non trap) level of operation. TL > 0 implies that one or more traps are being processed.
- The maximum valid value that the TL register may contain is *MAXPTL*, which is always equal to the number of supported trap levels beyond level 0.

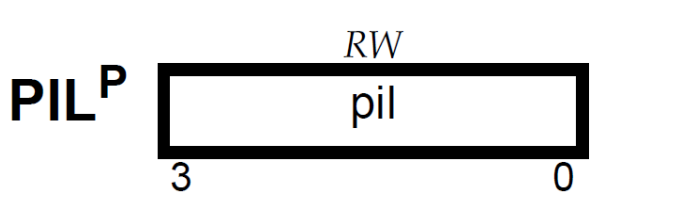
Effect of WRPR of Value x to Register TL

Value x Written with WRPR	Privilege Level when Executing WRPR		
	Nonprivileged	Privileged	
$x \leq MAXPTL$	*privileged_opcode* exception	TL ← x	
$x > MAXPTL$		TL ← *MAXPTL* (no exception generated)	

Non-Register-Window PR State Registers (Cont.)

8) Processor Interrupt Level (**PIL**) Register

- The privileged Processor Interrupt Level register (PIL; see FIGURE) specifies the interrupt level above which the virtual processor will accept an *interrupt_level_n* interrupt. Interrupt priorities are mapped so that interrupt level 2 has greater priority than interrupt level 1, and so on.
- On SPARC V8 processors, the level 15 interrupt is considered to be non maskable, so it has different semantics from other interrupt levels.



Non-Register-Window PR State Registers (Cont.)

9) Global Level (**GL**) Register

- The privileged Global Level (GL) register selects which set of global registers is visible at any given time.
- When a trap occurs, GL is stored in TSTATE[TL].GL, GL is incremented, and a new set of global registers (R[1] through R[7]) becomes visible. A DONE or RETRY instruction restores the value of GL from TSTATE[TL].
- The valid range of values that the GL register may contain is 0 to *MAXPGL*, where *MAXPGL* is one fewer than the number of global register sets available to the virtual processor.

Effect of WRPR to Register GL

Value x Written with WRPR	Privilege Level when WRPR Is Executed	
	Nonprivileged	Privileged
$x \leq MAXPGL$	*privileged_opcode* exception	GL ← x
$x > MAXPGL$		GL ← *MAXPGL* (no exception generated)

Instruction Set

Oracle SPARC Architecture Feature Sets

Architectural Feature Set Name	Sun Studio Compiler Option that enables generation of instructions in Feature Set (-xarch=)	SPARC Processor in which Feature Set was first Implemented (year)	Oracle SPARC Architecture Specification in which Feature Set first appeared
SPARC V9	(enabled by default)	UltraSPARC I (1995)	*The SPARC Architecture Manual-Version 9*
VIS 1	`sparcvis`	UltraSPARC I (1995)	UltraSPARC Architecture 2005
VIS 2	`sparcvis2`	UltraSPARC III (2001)	UltraSPARC Architecture 2005
FMAf	`sparcfmaf`	SPARC T3 (2010)	Oracle SPARC Architecture 2011
VIS 3	`sparcvis3`	SPARC T3 (2010)	Oracle SPARC Architecture 2011
IMA	`sparcima`	SPARC64 VII (~2009), SPARC T4 (2011)	Oracle SPARC Architecture 2011
SPARC4	`sparc4`	SPARC T4 (2011)	Oracle SPARC Architecture 2011

Instruction Set (Cont.)

Instruction Superscripts

Superscript	Meaning
D	Deprecated instruction (do not use in new software)
H_{dis}	Privileged action if in nonprivileged or privileged mode and access is disabled
N	Nonportable instruction
P	Privileged instruction
P_{ASI}	Privileged action if bit 7 of the referenced ASI is 0
P_{ASR}	Privileged instruction if the referenced ASR register is privileged
P_{dis}	Privileged action if in nonprivileged mode (PSTATE.priv = 0) and nonprivileged access is disabled

Within these tables and throughout the rest of instruction set, and in ISA, *Opcode Maps*, certain opcodes are marked with mnemonic superscripts.
The superscripts and their meanings are defined in TABLE.

Instruction Set (Cont.)

Instruction	*Category and Function*
	Data Movement Operations, Between R Registers
MOVcc	Move integer register if condition is satisfied
MOVr	Move integer register on contents of integer register
	Data Movement Operations, Between F Registers
FMOV<s|d|q>	Floating-point move
FMOV<s|d|q>cc	Move floating-point register if condition is satisfied
FMOV<s|d|q>R	Move f-p reg. if integer reg. contents satisfy condition
FSRC<1|2><s|d>	Copy source to destination
	Data Movement Operations, Between R and F Registers
MOVfTOi	Move floating-point register to integer register

Instruction Set (Cont.)

Instruction	*Category and Function*
MOViTOf	Move integer register to floating-point register
	Data Conversion Instructions
FiTO<s|d|q>	Convert 32-bit integer to floating-point
F<s|d|q>TOi	Convert floating point to integer
F<s|d|q>TOx	Convert floating point to 64-bit integer
F<s|d|q>TO<s|d|q>	Convert between floating-point formats
FxTO<s|d|q>	Convert 64-bit integer to floating-point
	Logical Operations on R Registers
AND (ANDcc)	Logical **and** (and modify condition codes)
OR (ORcc)	Inclusive-**or** (and modify condition codes)

Instruction Set (Cont.)

Instruction	*Category and Function*
ORN (ORNcc)	Inclusive-**or not** (and modify condition codes)
XNOR (XNORcc)	Exclusive-**nor** (and modify condition codes)
XOR (XORcc)	Exclusive-**or** (and modify condition codes)
	Logical Operations on F Registers
FAND<s|d>	Logical **and** operation
FANDNOT<s|d>	Logical **and** operation with one inverted source
FNAND<s|d>	Logical **nand** operation
FNOR<s|d>	Logical **nor** operation
FNOT<1|2><s|d>	Copy negated source
FONE<s|d>	One fill
FOR<s|d>	Logical **or** operation

Instruction Set (Cont.)

Instruction	*Category and Function*
FORNOT<s|d>	Logical **or** operation with one inverted source
FXNOR<s|d>	Logical **xnor** operation
FXOR<s|d>	Logical **xor** operation
FZERO<s|d>	Zero fill
	Shift Operations on R Registers
SLL	Shift left logical
SLLX	Shift left logical, extended
SRA	Shift right arithmetic
SRAX	Shift right arithmetic, extended
SRL	Shift right logical
SRLX	Shift right logical, extended

Instruction Set (Cont.)

Instruction	*Category and Function*
	Special Addressing and Data Alignment Operations
ALIGNADDRESS	Calculate address for misaligned data
ARRAY<8|16|32>	3-D array addressing instructions
FALIGNDATAg	Perform data alignment for misaligned data(using GSR.align)
	Control Transfers
Bicc	Branch on integer condition codes
BPcc	Branch on integer condition codes with prediction
BCr	Branch on contents of integer register with prediction
CALL	Call and link
Cbcond	Compare and Branch
DONE^{P}	Return from trap

Instruction Set (Cont.)

Instruction	*Category and Function*
FBfccD	Branch on floating-point condition codes
FBPfcc	Branch on floating-point condition codes with prediction
ILLTRAP	Illegal instruction
JMPL	Jump and link
RETRY	Return from trap and retry
RETURN	Return
Tcc	Trap on integer condition codes
	Byte Permutation
BMASK	Set the GSR.mask field
BSHUFFLE	Permute bytes as specified by GSR.mask
CMASK<8|16|32>^{N}	Create GSR.mask from SIMD comparison result

Instruction Set (Cont.)

Instruction	*Category and Function*
	Data Formatting Operations on F Registers
FEXPAND	Pixel expansion
FPACK<16|32|FIX>	Pixel packing
FPMERGE	Pixel merge
	Memory Operations to/from F Registers
LDBLOCKF^{D}	Block loads
STBLOCKF	Block stores
LDDF	Load double floating-point
LDDFA$^{P}_{ASI}$	Load double floating-point from alternate space
LDF	Load floating-point
LDFA$^{P}_{ASI}$	Load floating-point from alternate space

Instruction Set (Cont.)

Instruction	*Category and Function*
LDQF	Load quad floating-point
LDQFA	Load quad floating-point from alternate space
LDSHORTF	Short floating-point loads
STDF	Store double floating-point
STDFA	Store double floating-point into alternate space
STF	Store floating-point
STFA	Store floating-point into alternate space
STPARTIALF	Partial Store instructions
STQF	Store quad floating point
STQFA	Store quad floating-point into alternate space
STSHORTF	Short floating-point stores

Instruction Set (Cont.)

Instruction	*Category and Function*
	Memory Operations — Miscellaneous
LDFSR	Load floating-point state register (lower)
LDXEFSR	Load Entire floating-point state register
LDXFSR	Load floating-point state register
MEMBAR	Memory barrier
PREFETCH	Prefetch data
PREFETCHA	Prefetch data from alternate space
STFSR	Store floating-point state register (lower)
STXFSR	Store floating-point state register
	Atomic (Load-Store) Memory Operations to/from R Registers
CASA	Compare and swap word in alternate space

Instruction Set (Cont.)

Instruction	*Category and Function*
CASXA	Compare and swap doubleword in alternate space
LDSTUB	Load-store unsigned byte
LDSTUBA	Load-store unsigned byte in alternate space
SWAP	Swap integer register with memory
SWAPA	Swap integer register with memory in alternate space
	Memory Operations to/from R Registers
LDSB	Load signed byte
LDSBX	Load signed byte from alternate space
LDSH	Load signed halfword
LDSHA	Load signed halfword from alternate space
LDSW	Load signed word

Instruction Set (Cont.)

Instruction	*Category and Function*
LDSWA	Load signed word from alternate space
LDTXA	Load integer twin extended word from alternate space
LDTW	Load integer twin word
LDTWA	Load integer twin word from alternate space
LDUB	Load unsigned byte
LDUBA	Load unsigned byte from alternate space
LDUH	Load unsigned halfword
LDUHA	Load unsigned halfword from alternate space
LDUX	Load unsigned word
LDUWZ	Load unsigned word from alternate space
LDX	Load extended
LDXA	Load extended from alternate space

Instruction Set (Cont.)

Instruction	*Category and Function*
STB	Store byte
STBA	Store byte into alternate space
STTW	Store twin word
STTWA	Store twin word into alternate space
STH	Store halfword
STHA	Store halfword into alternate space
STW	Store word
STWA	Store word into alternate space
STX	Store extended
STXA	Store extended into alternate space

Instruction Set (Cont.)

Instruction	*Category and Function*
	Floating-Point Arithmetic Operations
FABS<s|d|q>	Floating-point absolute value
FADD<s|d|q>	Floating-point add
FDIV<s|d|q>	Floating-point divide
FdMULq	Floating-point multiply double to quad
FHADD<s|d>	Floating-point add and halve
FHSUB<s|d>	Floating-point subtract and halve
FMADD<s|d>	Floating-point multiply-add single/double (fused)
FMSUB<s|d>	Floating-point multiply-subtract single/double (fused)
FMUL<s|d|q>	Floating-point multiply
FNADD<s|d>	Floating-point add and negate
FNHADD<s|d>	Floating-point add, halve, and negate

Instruction Set (Cont.)

Instruction	*Category and Function*
FNMADD(s,d)	Floating-point negative multiply-add single/double (fused)
FNMUL<s|d>	Floating-point multiply and negate
FNEG<s|d|q>	Floating-point negate
FNMSUB(s,d)	Floating-point negative multiply-subtract single/double (fused)
FNsMULd	Floating-point multiply single to double, and negate
FsMULd	Floating-point multiply single to double
FSQRT<s|d|q>	Floating-point square root
	Floating-Point Comparison Operations
FCMP<s|d|q>	Floating-point compare
FCMPE<s|d|q>	Floating-point compare (exception if unordered)
FLCMP{s,d}	Lexicographic compare

Instruction Set (Cont.)

Instruction	*Category and Function*
	Register-Window Control Operations
ALLCLEAN	Mark all register window sets as “clean”
INVALW	Mark all register window sets as “invalid”
FLUSHW	Flush register windows
RESTORE	Restore caller’s window
RESTORED	Window has been restored
SAVE	Save caller’s window
SAVED	Window has been saved
	Miscellaneous Operations
FLUSH	Flush instruction memory
NOP	No operation

Instruction Set (Cont.)

Instruction	*Category and Function*
	Integer Arithmetic Operations on R Registers
ADD (ADDcc)	Add (and modify condition codes
ADDC (ADDCcc)	Add with carry (and modify condition codes)
ADDXC (ADDXCcc)	Add with extended carry (and modify condition codes)
MULX	Multiply 64-bit integers
SDIV (SDIVcc)	32-bit signed integer divide (and modify condition codes)
SDIVX	64-bit signed integer divide
SMUL (SMULcc)	Signed integer multiply (and modify condition codes)
SUB (SUBcc)	Subtract (and modify condition codes)
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)
UDIVX	64-bit unsigned integer divide
XMULX[HI]	XOR Multiply

Instruction Set (Cont.)

Instruction	*Category and Function*
	Integer SIMD (Partitioned) Operations on F Registers
FMEAN16	16-bit partitioned average
FPADD	Partitioned integer add
FPADD64	Partitioned integer add, 64-bit
FPADDS	Partitioned integer add with saturation
FPCMP	Partitioned Compare signed integer values
FPCMPU	Partitioned Compare unsigned integer values
FPSUB<16,32>[S]	Partitioned integer subtract
FPSUB64	Partitioned integer add, 64-bit
FPSUBS	Partitioned integer subtract with saturation

Instruction Set (Cont.)

Instruction	*Category and Function*
	Integer Arithmetic Operations on F Registers
FMUL8x16	8x16 partitioned product
FMUL8x16[AU|AL]	8x16 upper/lower α partitioned product
FMUL8[SU|UL]x16	8x16 upper/lower partitioned product
FMULD8[SU|UL]$_{x16}$	8x16 upper/lower partitioned product
FPMADDX[HI]	Integer multiply-add (low and high 64 bit results)
FS<LL|RL|RA>	16- or 32-bit partitioned shift, left or right
	Miscellaneous Operations on R Registers
LZCNT	Leading zero detect on 64-bit integer register
POPC	Population count
SETHI	Set high 22 bits of low word of integer register

Instruction Set (Cont.)

Instruction	*Category and Function*
	Miscellaneous Operations on F Registers
EDGE<8|16|32>[L]	Edge handling instructions (and modify condition codes)
EDGE<8|16|32>[L]	Edge handling instructions
FCHKSM16	16-bit partitioned checksum
PDISTD	Pixel component distance
PDISTN	Distance between eight 8-bit components with no accumulation 285
	Cryptographic and Secure Hash Operations
AES_DROUND	AES Decryption, Columns 0&1 (or 2&3)
AES_EROUND	AES Encryption, Columns 0&1 (or 2&3)
SHA256	SHA256 Secure Hash operation
SHA512	SHA512 Secure Hash operation

Instruction Set (Cont.)

Instruction	*Category and Function*
	Control and Status Register Access
PAUSE	Pause Virtual Processor
RDASI	Read ASI register
RDasr	Read ancillary state register
RDCCR	Read Condition Codes register (CCR)
RDFPRS	Read Floating-Point Registers State register (FPRS)
RDGSR	Read General Status register (GSR)
RDPC	Read Program Counter register (PC)
RDPR	Read privileged register
RDSOFTINT	Read per-virtual processor Soft Interrupt register (SOFTINT)
RDSTICK	Read System Tick register (STICK)
RDTICK	Read Tick register (TICK)

Instruction Set (Cont.)

Instruction	*Category and Function*
	Control and Status Register Access
SIAM	Set interval arithmetic mode
WDASI	Write ASI register
WDasr	Write ancillary state register
WDCCR	Write Condition Codes register (CCR)
WDFPRS	Write Floating-Point Registers State register (FPRS)
WDGSR	Write General Status register (GSR)
WDPC	Write Program Counter register (PC)
WDPR	Write privileged register
WDSOFTINT	Write per-virtual processor Soft Interrupt register (SOFTINT)
WDSTICK	Write System Tick register (STICK)
WDTICK	Write Tick register (TICK)

Memory Model

- The **Oracle SPARC Architecture** *memory models* define the semantics of memory operations. The instruction set semantics require that loads and stores behave *as if* they are performed in the order in which they appear in the dynamic control flow of the program. The *actual* order in which they are processed by the memory may be different. The purpose of the memory models is to specify what constraints, if any, are placed on the order of memory operations.

- The memory models apply both to uniprocessor and to shared memory multiprocessors. Formal memory models are necessary for precise definitions of the interactions between multiple virtual processors and input/output devices in a shared memory configuration. Programming shared memory multiprocessors requires a detailed understanding of the operative memory model and the ability to specify memory operations at a low level in order to build programs that can safely and reliably coordinate their activities.

Memory Model (Cont.)

- Memory Location Identification
  - ✓ A memory location is identified by an 8-bit address space identifier (ASI) and a 64-bit memory address. The 8-bit ASI can be obtained from an ASI register or included in a memory access instruction. The ASI used for an access can distinguish among different 64-bit address spaces, such as Primary memory space, Secondary memory space, and internal control registers. It can also apply attributes to the access, such as whether the access should be performed in big- or little-endian byte order, or whether the address should be taken as a virtual or real.
- Memory Accesses and Cacheability
  - ✓ Memory is logically divided into real memory (cached) and I/O memory (non cached with and without side effects) spaces.

Memory Model (Cont.)

1. *Real memory* stores information without side effects. A load operation returns the value most recently stored. Operations are side-effect-free in the sense that a load, store, or atomic load-store to a location in real memory has no program-observable effect, except upon that location (or, in the case of a load or load-store, on the destination register).
2. *I/O locations* may not behave like memory and may have side effects. Load, store, and atomic load store operations performed on I/O locations may have observable side effects, and loads may not return the value most recently stored. The value semantics of operations on I/O locations are *not* defined by the memory models, but the constraints on the order in which operations are performed is the same as it would be if the I/O locations were real memory. The storage properties, contents, semantics, ASI assignments, and addresses of I/O registers are implementation dependent.

Memory Model (Cont.)

Coherence Domains

- Two types of memory operations are supported in the Oracle SPARC Architecture: cacheable and non cacheable accesses. The manner in which addresses are differentiated is implementation dependent. In some implementations, it is indicated in the page translation entry (TTE.cp).
- Although SPARC V9 does not specify memory ordering between cacheable and non cacheable accesses, the Oracle SPARC Architecture maintains TSO ordering between memory references regardless of their cacheability.
  - Cacheable Accesses

    Accesses within the coherence domain are called cacheable accesses. They have these properties:

    I. Data reside in real memory locations.

    II. Accesses observe supported cache coherency protocol(s).

    III. The cache line size is 2^n bytes (where $n \geq 4$), and can be different for each cache.

Memory Model (Cont.)

- Noncacheable Accesses
  - Noncacheable accesses are outside of the coherence domain. They have the following properties:

I. Data might not reside in real memory locations. Accesses may result in programmer-visible side effects. An example is memory-mapped I/O control registers.

II. Accesses do not observe supported cache coherency protocol(s).

III. The smallest unit in each transaction is a single byte.

  - The Oracle SPARC Architecture MMU optionally includes an attribute bit in each page translation, TTE.e, which when set signifies that this page has side effects.
  - Noncacheable accesses without side effects (TTE.e = 0) are processor-consistent and obey TSO memory ordering. In particular, processor consistency ensures that a noncacheable load that references the same location as a previous noncacheable store will load the data from the previous store.
  - Noncacheable accesses with side effects (TTE.e = 1) are processor consistent and are strongly ordered.

Memory Model (Cont.)

- Address Space Identifiers

- The virtual processor provides an address space identifier with every address. This ASI may serve several purposes:
  I. To identify which of several distinguished address spaces the 64-bit address offset is addressing
  II. To provide additional access control and attribute information, for example, to specify the endianness of the reference
  III. To specify the address of an internal control register in the virtual processor, cache, or memory management hardware

- Memory management hardware can associate an independent 264-byte memory address space with each ASI. In practice, the three independent memory address spaces (contexts) created by the MMU are Primary, Secondary, and Nucleus.

Memory Model (Cont.)

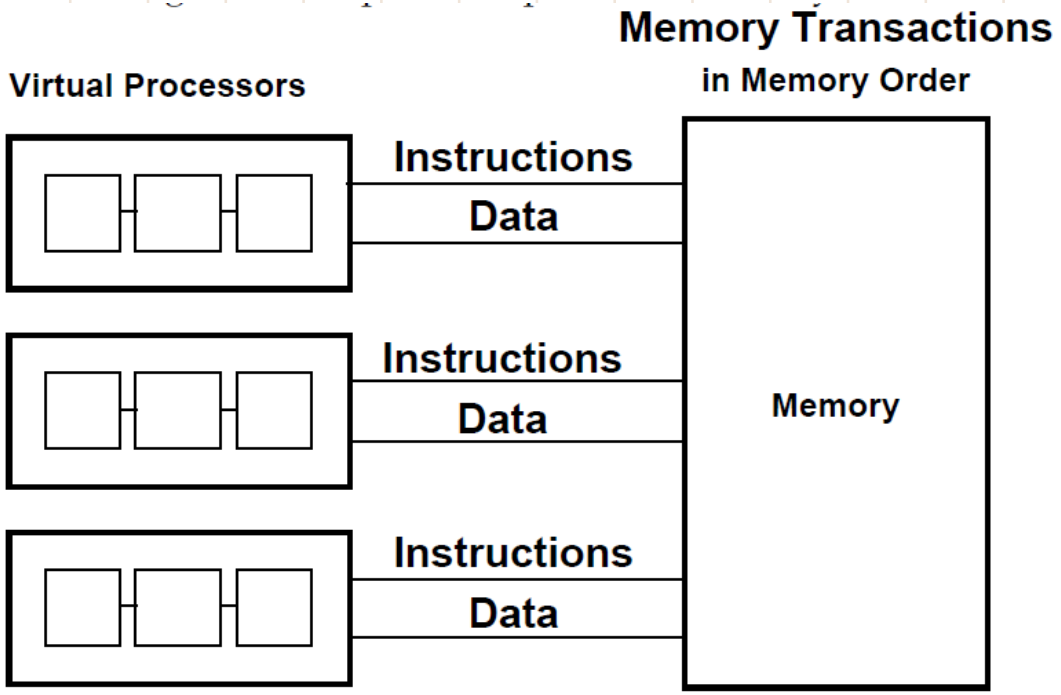
- Accesses to other address spaces use the load/store alternate instructions. For these accesses, the ASI is either contained in the instruction (for the register+register addressing mode) or taken from the ASI register (for register+immediate addressing).
- ASIs are either non restricted or restricted-to-privileged:
  a. A non restricted ASI (ASI range 8016 – FF16) is one that may be used independently of the privilege level (PSTATE.priv) at which the virtual processor is running.
  b. A restricted-to-privileged ASI (ASI range 0016 – 2F16) requires that the virtual processor be in privileged mode for a legal access to occur.

Allowed Accesses to ASIs

ASI Value	Type	Result of ASI Access in NP Mode	Result of ASI Access in P Mode
00_{16} –– $2F_{16}$	Restricted-to-privileged	*privileged_action* exception	Valid Access
80_{16} – FF_{16}	Nonrestricted	Valid Access	Valid Access

Memory Model (Cont.)

- Virtual Processor/Memory Interface Model



Each Oracle SPARC Architecture virtual processor in a multiprocessor system is modelled as shown in FIGURE ; that is, having two independent paths to memory: one for instructions and one for data.

Memory Model (Cont.)

- Data caches are maintained by hardware so their contents always appear to be consistent (coherent). Instruction caches are *not* required to be kept consistent with data caches and therefore require explicit program (software) action to ensure consistency when a program modifies an executing instruction stream. Memory is shared in terms of address space, but it may be nonhomogeneous and distributed in an implementation. Caches are ignored in the model, since their functions are transparent to the memory model.

- In real systems, addresses may have attributes that the virtual processor must respect. The virtual processor executes loads, stores, and atomic load-stores in whatever order it chooses, as constrained by program order and the memory model.

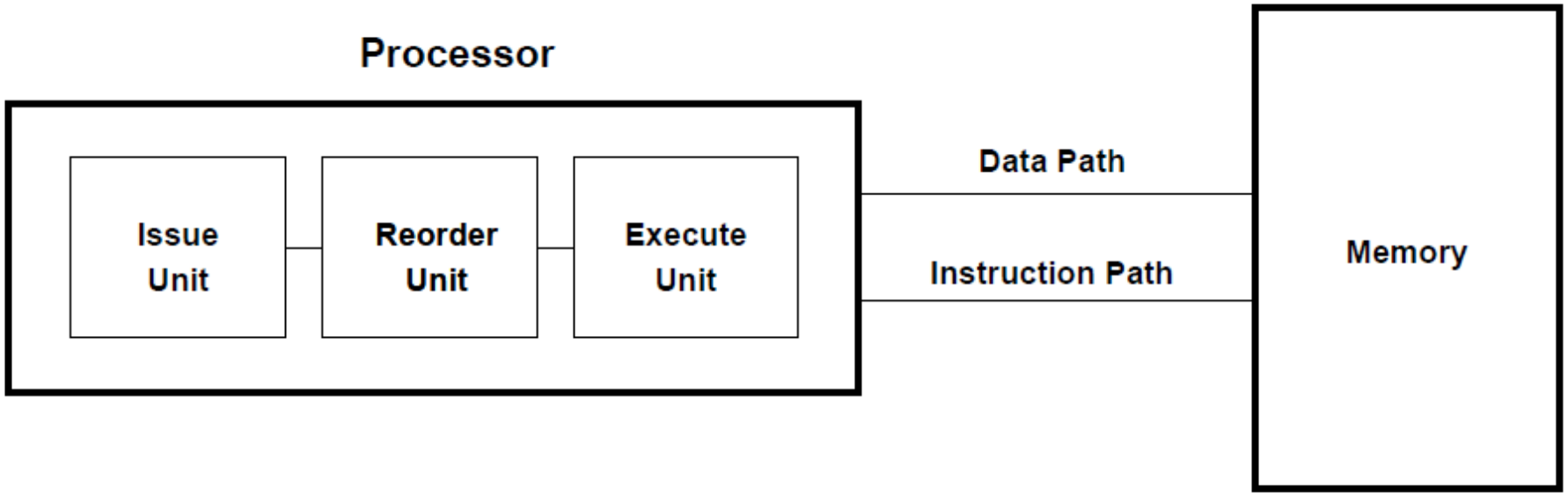
- Instructions are performed in an order constrained by local dependencies. Using this dependency ordering, an execution unit submits one or more pending memory transactions to the memory. The memory performs transactions in *memory order*. The memory unit may perform transactions submitted to it out of order; hence, the execution unit must not concurrently submit two or more transactions that are required to be ordered, unless the memory unit can still guarantee in-order semantics.

Memory Model (Cont.)

- The memory accepts transactions, performs them, and then acknowledges their completion. Multiple memory operations may be in progress at any time and may be initiated in a nondeterministic fashion in any order, provided that all transactions to a location preserve the per-virtual processor partial orderings. Memory transactions may complete in any order. Once initiated, all memory operations are performed atomically: loads from one location all see the same value, and the result of stores is visible to all potential requestors at the same instant.
- The order of memory operations observed at a single location is a *total order* that preserves the partial orderings of each virtual processor's transactions to this address. There may be many legal total orders for a given program's execution.

Memory Model (Cont.)

- Program Execution Model



Processor Model: Uniprocessor System

The SPARC V9 strand model of a virtual processor consists of three units: an Issue Unit, a Reorder Unit, and an Execute Unit, as shown in FIGURE.

Memory Model (Cont.)

- The Issue Unit reads instructions over the instruction path from memory and issues them in *program order to the Reorder Unit.* Program order is precisely the order determined by the control flow of the program and the instruction semantics, under the assumption that each instruction is performed independently and sequentially.
- Issued instructions are collected and potentially reordered in the Reorder Unit, and then dispatched to the Execute Unit. Instruction reordering allows an implementation to perform some operations in parallel and to better allocate resources. The reordering of instructions is constrained to ensure that the results of program execution are the same as they would be if the instructions were performed in program order. This property is called *processor self-consistency*.
- Processor self-consistency requires that the result of execution, in the absence of any shared memory interaction with another virtual processor, be identical to the result that would be observed if the instructions were performed in program order. In the model, instructions are issued in program order and placed in the reorder buffer. The virtual processor is allowed to reorder instructions, provided it does not violate any of the data-flow constraints for registers or for memory.

Memory Model (Cont.)

- The data-flow order constraints for register reference instructions are these:
  1. An instruction that reads from or writes to a register cannot be performed until all earlier instructions that write to that register have been performed (read-after-write hazard; write-afterwrite hazard).
  2. An instruction cannot be performed that writes to a register until all earlier instructions that read that register have been performed (write-after-read hazard).
  - **Compatibility Note**
    - ✓ An implementation can avoid blocking instruction execution in case 2 and the write-after-write hazard in case 1 by using a renaming mechanism that provides the old value of the register to earlier instructions and the new value to later uses.

Memory Model (Cont.)

- The data-flow order constraints for memory-reference instructions are those for register reference instructions, plus the following additional constraints:
  1. A memory-reference instruction that uses (loads or stores) the value at a location cannot be performed until all earlier memory-reference instructions that set (store to) that location have been performed (read-after-write hazard, write-after-write hazard).
  2. A memory-reference instruction that writes (stores to) a location cannot be performed until all previous instructions that read (load from) that location have been performed (write-after-read hazard).

Memory Model (Cont.)

- Hardware Primitives for Mutual Exclusion
  - In addition to providing memory-ordering primitives that allow programmers to construct mutual exclusion mechanisms in software, the Oracle SPARC Architecture provides three hardware primitives for mutual exclusion:
    I. Compare and Swap (CASA and CASXA)
    II. Load Store Unsigned Byte (LDSTUB and LDSTUBA)
    III. Swap (SWAP and SWAPA)
  - Each of these instructions has the semantics of both a load and a store in all three memory models. They are all *atomic*, in the sense that no other store to the same location can be performed between the load and store elements of the instruction. All of the hardware mutual-exclusion operations conform to the TSO memory model and may require barrier instructions to ensure proper data visibility.

Memory Model (Cont.)

1) Compare-and-Swap (CASA, CASXA)
   - Compare-and-swap is substantially more powerful than the other hardware synchronization primitives. It has an infinite consensus number; that is, it can resolve, in a wait-free fashion, an infinite number of contending processes. Because of this property, compare-and-swap can be used to construct wait-free algorithms that do not require the use of locks.
2) Swap (SWAP)
   - SWAP atomically exchanges the lower 32 bits in a virtual processor register with a word in memory. SWAP has a consensus number of two; that is, it cannot resolve more than two contending processes in a wait-free fashion.
3) Load Store Unsigned Byte (LDSTUB)
   - LDSTUB loads a byte value from memory to a register and writes the value FF_{16} into the addressed byte atomically. LDSTUB is the classic test-and-set instruction. Like SWAP, it has a consensus number of two and so cannot resolve more than two contending processes in a wait-free fashion.

Traps

- A trap is a vectored transfer of control to privileged software through a trap table that may contain the first 8 instructions (32 for some frequently used traps) of each trap handler. The base address of the table is established by software in a state register (the Trap Base Address register, TBA. The displacement within the table is encoded in the type number of each trap and the level of the trap. Part of the trap table is reserved for hardware traps, and part of it is reserved for software traps generated by trap (Tcc) instructions.
- A trap causes the current PC and NPC to be saved in the TPC and TNPC registers. It also causes the CCR, ASI, PSTATE, and CWP registers to be saved in TSTATE. TPC, TNPC, and TSTATE are entries in a hardware trap stack, where the number of entries in the trap stack is equal to the number of supported trap levels. A trap also sets bits in the PSTATE register and typically increments the GL register. Normally, the CWP is not changed by a trap; on a window spill or fill trap, however, the CWP is changed to point to the register window to be saved or restored.

Traps (Cont.)

- A trap can be caused by a Tcc instruction, an asynchronous exception, an instruction-induced exception, or an interrupt request not directly related to a particular instruction. Before executing each instruction, a virtual processor determines if there are any pending exceptions or interrupt requests. If any are pending, the virtual processor selects the highest-priority exception or interrupt request and causes a trap.
- A trap behaves like an unexpected procedure call. It causes the hardware to do the following:
  1. Save certain virtual processor state (such as program counters, CWP, ASI, CCR, PSTATE, and the trap type) on a hardware register stack.
  2. Enter privileged execution mode with a predefined PSTATE.
  3. Begin executing trap handler code in the trap vector.
- When the trap handler has finished, it uses either a DONE or RETRY instruction to return.

Traps (Cont.)

- Virtual Processor Privilege Modes
  - An Oracle SPARC Architecture virtual processor is always operating in a discrete privilege mode. The privilege modes are listed below in order of increasing privilege:
    I. Non privileged mode (also known as "user mode")
    II. Privileged mode, in which supervisor (operating system) software primarily operates
    III. Hyper privileged mode (not described in this document)

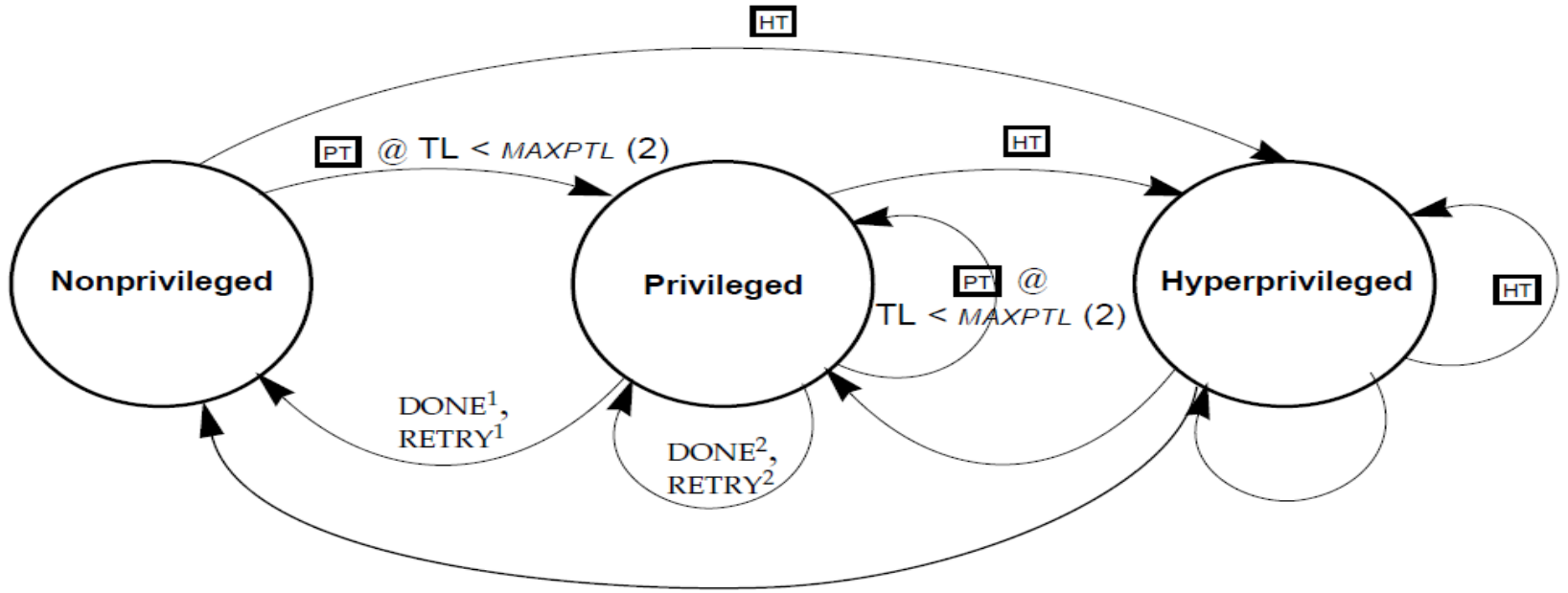
PSTATE.priv	Virtual Processor Privilege Mode
0	Nonprivileged
1	Privileged

The virtual processor's operating mode is determined by the state of two mode bits, as shown in TABLE.

Traps (Cont.)

- A trap is delivered to the virtual processor in either privileged mode or hyper privileged mode; in which mode the trap is delivered depends on:
  - ✓ Its trap type
  - ✓ The trap level (TL) at the time the trap is taken
  - ✓ The privilege mode at the time the trap is taken
- Traps detected in non privileged and privileged mode can be delivered to the virtual processor in privileged mode or hyper privileged mode.
- A trap delivered to privileged mode uses the privileged-mode trap vector, based upon the TBA register.
- The maximum trap level at which privileged software may execute is *MAXPTL*.

Traps (Cont.)



Virtual Processor Privilege Mode Transition Diagram

FIGURE shows how a virtual processor transitions between privilege modes, excluding transitions that can occur due to direct software writes to PSTATE.priv. In this figure, indicates a "trap destined for privileged mode" and indicates a "trap destined for hyper privileged mode".

Traps (Cont.)

❖Uses of the Trap Categories

The SPARC V9 *trap model* stipulates the following:

1. Reset traps occur asynchronously to program execution.
2. When recovery from an exception can affect the interpretation of subsequent instructions, such exceptions shall be precise.
3. In an Oracle SPARC Architecture implementation, all exceptions that occur as the result of program execution are precise.
4. An error detected after the initial access of a multiple-access load instruction (for example, LDTX or LDBLOCKFD) should be precise. Thus, a trap due to the second memory access can occur. However, the processor state should not have been modified by the first access.
5. Exceptions caused by external events unrelated to the instruction stream, such as interrupts, are disrupting. A deferred trap may occur one or more instructions after the trap-inducing instruction is dispatched.

Traps (Cont.)

❖Trap Control

- Several registers control how any given exception is processed, for example:
  - ✓ The interrupt enable (ie) field in PSTATE and the Processor Interrupt Level (PIL) register control interrupt processing.
  - ✓ The enable floating-point unit (fef) field in FPRS, the floating-point unit enable (pef) field in PSTATE, and the trap enable mask (tem) in the FSR control floating-point traps.
  - ✓ The TL register, which contains the current level of trap nesting, affects whether the trap is processed in privileged mode or hyper privileged mode.
  - ✓ PSTATE.tle determines whether implicit data accesses in the trap handler routine will be performed using big-endian or little-endian byte order.

Traps (Cont.)

Trap Control

- Between the execution of instructions, the virtual processor prioritizes the outstanding exceptions, errors, and interrupt requests. At any given time, only the highest-priority exception, error, or interrupt request is taken as a trap. When there are multiple interrupts outstanding, the interrupt with the highest interrupt level is selected. When there are multiple outstanding exceptions, errors, and/or interrupt requests, a trap occurs based on the exception, error, or interrupt with the highest priority (numerically lowest priority number).

Interrupt Handling

- Virtual processors and I/O devices can interrupt a selected virtual processor by assembling and sending an interrupt packet. The contents of the interrupt packet are defined by software convention. Thus, hardware interrupts and cross-calls can have the same hardware mechanism for interrupt delivery and share a common software interface for processing.
- The interrupt mechanism is a two-step process:
  1. Sending of an interrupt request (through an implementation-specific hardware mechanism) to an interrupt queue of the target virtual processor
  2. Receipt of the interrupt request on the target virtual processor and scheduling software handling of the interrupt request
- Privileged software running on a virtual processor can schedule interrupts to *itself* (typically, to process queued interrupts at a later time) by setting bits in the privileged SOFTINT register

Interrupt Handling (Cont.)

Software Interrupt Register (SOFTINT)

- To schedule interrupt vectors for processing at a later time, privileged software running on a virtual processor can send itself signals (interrupts) by setting bits in the privileged SOFTINT register.
- Setting the Software Interrupt Register
  - ✓ SOFTINT{*n*} is set to 1 by executing a WRSOFTINT_SETinstruction (WRasr using ASR 20) with a '1' in bit *n* of the value written (bit *n* corresponds to interrupt level *n*). The value written to the SOFTINT_SET register is effectively **or**ed into the SOFTINT register. This approach allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction.
- Clearing the Software Interrupt Register
  - ✓ When all interrupts scheduled for service at level *n* have been serviced, kernel software executes a WRSOFTINT_CLRinstruction (WRasr using ASR 21) with a '1' in bit *n* of the value written, to clear interrupt level *n* .The complement of the value written to the SOFTINT_CLR register is effectively **and**ed with the SOFTINT register. This approach allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction.

Interrupt Handling (Cont.)

Interrupt Queue Registers

- The active contents of each queue are delineated by a 64-bit head register and a 64-bit tail register.

Interrupt Queue Register ASI Assignments

Register	ASI	Virtual Address	Privileged mode Access
CPU Mondo Queue Head	25_{16} (ASI_QUEUE)	$3C0_{16}$	RW
CPU Mondo Queue Tail	25_{16} (ASI_QUEUE)	$3C8_{16}$	R or RW
Device Mondo Queue Head	25_{16} (ASI_QUEUE)	$3D0_{16}$	RW
Device Mondo Queue Tail	25_{16} (ASI_QUEUE)	$3D8_{16}$	R or RW
Resumable Error Queue Head	25_{16} (ASI_QUEUE)	$3E0_{16}$	RW
Resumable Error Queue Tail	25_{16} (ASI_QUEUE)	$3E8_{16}$	R or RW
Nonresumable Error Queue Head	25_{16} (ASI_QUEUE)	$3F0_{16}$	RW
Nonresumable Error Queue Tail	25_{16} (ASI_QUEUE)	$3F8_{16}$	R or RW

The interrupt queue registers are accessed through ASI ASI_QUEUE (25_{16}). The ASI and address assignments for the interrupt queue registers are provided in TABLE.

Interrupt Handling (Cont.)

- The status of each queue is reflected by its head and tail registers:
  - ✓ A Queue Head Register indicates the location of the oldest interrupt packet in the queue
  - ✓ A Queue Tail Register indicates the location where the next interrupt packet will be stored
- An event that results in the insertion of a queue entry causes the tail register for that queue to refer to the following entry in the circular queue. Privileged code is responsible for updating the head register appropriately when it removes an entry from the queue.
- A queue is *empty* when the contents of its head and tail registers are equal. A queue is *full* when the insertion of one more entry would cause the contents of its head and tail registers to become equal.

Reference

- “Oracle SPARC Architecture 2011", Oracle Corporation. *http://www.oracle.com/technetwork/systems/opensparc*
- "OpenSPARC T2", OpenSPARC Oracle Corporation. *http://www.opensparc.net/opensparc-t2/index.html*
- "Company Info“, Sun Microsystems. *http://sun.com/aboutsun/company/facts.jsp*
- "Oracle Academy“, Oracle Corporation. *https://academy.oracle.com/*
- *Wikipedia Documents (http://wikipedia.org)*
- *Google Images (http://images.google.com)*

Project Contributor

Chaitanya Patel (11BCE069)

Vivek Patel (11BCE073)

Project Advisor

Prof. Ankit Thakkar

Mr. Pushak Raval

“When you know better you do better”.

- Maya Angelou

THANK YOU !!!