MICROPROCESSOR REPORT

THE INSIDERS' GUIDE TO MICROPROCESSOR HARDWARE

VOLUME 7 NUMBER 2 FEBRUARY 15, 1993

SPARC V9 Adds Wealth of New Features

64-Bit Extensions Supplemented by New Instructions, Revamped Trap Model

By Brian Case

The latest revision of the SPARC architecture—version 9—incorporates an impressive number of improvements. V9 makes SPARC a much richer architecture, and as a result, it is a stronger competitor to other RISCs.

In addition to a 64-bit linear address space, 64-bit integers, and the changes needed specifically to accommodate them, V9 adds several new classes of instructions, an enhanced trap model, a revamped privileged register set, and a "relaxed-memory-order" memory model. These changes will require considerable modifications to operating systems, but V9 is upward compatible with V8 for application programs.

SPARC-V9 is the result of work done by SPARC International's architecture committee, and, in the words of the committee, it is designed to be a peer to V8 for higher-performance systems, not a replacement. Although Sun Microsystems contributed significantly to V9 through the efforts of several contributors—including Dave Ditzel as the committee chair—V9 appears to have been influenced by a variety of companies.

The most influential company was undoubtedly HaL Computer Systems, which chose SPARC because HaL was, through SPARC International, able to have considerable influence on its evolution. In effect, HaL chose SPARC for what it could be, not for what it was. HaL probably will be the first with a SPARC-V9 system, and its first systems will use a microprocessor of HaL's own design. A future version of TI's SuperSPARC will implement V9, but it is probably two years from completion.

V9 Overview

The effort to define SPARC-V9 probably began as a response to rumors of 64-bit linear addressing capabilities of competing architectures like MIPS-III (R4000) and Alpha, but V9 is much more than a 64-bit SPARC.

The foundation of the 64-bit architecture is threefold: all integer registers are expanded to 64 bits, an extra set of condition codes records the outcome of ALU operations across all 64 bits, and new conditional branch instructions test the 64-bit condition codes. The V9 documentation claims that there is no "32-bit mode" bit, but this claim is only partially true.

Many of the new instructions in SPARC-V9 go beyond what is required for a modeless 32-bit/64-bit environment. New branches, which can test either the old or new condition codes, also encode a static prediction bit. As in Alpha, there are instructions that perform conditional register-to-register moves. A new class of prefetch instructions allows software to improve performance in sophisticated implementations.

The privileged register set of V9 is dramatically different from that of earlier versions. Some privileged registers are deleted, some registers in V8 that contained two or more fields have been expanded into separate registers, and many new registers have been added. In particular, the windowed register file is now managed by several registers instead of just two.

Trap handling is significantly different in V9. Instead of a single level of trap handlers, at least four levels are available in all implementations. This allows a degree of recursive trapping and improves performance, since trap handlers can process their own work without first taking precautions to prevent all other possible trap conditions. Trap handlers can be coded for the common case of no additional traps, resulting in higher performance. A few new instructions have been added to accommodate these changes.

Privileged Registers

V9 implements a large number of special registers, partly because fields in pre-V9 registers are broken out into separate registers. Figure 1 shows the pre-V9 processor state register (PSR) and many of the V9 special registers. The V8 PSR is replaced by the five V9 registers shown in gray: PSTATE, processor interrupt level (PIL), version (VER), integer condition codes (CCR), and Current Window Pointer (CWP).

The PSTATE register holds several new bits. The MM field selects the memory model used, which determines the ordering for memory references (memory models are discussed below). AM (address mask) is a mode bit that determines whether data and instruction addresses are masked to 32 bits. AG (alternate globals) selects one of two sets of eight global registers.

The version register contains much more information than the V8 PSR field. This register holds the processor's manufacturer number, implementation number, mask revision, maximum number of trap levels, and maximum number of register windows.

The register-window management registers—WSTATE, CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN—help make register window operations more efficient, and are discussed below.

The FP state register has been expanded to 64 bits in a compatible way. The lower 32 bits are unchanged from V8, so a 32-bit program will see what it expects in the low 32 bits of a register after a RDFPSR instruction. The upper 32 bits allow a 64-bit program to access the additional FP condition codes.

Non-Privileged Registers

The non-privileged integer register set is unchanged in SPARC-V9. Eight fixed, global registers are shared by all subroutine contexts, and each subroutine context has access to 24 registers in the windowed register file. As before, the particular 24 registers that are accessed is determined by the Current Window Pointer.

There are two sets of global registers instead of the single set in earlier versions of the architecture, although only one set is active at a time. The AG (alternate global) bit in the PSTATE register determines which set is active. The second set of globals can give trap handlers access to scratch registers without having to disturb the registers of the interrupted process. If the globals of the interrupted process must be accessed, the AG bit can simply be complemented.

In contrast to the integer register set, the floating-point register file has been expanded for non-privileged code. The pre-V9 SPARC architecture specified 32 single-precision floating-point registers that could be combined to form 16 double-precision registers or 8 quad-precision registers. V9 doubles the number of available double- and quad-precision registers with a clever encoding scheme.

Table 1 shows how register numbers in instruction words are mapped into register-file addresses. For single-precision operands, the mapping is as expected. Luckily for the V9 committee, the original SPARC architects required that double- and quad-register operands be aligned within the register file. As a result, the low register-number bit for doubles and the low two register-number bits for quads must be zero in pre-V9 SPARC implementations. This allows V9 to use these bits to encode the high bits of extra register numbers. Note that a future version of SPARC can double the number of quad registers.

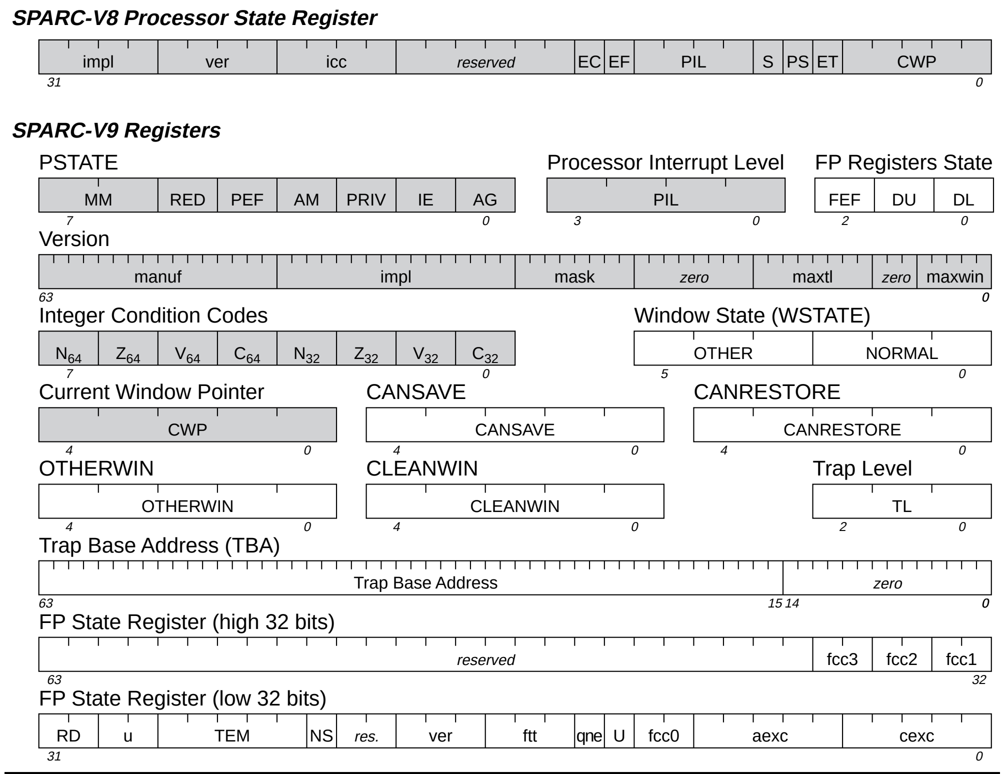

Figure 1. Most of the SPARC V9 special registers. The single V8 PSR (top of figure in gray) is replaced by five separate registers (shown in gray). The FP State register low 32 bits is the same as in V8; the high 32 bits has been added to hold the additional FP condition codes.

There is a growing trend in application and operating-system design toward multi-threading, which requires "light-weight" process switches between threads. With so many FP registers, saving and restoring them all on a thread switch can be time consuming. The FP Registers State register shown in Figure 1 is non-privileged to help software reduce unnecessary FP register saving. DU and DL are "dirty bits" for the upper and lower half of the FP register file, respectively. When a reference is made to an FP register, the appropriate dirty bit is set. On a thread switch, these bits can indicate how many registers to save.

Register Windows

The V9 integer register file implements the same circular buffer of overlapping register sets as in earlier SPARC versions, but the new architecture

Operand Type	6-bit Register-File Address						Register Number Encoded In Instruction Word				
Single	0	ra_4	ra_3	ra_2	ra_1	ra_0	ra_4	ra_3	ra_2	ra_1	ra_0
Double	ra_5	ra_4	ra_3	ra_2	ra_1	0	ra_4	ra_3	ra_2	ra_1	ra_5
Quad	ra_5	ra_4	ra_3	ra_2	0	0	ra_4	ra_3	ra_2	0	ra_5

Table 1. FP register mapping for the three different precisions (ra_x = register address bit x). This mapping from the actual bits encoded in the instruction (right side) to the bits used to address the register file (middle) allows programs to access 32 single-precision registers, 32 double-precision registers, or 16 quad-precision registers.

has significantly changed the hardware support for the windows to increase performance in environments where process switching is frequent. In typical UNIX workstation environments, register windows are a net win because the performance boost during process execution more than compensates for the extra time spent swapping registers when a process switch is required.

In other environments, such as transaction processing, process switching can be very frequent. With frequent switches, the time spent saving registers can limit performance because a process often needs to execute only a small number of instructions before relinquishing control of the processor. The result is more time spent switching processes than executing them. One solution is to leave some or all of the register contents for a deactivated process valid in the register file, but this requires isolating the windows of the old process from those of the new process.

Another performance-robbing situation occurs in secure environments when a process is finished executing and exits. In secure environments, it is necessary to clean (zero) the windows of the old process before allowing the new one to use the windows to make sure that the new process has no access to sensitive data. Unfortunately, clearing all the registers can be nearly as time consuming as saving them.

The solution in each case is to defer the work of saving or clearing registers until it is absolutely necessary. Deferring the work requires remembering which windows belong to the old process and which to the new.

Register-file sharing is facilitated in V9 by five, 5-bit special support registers: CWP, CANSAVE, CANRESTORE, OTHERWIN, and CLEANWIN. These registers keep track of which windows are available to the currently running program and which have some other program's data. Figure 2 shows an eight-window register file and possible values of the support registers.

CANSAVE and CANRESTORE tell how many consecutive subroutine calls and returns, respectively, can be done before causing a spill or fill trap. As before, the SAVE and RESTORE instructions, respectively, are used to enter and exit subroutines. (These instructions are not new to V9, but their semantics have been changed to manipulate the new support registers.) SAVE increments CWP and CANRESTORE and decrements CANSAVE; RESTORE does the opposite.

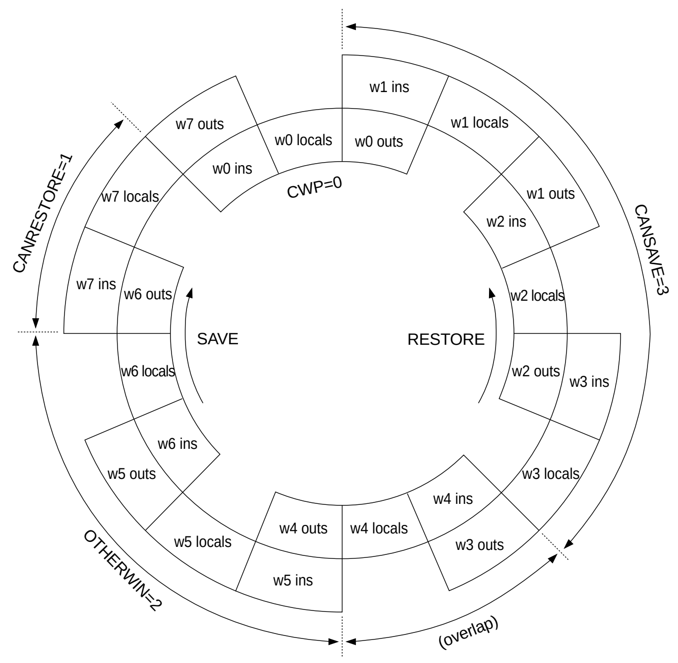

Figure 2. Register windows and possible values for window management special registers. In this example, two windows have data for a deactivated but live process (OTHERWIN=2). The current window is zero, and one RESTORE (subroutine return) or three consecutive SAVEs (subroutine calls) can be executed without causing a spill or fill trap (since CANRESTORE=1 and CANSAVE= 3).

In V8, the window-mask register held a bit for each window; if a bit was set, the corresponding window was unavailable to the current process. When CWP equalled the number of a bit that was set, a spill or fill trap occurred depending on whether the SAVE or RESTORE instruction caused CWP to equal that number. In V9, register spill and fill trap routines are invoked when a SAVE is executed and CANSAVE is zero or when a RESTORE is executed and CANRESTORE is zero.

OTHERWIN keeps track of which windows have valid data belonging to a different process. If OTHERWIN is not zero when a spill or fill occurs, the window to be spilled or filled contains valid data for a different process. The WSTATE register shown in Figure 1 holds two 3-bit fields that select the appropriate trap handler for each situation (also see Figure 4).

CLEANWIN (not shown in Figure 2) comes into play when a window that belongs to a dead process needs to be allocated for a SAVE. When CLEANWIN is equal to CANRESTORE during a spill trap, a clean_window trap is taken instead of a regular spill trap. The clean_window handler does not need to save the window but only to zero the registers to prevent a nosy process from getting access to data from a dead process.

Preventing access to the register contents of a dead process is important for absolute security in commercial,

Old Instructions With Modified Behavior	
FCMP	FP compare
FCMPE	FP compare, exception if unordered
LDFSR, STFSR	Load/store low 32 bits of FP status register
RDASR, WRASR	Read/write ancillary state register
LDUW	Load unsigned word (same as LD in V8)
SETHI	Set high 22 bits, clear upper 32 bits
Ticc	Trap on either 32-bit or 64-bit int. cond. codes
64-bit Addressing & Shift	
LDSW[A]	Load signed 32-bit word [alternate address space]
LDX[A]	Load 64-bit word [alternate address space]
STX[A]	Store 64-bit word [alternate address space]
LDXFSR	Load all 64 bits of Floating Status Register
STXFSR	Store all 64 bits of FSR
LDFA	Load floating single, alternate address space
LDDFA	Load floating double, alternate address space
STFA	Store floating single, alternate address space
STDFA	Store floating double, alternate address space
SLLX	64-bit shift left logical
SRAX	64-bit shift right logical
SRLX	64-bit shift right arithmetic
New FP Operations	
F[sdq]TOx	Convert FP to 64-bit integer
FxTO[sdq]	Convert 64-bit integer to FP
FMOV[dq]	FP move, double & quad
FNEG[dq]	FP negate, double & quad
FABS[dq]	FP absolute value, double & quad
Support For New Trap Model	
DONE	Return from trap, skip trapped instruction
RETRY	Return from trap, retry trapped instruction
RETURN	Return from user trap handler (branch + RESTORE)
SAVED	Adjust register window control registers
RESTORED	Adjust register window control registers
RDPR, WRPR	Read/write privileged registers
Support For Higher Performance	
BPcc	Predicted branch on integer condition code
BPr	Predicted branch on integer reg. contents
MOVcc	Move integer register on condition code
MOVr	Move int. register on int. register contents
FBPfcc	Predicted branch on FP condition code
FMOVcc	Move FP register on int. condition code
FMOVr	Move FP register on int. register contents
CASA	Compare and swap 32-bit, alternate address space
CASXA	Compare and swap 64-bit, alternate address space
MULX	Generic 64-bit multiply
SDIVX	Signed 64-bit divide
UDIVX	Unsigned 64-bit divide
FLUSHW	Flush all register windows
POPC	Population count of 64-bit value
LDQF[A]	Load quad-precision FP [alternate address space]
STQF[A]	Store quad-precision FP [alternate address space]
PREFETCH[A]	Prefetch data [alternate address space]
MEMBAR	Memory barrier
Deleted instructions	
All coprocessor loads and stores	
RDTBR, WRTBR	TBR no longer exists
RDWIM, WRWIM	WIM no longer exists
RDPSR, WRPSR	PSR no longer exists
RETT	Return from trap (DONE/RETRY instead)
STDFQ	Store double from FP queue (RDPR FQ instead)

Table 2. New or changed instructions in SPARC-V9.

"big system" environments where, for example, financial data might be manipulated and relatively public access is allowed. The window management of V9 provides this security without significant degradation of process-switch latency.

64-Bit Architecture

The SPARC architecture makes the transition from 32 bits to 64 bits in much the same way that the MIPS architecture did: all integer registers are widened to 64 bits and all integer ALU computations operate on all 64 bits. Programs written for the 32-bit pre-V9 versions of SPARC will still operate correctly because of the nice property of two's-complement arithmetic: a 64-bit ALU computes the low 32 bits the same as a 32-bit ALU.

One difference between 64-bit MIPS-III and SPARC-V9 is that the MIPS-III architecture has mode bits that enable the new 64-bit instructions; if a mode bit is not set, the new instructions cause illegal-instruction traps as they would on prior versions of the MIPS architecture. In contrast, the new SPARC-V9 instructions are always enabled.

For this reason, the SPARC-V9 specification claims that there is no 32-bit/64-bit mode bit. This is not quite true, however, because the AM (address mask) bit in the PSTATE register is intended to be set for applications (i.e., pre-V9 binaries) that use 32-bit addressing. (Another approach is to set up the MMU mapping tables to have the same effect, but the AM bit is a far better solution.) The AM bit causes the upper 32 bits of data and instruction addresses to be masked to zero before they are sent to the MMU and caches.

V9 adds several new instructions to implement 64-bit functions within the context of the 32-bit base SPARC architecture. Table 2 shows a list of all the major instruction-set changes for V9.

The new instructions that specifically address the 64-bit architecture features are loads, stores, shifts, and the new conditional branches. Even though SPARC already had load/store double-word instructions, the semantics for these instructions specified a pair of 32-bit registers. Since all the old semantics must be maintained, new loads and stores for 64-bit values in a single register (LDX[A], STX[A]) and for sign-extending a 32-bit value to 64 bits (LDSW) have been added.

As with the MIPS-III and PowerPC 64-bit extensions, V9 had to add new 64-bit shift instructions. SRLX and SRAX operate on all 64 bits of a register. (SLLX has exactly the same effect as SLL: it shifts all 64 bits left.)

Although all the old conditional branches are still implemented for compatibility, the V9 specification discourages their use in new software. Of course, the old branches test only the 32-bit condition codes. All the new conditional instructions (BPcc) can test either the 32-bit or 64-bit condition codes, and then encode a static prediction bit as well. The prediction bit helps boost performance with a very small hardware cost. More costly, dynamic techniques are still possible.

The cost for the added features is a reduction in the size of the branch displacement, as shown in Figure 3. The cc field determines whether the instruction tests the old 32-bit or the new 64-bit condition codes.

One interesting detail is that there are no 64-bit "with-carry" versions of the add and subtract instructions. Thus, while algorithms that implement multiple-precision integer arithmetic cannot benefit from the 64-bit architecture, this should be of little concern since such programs are probably rare.

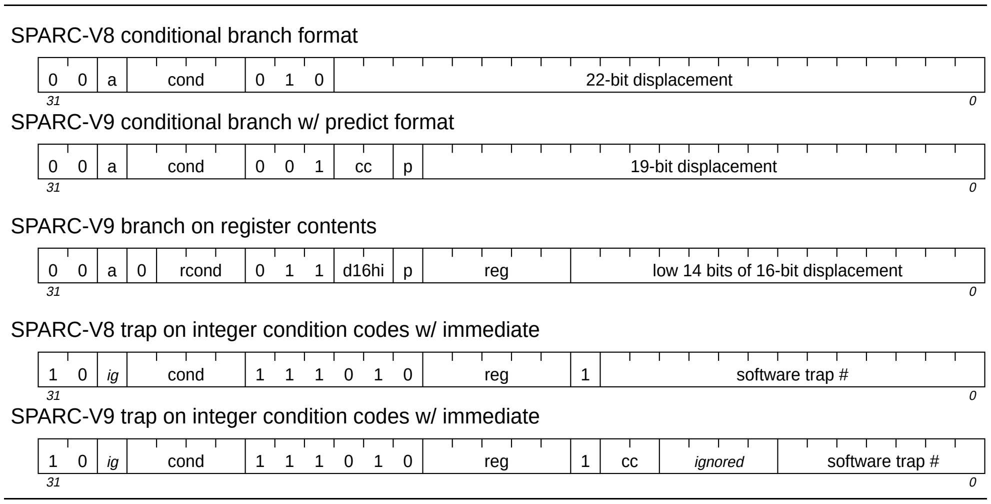
SPARC-V8 conditional branch format
0 0 a cond 0 1 0 22-bit displacement
31 0
SPARC-V9 conditional branch w/ predict format
0 0 a cond 0 0 1 cc p 19-bit displacement
31 0
SPARC-V9 branch on register contents
0 0 a 0 rcond 0 1 1 d16hi p reg low 14 bits of 16-bit displacement
31 0
SPARC-V8 trap on integer condition codes w/ immediate
1 0 ig cond 1 1 1 0 1 0 reg 1 software trap #
31 0
SPARC-V9 trap on integer condition codes w/ immediate
1 0 ig cond 1 1 1 0 1 0 reg 1 cc ignored software trap #
31 0

Figure 3. Some SPARC instruction formats. The top compares old conditional branches to the new conditional branches, which encode condition-code selection (cc) and prediction (p). The middle shows the branch-on-register format, which has an even shorter displacement. The bottom shows the old and new trap formats; the placement of the cc field creates the possibility of a compatibility problem (see text).

New Instructions & Semantics

Several new categories of instructions have been added to SPARC-V9 that are not a direct result of the 64-bit architecture.

To allow optimizing compilers to eliminate conditional branches in simple code sequences, V9 offers conditional move instructions for both integer and floating-point registers. Conditional moves must be proving their worth because they have shown up in new architectures including Alpha, PowerPC, and TFP (Silicon Graphics' high-end MIPS processor) *(see **070202.PDF**)*. These instructions help reduce pipeline bubbles by eliminating some branches. The move can be based either on integer condition codes or on the value in an integer register. If the value of a register is used, only comparisons against zero are available: equal-to zero, less-than-or-equal-to zero, less-than zero, not zero, greater-than zero, and greater-than-or-equal-to zero.

A new set of conditional-branch-on-register instructions eliminates the need for an explicit compare instruction in many common cases. These instructions use the same set of six "against-zero" conditions used by the conditional-move-on-register instructions. As shown in Figure 3, these instructions sacrifice some offset bits for the register field and predict bit.

The population-count instruction is a primitive not found in other RISC architectures. This function is useful for cryptography and compression. Without this instruction, a population count requires a loop or sequence of instructions that is an order of magnitude slower. In addition, population count enables the "find-first-one" function—common in other RISCs—to be implemented with only four instructions, which is reasonably fast.

The prefetch instructions allow software and hardware to work together to improve performance. When a program can anticipate the addresses of data it will soon be referencing, it can issue prefetch instructions for those addresses. Armed with the addresses, hardware can bring the data into cache before its expected use. There are several variants of the prefetch instruction so that a program can give hardware a good idea of how the data will be used. These variants are prefetch for several reads, prefetch for one read, prefetch for several writes, prefetch for one write, prefetch page, and "implementation-dependent" prefetch.

For example, a prefetch-for-several-reads instruction might be used for a record in a data base, while a prefetch-for-one-read instruction might be used for a long array of data in a scientific calculation. Prefetch instructions allow TLB and cache miss processing to start early; then, when the data is actually accessed, it is either already available or it will be available sooner than it would have been without the prefetch instruction. Thus, even though a prefetch instruction adds to the instruction stream length and costs an instruction "opportunity" (only a fraction of a cycle in a superscalar machine), it can have a net savings of many cycles in overall program execution time.

For instruction prefetching, the branch-never-with-prediction instruction is used. This instruction never causes a transfer of control, but compiler writers and processor implementors are told to use it as a hint of a possible upcoming branch with the same target.

One change in instruction semantics from V8 to V9 creates a possible incompatibility for old binaries. The trap-on-integer-condition-codes instruction (Tcc) with the trap number specified in an immediate field is shown in

Figure 3. In V8, the format specified a 13-bit trap number even though the hardware only used the lower seven bits. In V9, the upper two bits of that immediate field have been redefined to encode which set of integer condition codes to use.

If by some chance a V8 program used a negative software trap number in a Tcc instruction (which could be used to convey information to the trap handler), it would cause a V9 processor to use the 64-bit condition codes instead of the 32-bit condition codes, which would likely result in incorrect program execution.

The V9 architects have found no occurrences of such an incompatible Tcc instruction in any programs, and in fact, they are forbidden in the SPARC ABI, but this is exactly the kind of incompatibility that architects typically regret.

Traps

On pre-V9 SPARC implementations, traps are handled by allocating a new register window and saving the PC and next PC in two registers of that window; other critical state bits are saved in shadow copies in the PSR. In V9, state is saved on small internal stacks of shadow registers, and no register window movement occurs.

There are four multi-level stacks of shadow registers that save the two program counters, the trap type, and trap state (condition code register, ASI register, PSTATE register, and CWP register). The value in the TL register (see Figure 1) determines which of the shadow copies in each stack is active and accessible. Although four is the minimum, the three-bit TL register allows a V9 implementation to implement up to seven interrupt levels (level zero means normal, non-trap-mode execution).

When a trap occurs at the next to last level or when any of the four possible software and hardware resets occurs, the processor enters the "RED" mode (Reset, Error, Debug). If a trap occurs while the processor is at the maximum trap level, the processor enters Error mode and halts.

Traps in V9 are vectored through one of two tables depending on whether TL is zero or not. Figure 4 shows how a table vector entry is formed at trap time. Because the low five bits of the vector address are zero, each table entry is eight instructions long (table entries in V8 have only four instructions). The trap type selects one of 512 entries, while the value of TL at trap time determines which of the two tables will be used. The high 49 bits of the Trap Base Address special register determine the location of the tables in main memory.

The trap type for register window spill and fill activity is determined in a special way, as shown at the bottom of Figure 4. The low two bits of the trap type are always zero; this gives all spill and fill handlers room for 32 instructions within the table, which is often enough to implement the entire routine. As explained earlier, since part of the trap vector is determined by the state of OTHERWIN, the register file can be shared easily without requiring the spill/fill handlers themselves to check the ownership of a window.

In SPARC-V8, trap routines were exited with the RETT instruction. In V9, that instruction is replaced with DONE and RETRY. The RETT instruction was used in the delay slot of a branch to effect a proper return, but the DONE and RETRY instructions perform all necessary state restoration atomically. DONE is used to return from a handler that needs to skip past the trapping instruction while RETRY is used to return to the trapping instruction.

Memory Models

With simple RISC pipelines and caches, it is natural for an implementation to issue and complete memory references in exactly the order specified in a program. For sophisticated superscalar processors with write buffers and non-blocking caches, on the other hand, constraining memory references to complete in program order—using a so-called strongly ordered memory model—can waste significant available performance by preventing the exploitation of parallelism. For this reason, there is a trend in new architectures (e.g., Alpha) toward weakly ordered memory models.

SPARC-V8 specified two memory models: total store ordering (TSO) and partial store ordering (PSO). SPARC-V9 adds a third, relaxed memory order (RMO). The model in force at a given time is determined by the MM field in the PSTATE register (see Figure 1).

RMO essentially lets a processor re-order memory references as it sees fit as long as self-consistency is not violated, i.e., within a given processor, a load will see any previously stored values to the same address.

The benefit of RMO is that an implementation is allowed to re-order the completion of memory references to take full advantage of superscalar execution, write buffers, and non-blocking caches. With weak ordering, a superscalar processor can group memory references for simultaneous execution with much greater freedom. If a store misses in the cache, a simultaneously grouped load will be able to complete anyway because it can get its value from the store buffer (if the store and load have the same address) or from the non-blocking cache (if the store and load have different addresses). Thus, weakly ordered memory models are important for the full exploitation of the sophisticated implementations that increasing transistor budgets will allow and encourage.

V9 provides both the V8 STBAR (store barrier) and the new MEMBAR (memory barrier) instructions to allow programs to temporarily enforce stronger ordering (MEMBAR uses the same opcode as STBAR and so is just a generalization of it). Stronger ordering is required for interacting with I/O devices and for multiprocessor synchronization. The STBAR instruction simply forces the completion of all previous stores before a subsequent store can complete, but the

MEMBAR instruction is much more versatile. It allows a program to encode the specific kind of temporary memory ordering it needs. For example, a MEMBAR instruction can specify that all loads issued prior to the MEMBAR instruction complete before any subsequent stores; this could be used under TSO to force temporary strong ordering.

Miscellaneous Changes

There are so many subtle implications of V9 it is impossible to mention all the changes here, but a few more are worth noting. As shown in Figure 1, the FP state register has been extended with three extra sets of FP condition codes. They allow FP programs to compute comparisons ahead of the time they will be used and are reminiscent of the multiple condition codes implemented in IBM's POWER architecture.

In V8, the load and store instructions that use address space identifiers are all privileged. In V9, half of the ASIS (0x80–0xFF) are now available to non-privileged programs.

On previous SPARC systems, interrupts with priority level 15 were non-maskable. V9 makes them maskable as well so that only catastrophic resets are not maskable.

Conclusions

SPARC-V9 has so many enhancements to V8 that it almost seems like a different architecture. In absorbing such dramatic changes, SPARC has attained the richness of other high-end architectures such as Alpha.

While a committee may not be a suitable vehicle for the original definition of an architecture, V9 is an example that committees can successfully, and in a timely manner, evolve architectures. PowerPC is another example.

This evolution of the SPARC architecture is happening now for several reasons. First, features such as conditional moves are motivated by developments in superscalar and superpipelined implementation where branches can significantly reduce performance. Second, there is a trend toward bigger systems with multiple processors where 64-bit linear addressing and weakly ordered memory models can significantly improve performance. Third, the technology constraints—such as 10K gate arrays—that influenced the original SPARC architecture are a dim memory. Increasing transistor budgets encourage out-of-order, superscalar processors and large, non-blocking caches. Judging by the enhancements in V9, the committee clearly had big systems and aggressive microprocessors in mind.

While it is conceivable that RISC detractors will say that V9 makes SPARC "CISCier," this is not true. The new features satisfy the basic RISC criteria, e.g., the instructions are fixed-length, have simple, consistent encodings, and implement a load/store architecture. ♦

Next issue, we'll compare SPARC-V9 to other 64-bit architectures (see ***070302.PDF****).*

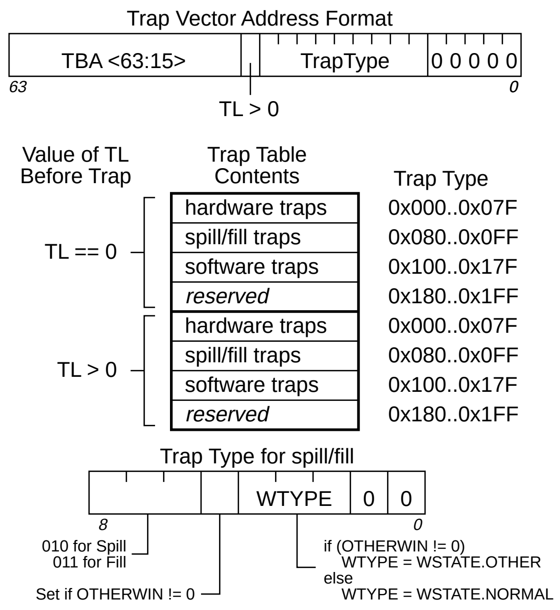

Figure 4. Trap vector formation and trap table layout. The vector address is a combination of the trap base address, a bit set on whether or not the current trap level is greater than zero, and the trap type. There are two trap tables, one for traps that occur when trap level is equal to zero and one for traps when TL is greater than zero. The trap type for register window spill and fill is determined specially to create different vectors for different situations.