10.9 Instructions unique to SPARC v9

(Original section[1])

Several features are unique to SPARC

Register windows

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer, providing unlimited depth. The knee of the cost/performance curve seems to be six to eight banks.

SPARC can have between two and 32 windows, typically using eight registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given that each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions `SAVE` and `RESTORE`. `SAVE` is used to "save" the caller's window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller's window of the addition operation, while the destination register is in the callee's window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. `RESTORE` is the inverse of `SAVE`, bringing back the caller's window while acting as an add instruction, with the source registers from the callee's window and the destination register in the caller's window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee's final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) have been built in several technologies since 1987. For several generations, the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache access times dominate register access times in these implementations. The current-generation machines took different implementation strategies—in order versus out of order—and it's unlikely that the number of registers by themselves determined the clock rate in either machine. Recently, other architectures have included register windows: Tensilica and IA-64.

Another data transfer feature is alternate space option for loads and stores. This simply allows the memory system to identify memory accesses to input/output devices, or to control registers for devices such as the cache and memory management unit.

Fast traps

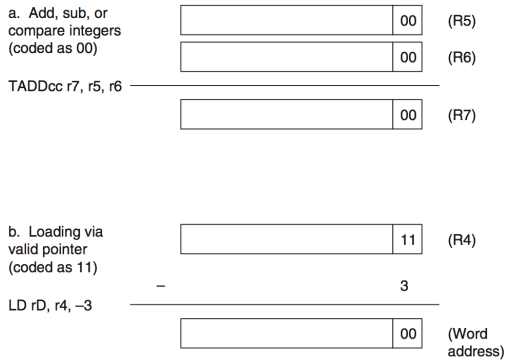
Version 9 SPARC includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or misaligned stack pointers explicitly in the code, thereby making the handler faster. Two new instructions were added to return from this multilevel handler: `RETRY` (which retries the interrupted instruction) and `DONE` (which does not). To support user-level traps, the instruction `RETURN` will return from the trap in nonprivileged mode.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multiword arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and, hence, comparison. The two least significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. The figure below shows both types of tag support.

Figure 10.9.1: SPARC uses the two least significant bits to encode different data types for the tagged arithmetic instructions (COD Figure D.9.1).

a. Integer arithmetic takes a single cycle as long as the operands and the result are integers. b. The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of -3 can be used to access the even word of a pair (CAR) and +1 can be used for the odd word of a pair (CDR).



Overlapped integer and floating-point operations

SPARC allows floating-point instructions to overlap execution with integer instructions. To recover from an interrupt during such a situation, `SPARC` has a queue of pending floating-point instructions and their addresses. `RDPR` allows the processor to empty the queue. The second floating-point feature is the inclusion of floating-point square root instructions `FSQRTS`, `FSQRTD`, and `FSQRTQ`.

Remaining instructions

The remaining unique features of SPARC are as follows:

- `JMPL` uses `Rd` to specify the return address register, so specifying `r31` makes it similar to `JALR` in `MIPS` and specifying `r0` makes it like `JR`.
- `LDSTUB` loads the value of the byte into `Rd` and then stores `FF16` into the addressed byte. This version 8 instruction can be used to implement synchronization (see COD Chapter 2 (Instructions: Language of the Computer)).
- `CASA` (*`CASXA`*) atomically compares a value in a processor register to a 32-bit (64-bit) value in memory; if and only if they are equal, it swaps the value in memory with the value in a second processor register. This version 9 instruction can be used to construct wait-free synchronization algorithms that do not require the use of locks.
- `XNOR` calculates the exclusive OR with the complement of the second operand.
- `BPcc`, `BPr`, and `FBPcc` include a branch prediction bit so that the compiler can give hints to the machine about whether a branch is likely to be taken or not.
- `ILLTRAP` causes an illegal instruction trap. Muchnick [1988] explains how this is used for proper execution of aggregate returning procedures in C.
- `POPC` counts the number of bits set to one in an operand, also found in the third version of the Alpha architecture.
- *Nonfaulting loads* allow compilers to move load instructions ahead of conditional control structures that control their use. Hence, nonfaulting loads will be executed speculatively.
- *Quadruple-precision floating-point arithmetic and data transfer* allow the floating-point registers to act as eight 128-bit registers for floating-point operations and data transfers.
- *Multiple-precision floating-point results for multiply* mean that two single-precision operands can result in a double-precision product and two double-precision operands can result in a quadruple-precision product. These instructions can be useful in complex arithmetic and some models of floating-point calculations.

(*1) This section is in original form.