Register Window

Architecture Comparison of MIPS, ARM and SPARC

8.1.5. An Example RISC Architecture: SPARC

In 1987, Sun Microsystems announced an open RISC architecture, called **SPARC (Scalable Processor ARChitecture)**, that would be the basis of future Sun products (e.g., Sun-4 series and SPARCstation). SPARC is an *architecture*, not a chip. The SPARC architecture reference manual describes what the machine looks like to the assembly language programmer or compiler writer, but does not specify how it is implemented. Sun also defined a standard memory management unit for use with SPARC chips.

About half a dozen semiconductor vendors were then licensed to produce SPARC chips using different technologies (CMOS, ECL, GaAs, gate array, custom VLSI, etc.). The intention was to encourage competition among chip vendors, in order to improve performance, reduce prices, and make an attempt at establishing the SPARC architecture as an industry standard.

For our purposes, SPARC is an interesting example because it is closely based on the pioneering RISC I, RISC II, and SOAR work of Patterson and Séquin at Berkeley. In this section we will describe the SPARC in some detail, starting with an overview, and then covering the registers, instructions, floating-point unit, interrupts, and memory management. After that we will give a simple example program for the SPARC. We will conclude by comparing the SPARC to the 80386 and 68030.

Technical Overview of the SPARC

The SPARC definition includes not only the CPU, called the **IU (Integer Unit)**, but also the **FPU (Floating Point Unit)** and an optional user-supplied **CP (CoProcessor)**. In addition, most SPARC-based systems will have also a memory management unit and cache, as illustrated in Fig. 8-14.

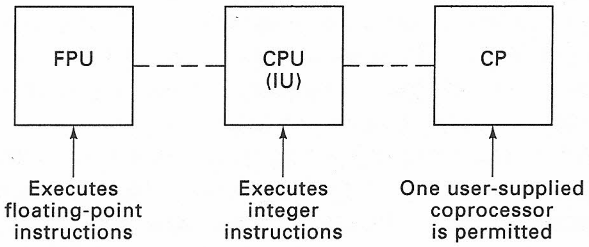
FPU
CPU (IU)
CP
Executes floating-point instructions
Executes integer instructions
One user-supplied coprocessor is permitted

Fig. 8-14. A SPARC system may have an IU, FPU, and coprocessor.

The SPARC is basically a 32-bit design. It has a (paged) linear address space, consisting of 2^{32} individually addressable 8-bit bytes. Words are 32 bits long and must be aligned on word boundaries (i.e., addresses that are multiples of four). This alignment requirement not only improves performance, but allows certain useful optimizations in the instruction set. Memory is big endian, like the 680x0 family, with byte 0 on the left-hand (high-order) end of a 32-bit word.

All instructions and registers are 32 bits, even the floating-point registers. However, instructions are provided for loading and storing 8-, 16-, 32-, and 64-bit quantities into the 32-bit registers, the latter using two consecutive registers. Like other RISC machines, the SPARC is a LOAD/STORE architecture, so all operations take place on operands located in the 32-bit registers.

The SPARC is itself a uniprocessor architecture, but provision has been made for connecting up multiple SPARC chips to form a multiprocessor. Special instructions have been included for multiprocessor synchronization, for example.

The SPARC architecture has been carefully specified to allow for highly pipelined implementations. Among other aspects, it defines delayed loads, stores, branches, calls, and returns. A typical implementation has a four-stage pipeline, as shown in Fig. 8-15. During the first cycle, the instruction word is fetched from memory. During the second, it is decoded. During the third, it is executed. Finally, during the fourth, the results are written back.

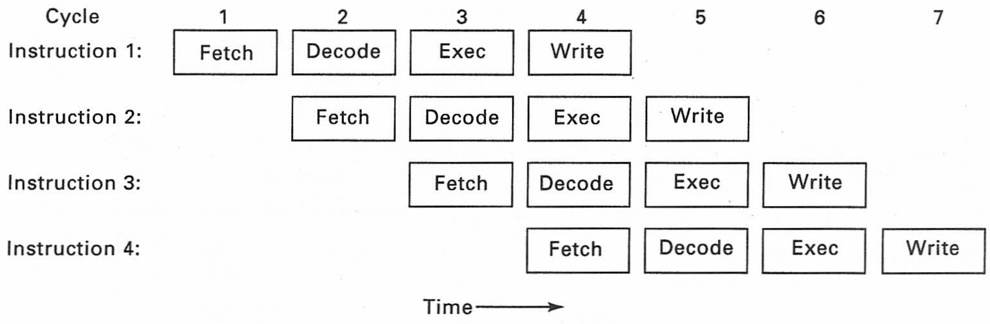

Fig. 8-15. Four-stage pipelining.

SPARC Registers

The SPARC has an overlapping register window scheme, very similar to the one described in Sec. 8.1.3. At any instant, 32 32-bit registers are visible. A *CWP* variable in the hardware points to the current set. The total size of the register file is not part of the architecture, allowing more registers to be added as the technology improves, up to a maximum of 32 windows (*CWP* is 5 bits). A maximum-length register file is then 32 × 16 for the windows plus eight for the globals, for a maximum of 520 registers. The initial implementations have about one-quarter of that.

The SPARC registers are shown in Fig. 8-16. The registers are numbered in the reverse order from Fig. 8-8 because *CWP* is *decremented* rather than *incremented* when a procedure is called. Thus the calling procedure puts the parameters in R8 through R15, and these registers become R24 through R31 in the called procedure. In other words, when a procedure is called, the window slides "upwards" rather than "downwards."

Some of the SPARC registers have specific functions, as shown in Fig. 8-16. All of them have alternative names, which are used by the compilers and assembly language programmers. G0 is hardwired to 0. Stores into it do not change its value. G1 through G7 are global, and may contain integer variables, pointers to tables, or other important data items.

O0 through O7 are the output registers, used by procedures to pass parameters to procedures being called. The first parameter goes in O0, the next in O1, and so on. O6 (SP) is used as a pointer to the memory stack. The stack is used for excess parameters, windows that have been spilled into memory due to register file overflow, dynamically allocated stack space, saved floating-point registers, pointers

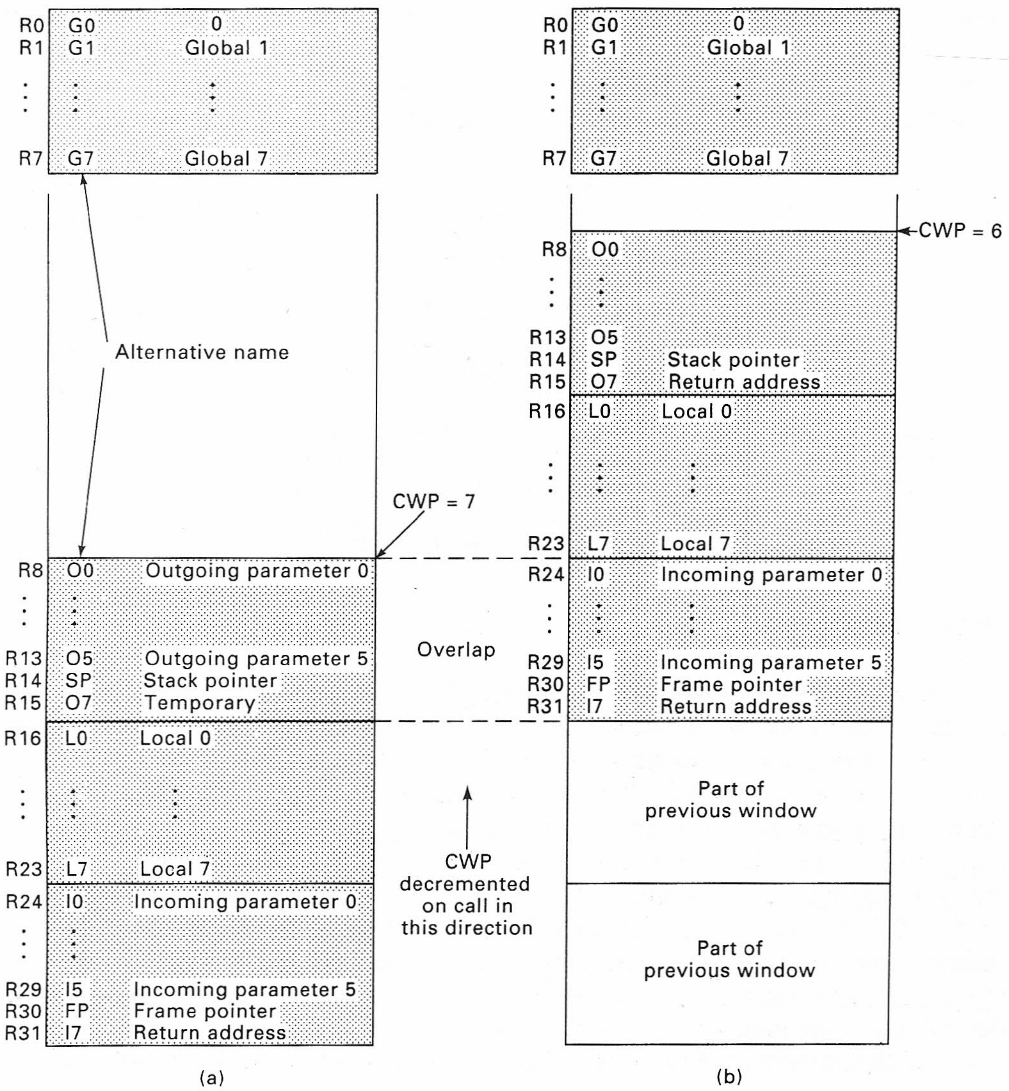

Fig. 8-16. SPARC registers. (a) Before call. (b) After call.

to buffers where called procedures can return structures and arrays, and so on. The CALL instruction deposits the return address in O7.

The eight registers for local variable, L0 through L7, can be used any way the programmer or compiler sees fit. The eight input variables, I0 through I7, are the parameters passed to the current procedure by its caller. Unused registers may be used for additional local variables. I6 (FP) is the frame pointer and is used to address variables in the stack frame. Unlike SP, which may change as the procedure executes, FP points at the same memory word during the entire procedure execution, thus making it more suitable for indexing from than SP. I7 contains the address to return to when the procedure has finished.

When a program first starts up, all the register windows are available for it to use. As procedures are called, windows are used up. Suppose that after a while, the program is so deep in calls that it has managed to use up all *n* windows. If another call has to be made, the oldest register window has to be saved to the stack. The question arises how the computer knows that all the windows are in use.

The solution lies in a special register, the **WIM (Window Invalid Mask)** that is visible only to the kernel. The WIM has one bit per window. When the *CWP* is decremented to advance to a new window, the hardware checks to see if the WIM bit for the new window is set. If so, a trap occurs. The trap handler then saves the window. In this manner, no software checking is needed on each call.

? WIM

Normally, the WIM contains all 0 bits except for a single 1 bit marking the oldest register window currently in use. When that window is reached, a trap occurs, the oldest register window is saved on the stack, and the WIM is rotated one bit, to mark the next lowest window as the oldest.

A nontrivial amount of overhead is incurred when the windows wrap around. A trap occurs, registers must be saved, and the WIM must be updated. It is therefore useful to restrict window usage wherever possible.

One optimization that many SPARC compilers make is handling **leaf procedures** in a special way. A leaf procedure is one that does not call any other procedures (i.e., is a leaf of the call graph). If a leaf procedure can live with 6 registers for input parameters and locals combined, the compiler can cheat and have it use what is left of O0 through O5 for locals.

When this optimization is used, the caller puts the parameters in O0 through O5, as usual, and then issues a normal CALL, which deposits the return address in O7. The SAVE instruction, which is what actually advances the window, is omitted. The leaf procedure runs in the usual way, except that its local variables go in O5, O4, and so on, instead of in L0 through L7. When it is done, it jumps indirectly through O7, but does not execute the RESTORE instruction that normally increments *CWP*. Measurements have shown that something like 40 percent of all procedures are leaf procedures, so this optimization is frequently applicable.