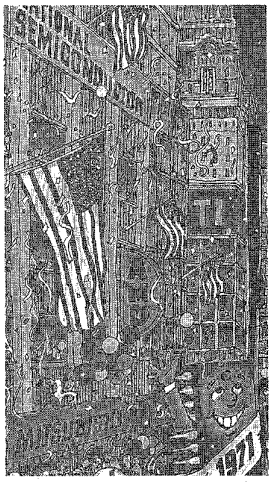
UltraSparc I: A Four-Issue Processor Supporting Multimedia

Marc Tremblay

J. Michael O'Connor

Sun Microelectronics



Issuing four instructions per cycle, this workstation/server processor uses the Visual Instruction Set and a nonblocking memory system to accelerate multimedia applications.

UltraSparc I is a second-generation superscalar processor. It is a high-performance, highly integrated, four-issue superscalar processor based on the Sparc Version 9 64-bit RISC architecture.[1] We have extended the core instruction set to include graphics instructions that provide the most common operations related to two-dimensional image processing; two- and three-dimensional graphics and image compression algorithms; and parallel operations on pixel data with 8-, 16-, and 32-bit components. Additional, new memory access instructions support the very high bandwidth requirements typical of graphics and multimedia applications.

A 200-MHz UltraSparc with a 2-Mbyte external cache delivers an estimated 322 SPECint92 and 462 SPECfp92. UltraSparc can decode broadcast-quality MPEG-2 streams and move data to or from main memory at a rate of 1.6 Gbytes/s peak or 700 Mbytes/s sustained.

UltraSparc can sustain the execution of up to four instructions per cycle, even in the presence of conditional branches and cache misses. This is due mainly to the decoupled aspect of the units feeding instructions and data to the rest of the pipeline. Those instructions predicted to execute issue in program order to multiple functional units, execute in parallel, and, for added parallelism, can complete out of order. To further increase the number of instructions executed per cycle, instructions from two basic blocks (that is, instructions before and after a conditional branch) can issue in the same group.

The UltraSparc die (Figure 1) includes

- a 16-Kbyte, pseudo-two-way set-associative instruction cache;
- a 64-entry, fully associative instruction translation look-aside buffer (TLB);
- a 16-Kbyte, direct-mapped write-through data cache;
- a 64-entry, fully associative data translation look-aside buffer;
- nine functional units;
- a nine-entry-deep load buffer;
- an eight-entry-deep store buffer; and
- logic to control the 144-bit external bus.

The first implementation of UltraSparc uses a 0.5-micron process with four layers of metal, resulting in a 310-mm^2 die composed of 5.2 million transistors. The 521-pin chip typically dissipates 28W at 167 MHz.

Microarchitecture

UltraSparc's design dedicates only 1.6 million transistors to the basic data arrays of the 16-Kbyte instruction cache and the 16-Kbyte data cache, which cover a combined area of around 15 percent of the chip. We could have dedicated an area two- or three-times larger than that to caches; 100 Kbytes of on-chip cache are possible using 0.5-micron technology. This, however, is still insufficient for large applications. Instead, we elected to dedicate transistors to a decoupled memory subsystem, to registers and control structures that reduce operating system overhead, and to the graphics unit that accelerates new-media functions. (New media includes networking functions such as data copying, encryption and check summing, as well as multimedia.) We describe the various blocks forming the microarchitecture of UltraSparc, introducing them in the same order in which instructions typically progress through the pipeline.

Front end. The first block handling instructions is the prefetch and dispatch unit (the PDU, which is the shaded part of Figure 2). To keep the execution units busy, the PDU fetches instructions before the execution unit actually needs them, or even before it is known that they will be needed. This unit can prefetch instructions from all

0272-1732/96/$5.00 © 1996 IEEE

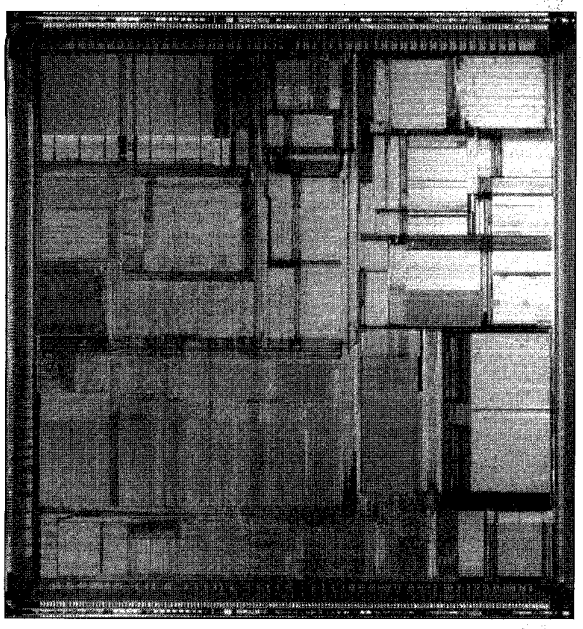


Figure 1. UltraSparc I die photo.

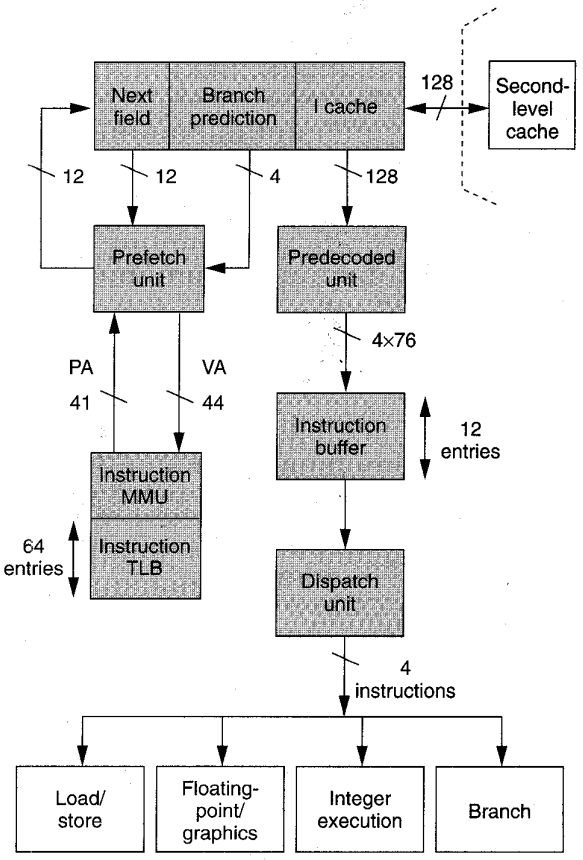


Figure 2. UltraSparc front end (shaded blocks); other blocks included to show front end's connection to the rest of the processor. The dotted line indicates that the second-level cache is off chip.

levels of the memory hierarchy: the instruction cache, external cache, and/or main memory. A dynamic branch prediction scheme implemented in hardware[2] prefetches across conditional branches. The prediction mechanism bases a branch outcome on a 2-bit history of the branch. A Next field[3] associated with every four instructions in the instruction cache points to the next instruction cache line for the PDU to fetch. Using the Next field makes it possible to follow taken branches and provide nearly the same instruction bandwidth achieved running sequential code. The instruction buffer stores up to 12 prefetched instructions until they proceed to the rest of the pipeline.

The 16-Kbyte instruction cache consists of two sets of 256 lines; each line contains eight 32-byte instructions. The 14 bits required to access any location in the instruction cache are the 13 least significant bits of the address and 1 bit that predicts the set in which instructions reside. (Since the minimum page size is 8 Kbytes, the 13 bits are always part of the page offset and do not need translation.) Out of a line of eight instructions, the cache sends up to four instructions to the instruction buffer, depending on the address. If the address points to one of the last three instructions in the line, the cache sends only that instruction and the ones between it and the end of the line. (We rejected hardware support for taking instructions from two adjacent lines for the sake of simplicity and timing considerations.) Consequently, for random accesses, the PDU fetches 3.25 instructions on the average from the instruction cache. For sequential accesses, the fetch rate may be greater than four instructions per cycle, equaling or surpassing the consumption rate of the pipeline (up to four instructions per cycle).

Due to the decoupled nature of the instruction buffer, an instruction cache miss does not necessarily result in bubbles being inserted into the pipeline. Part of the instruction cache miss processing, or even all of it, can overlap the execution of instructions already in the instruction buffer that await grouping and execution. The decoupled behavior of the PDU also helps when an instruction cache miss occurs during a pipeline stall (for example, due to a multicycle integer divide, floating-point divide dependency, dependency on load data that missed in the data cache, and so on). The miss (or part of it) may be transparent to the pipeline.

UltraSparc predicts the outcome of branches and based on those results, fetches the instructions that are likely to execute next. Although we accomplish this dynamically in hardware, the compiler has an impact on the initialization of the state machine. UltraSparc uses the static bit provided by BPcc and FBPfcc instructions to set the state machine in either the likely taken or the likely not-taken state (Figure 3, next page).

For branches without prediction (Bicc or FBfcc), UltraSparc initializes the state machine to likely not taken. A branch initialized to likely taken does not produce a correct next field for the very next instruction cache fetch, since it takes one

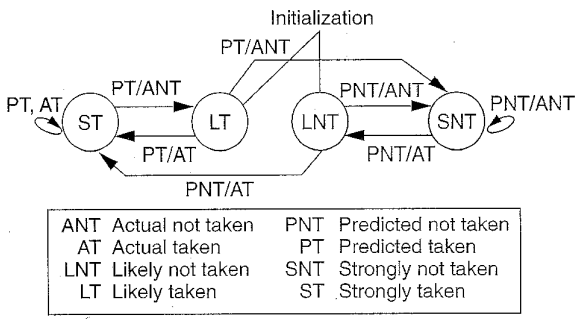


Figure 3. Dynamic branch prediction transition diagram.

extra cycle to generate the correct address (the branch offset added to the program counter). This results in the loss of two cycles for fetching instructions, yet does not necessarily lead to a pipeline stall. Such a penalty is much less than the mispredicted-branch penalty (four cycles) that would occur if the processor always ignored the branch prediction bit and used a static prediction (that is, always taken).

Figure 3 shows the state machine representing the branch prediction algorithm. Several slightly different algorithms based on this 2-bit counter mechanism can be implemented. Simulations on a large set of benchmarks showed that this one provides the smallest misprediction rate.

For loops in steady state, we designed the algorithm so that it requires two mispredictions for the prediction to change from taken to not taken. Each loop exit will thus cause a single misprediction (versus two for a 1-bit dynamic scheme).

Integer execution unit. The IEU (Figure 4) typically handles 60 percent of the dynamic instruction mix for applications such as transaction processing, compilers, text processing, and so on. All integer computations as well as branches requiring the contents of a register go through this unit. Two ALUs perform all common arithmetic and logical functions; each is 64 bits wide. The multiplier processes 2 bits of the multiplicand per cycle and has an early-out feature. With this mechanism, the result is ready as soon as the hardware detects that the most significant bits of the multiplicand are all zeroes (or all ones for a negative number). SPECint92 simulations showed that the multiplier's average latency is around five cycles.

The divider computes one bit per cycle. We dedicate a separate adder to computing the virtual address (VA) of loads and stores. In this way UltraSparc can perform a memory access in parallel with two other instructions requiring a 64-bit ALU, to allow more parallelism than other four-way, superscalar processors may offer.[4]

We implement the eight register windows as a three-dimensional register file.[5] A novel implementation groups corresponding bits from all windows together logically (that is, bit *i* for register r for all windows {0, ... 7}) and in the layout, hiding them underneath the multiple levels of metal. This implementation reduces the area of the register file significantly and speeds access time by dedicating a larger buffer to mutually exclusive bits. UltraSparc's design implements multilevel trap registers as specified by the 64-bit Sparc Version 9 instruction

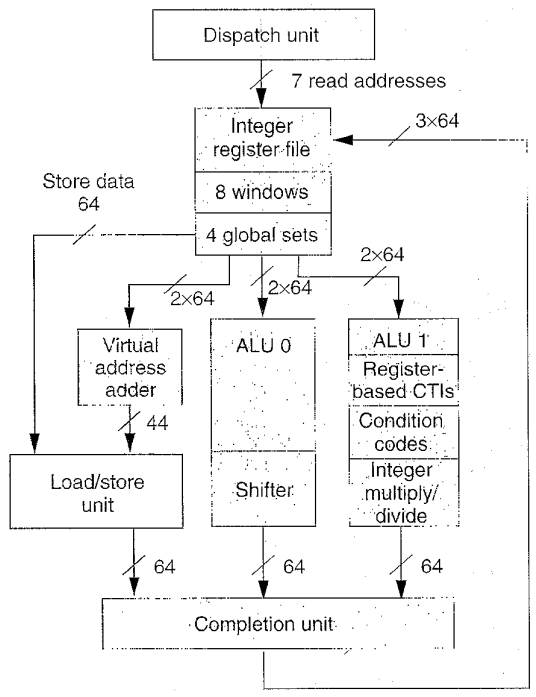


Figure 4. Integer execution unit (shaded blocks).

set architecture. The five trap levels that the architecture supports reduce system overhead significantly. In some instances—register window overflow, for example—these registers reduce the overhead by an order of magnitude.

Floating-point and graphics unit. We partitioned the FGU into five separate execution units, allowing the UltraSparc processor to issue and execute two floating-point instructions per cycle. As Figure 5 shows, UltraSparc includes a floating-point adder, multiplier,[6] and divide/square-root unit,[7] and a graphics adder and multiplier. Source and result data are stored in the 32-entry register file, in which each entry can contain a 32- or 64-bit value. Most instructions are fully pipelined (with a throughput of one per cycle), have a latency of 3 cycles, and remain unaffected by operand precision. Instructions have the same latency for single or double precision.

Our design does not pipeline the divide and square-root instructions, which execute in 12 cycles for single precision (22 for double precision). However, such instructions do not stall the processor. The processor may issue and execute other instructions that follow the divide or square-root instruction, and retire their results to the register file before the divide or square-root instruction finishes. Synchronizing the floating-point pipeline with the integer pipeline and predicting traps for long-latency operations maintains a precise-exception model.

UltraSparc introduces a comprehensive set of graphics instructions that provide fast hardware support for 2D and 3D image and video processing, image compression, audio

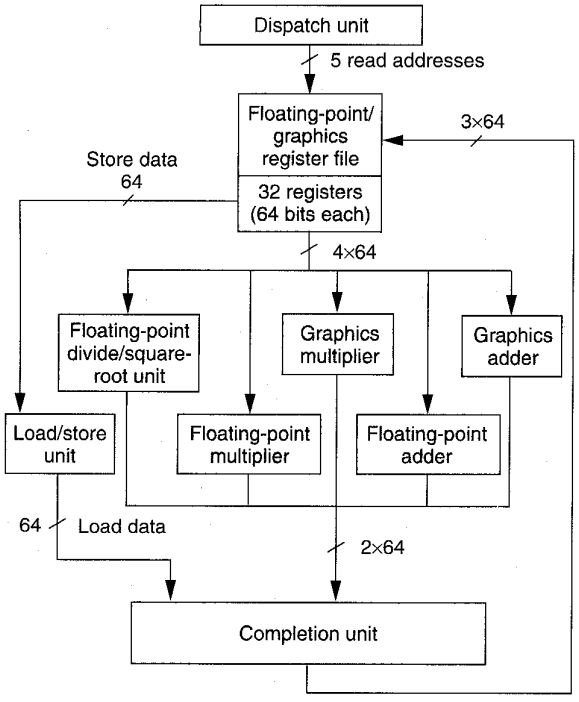


Figure 5. Floating-point and graphics unit (shaded blocks).

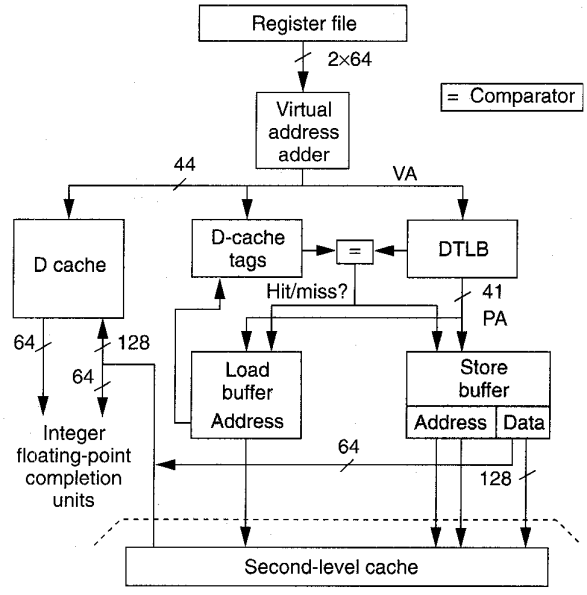


Figure 6. Load/store unit (shaded area).

processing, and so on. The processor provides 16- and 32-bit-partitioned add, Boolean, and compare operands, and also supports 8- and 16-bit-partitioned multiplies. Operations supported by the FGU include single-cycle pixel distance, data alignment, packing, and merge. Two dedicated functional units, the graphics multiplier and graphics adder, handle most of the multimedia instructions supported by UltraSparc.

Memory management and load/store units. The MMU provides mapping between a 44-bit virtual address and 41-bit physical address (PA). We accomplish this through a 64-entry instruction TLB and a 64-entry data TLB; both are fully associative. UltraSparc provides hardware support for a software-based TLB miss strategy as well as a separate set of global registers to process MMU traps. It also supports page sizes of 8 Kbytes (13-bit offset), 64 Kbytes (16-bit offset), 512 Kbytes (19-bit offset), and 4 Mbytes (22-bit offset).

The load/store unit (Figure 6) generates the virtual address of all loads and stores (including atomics and address space identifier loads). In addition, this unit handles data cache accesses and decouples load misses and stores from the pipeline through the load buffer and the store buffer. One load or one store can issue per cycle.

The data cache is a write-through, non-write-allocating, 16-Kbyte, direct-mapped cache with two 16-byte subblocks per line. We organized it as 512 lines with 32 bytes per line that are virtually indexed and physically tagged. The tag array is dual ported, so tag updates due to line fills do not collide with tag reads for incoming loads. Snoops to the data cache use the second tag port, so they do not delay incoming loads.

External cache and memory interface units. The main role of the ECU is to efficiently handle instruction and data cache misses. It also handles one access per cycle to the external cache. Accesses to the external cache are pipelined, take three cycles (pin to pin), and return 16 bytes of instructions or data per cycle. This effectively makes the external cache a part of the main processing pipeline. For programs with large data sets, we can maintain data in the external cache and schedule instructions with load latencies based on external-cache latency. Floating-point applications use this feature to effectively hide data cache misses. The external cache can contain 512 Kbytes, or 1, 2, or 4 Mbytes, but the line size is always 64 bytes. A MOESI (modified, own, exclusive, shared, invalid) protocol maintains coherency across the system.

The ECU provides overlap processing during load and store misses. For instance, stores that hit the external cache can proceed during load-miss processing. It also indiscriminately processes reads and writes without a costly turnaround penalty (only 2 cycles) and handles snoops. To provide high transfer bandwidth without polluting the external cache, the unit also efficiently processes block loads and block stores, which load or store a 64-byte line of data from memory to the floating-point register file.

The memory interface unit handles all transactions with the system controller such as external-cache misses, interrupts, snoops, write backs, and so on. It communicates with the system at either one-half or one-third of the processor frequency. A complete UltraSparc subsystem (Figure 7, next page) consists of the processor, synchronous SRAM components for the external-cache tags and data, and two data buffer chips. These chips isolate the external cache from the

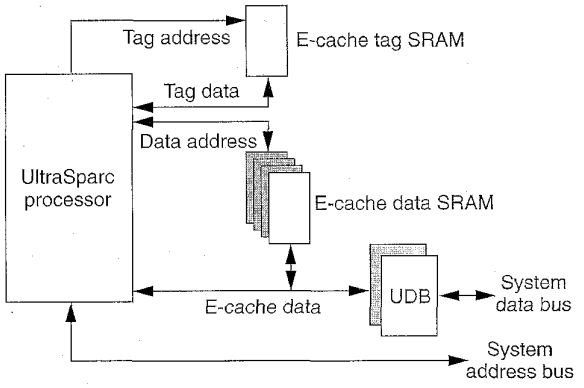


Figure 7. UltraSparc subsystem. (UDB is the UltraSparc data buffer.)

system, provide data buffers for incoming and outgoing system transactions, and error correction code generation and checking.

Processor pipeline

UltraSparc contains a nine-stage pipeline, and most instructions go through the pipeline in exactly nine stages. We consider instructions terminated after they go through the last (Write) stage; after that, changes to the processor state are irreversible. Figure 8 shows a diagram of the integer and floating-point pipeline stages.

To simplify pipeline synchronization and exception handling, we add three stages to the integer pipeline to make it symmetrical with the floating-point pipeline. This also eliminates the need for a floating-point queue. The design uses special logic to prevent these additional stages from creating new critical paths on the machine.

Floating-point instructions with a latency greater than 3 cycles (divide and square-root instructions) behave differently than others; the pipeline "extends" when the instruction reaches stage N1. Memory operations proceed asynchronously with the pipeline to support latencies longer than the latency of the on-chip data cache.

Hardware-software interaction

We designed UltraSparc to efficiently execute existing Sparc application programs in binary code and provide a performance improvement factor of about three over the previous generation of machines running the same code. Recompiling code to take advantage of several UltraSparc features offers a significantly larger performance gain.

Multiple-instruction issue. One of the most important contributions to improved application performance is UltraSparc I's ability to dispatch up to four instructions every cycle. The logic in the pipeline's grouping stage (Figure 8) enforces restrictions on precisely which instructions the processor can dispatch under which circumstances. During each cycle, the processor generally dispatches a group of four instructions that includes up to two integer, floating-point, or graphics instructions; one load or store; and one branch instruction. Because UltraSparc I only issues instructions in strict program order, the order of instructions in the code can be important. To maintain the maximum possible issue rate, the compiler must consider issue restrictions and arrange the code to initiate as many instructions as possible in each cycle.

Typically, the grouping stage does not allow a data- or control-dependent instruction to dispatch in the same cycle with the instruction that it depends on (Figure 9a). To improve performance, the grouping stage relaxes this requirement in

Integer pipeline

F Fetch	D Decode	G Group	E Execute	C Cache access	N_1	N_2	N_3	W Write
Fetches instructions from Instruction-cache	Decodes and sends instructions to instruction buffer	Groups and dispatches up to 4 instructions Accesses register file	Executes integer instructions Calculates virtual addresses	Accesses data-cache and TLB Resolves branches	Determines data-cache hit or miss Deferred load enters load buffer	Integer waits for floating-point/ graphics pipeline	Resolves traps	Writes all results to register files

R Register	X1	X2	X3
Further decodes floating-point/ graphics instructions Accesses register files	Execution starts	Execution continues	Execution finishes

Floating-point/graphics pipeline

Figure 8. UltraSparc pipeline stages (simplified).

(a)	add	G	E	C	N_1	N_2	N_3	W
	sub		G	E	C	N_1	N_2	N_3	W
(b)	setcc	G	E	C	N_1	N_2	N_3	W
	bicc	G	E	C	N_1	N_2	N_3	W
(c)	add	G	E	C	N_1	N_2	N_3	W
	st	G	E	C	N_1	N_2	N_3	W

Figure 9. Grouping of an add and its dependent subtract instruction (a); an instruction setting a condition code and its dependent conditional branch (b); and an add and its dependent store instruction (c).

some instances. Because of its organization, the pipeline requires neither the condition that a branch needs to test nor the data a store instruction will write until after the execution stage. This allows a branch to issue in the same group as the instruction that sets its condition codes (Figure 9b). Similarly, an integer instruction that calculates a result and the store of that result can issue in the same cycle (Figure 9c).

The ability to issue up to four instructions each cycle can yield a substantial performance gain over earlier processors. Proper scheduling of the code by the compiler (or assembly language programmer) is key to unlocking this performance improvement.

Data stream issues. Another area in which the hardware and software closely interact is the data memory subsystem. With a processor capable of executing four instructions per cycle, the rate of data accesses can increase substantially over that of a processor with a lower issue bandwidth. As a result, the data memory hierarchy must provide data to the execution pipelines at a sustained rate that keeps pace with the execution rate. Since code often consists of approximately 25 percent load and store instructions, UltraSparc I can expect a load or store every cycle. Therefore, the memory subsystem design supports very close to an average of one load or store per cycle and maintains this support even in the event of cache misses in the on-chip data cache. Compilers must schedule code appropriately to take advantage of this feature.

The latency of a load that hits in the data cache depends on the opcode. For unsigned loads, the processor can use data two cycles after the load. For instance, if the first two instructions in the instruction buffer are a load and an instruction dependent on that load, the grouping logic breaks the group after the load and inserts a bubble in the pipeline during the following cycle. Code compiled for an earlier Sparc processor with a load-use penalty of one cycle shows a penalty of about 0.1 cycle per instruction just for this rule. Thus, it is very important to separate loads from instructions that depend on them.

Under normal circumstances (that is, no snoops, arbitration conflict for the external-cache bus, and so on), the external cache returns a load that misses in the data cache but hits in the external cache six cycles later than if it hits in the

load r_1	F	D	G	E	C	N_1	Q	Q	Q	Q	Q						
use r_1	F	D	G	G	G	E	E	E	E	E	E	E	N_1	N_2	N_3	W	
					Group break							Execution resumes					

Figure 10. Data cache miss, external-cache hit. (Shading indicates seven-cycle stall).

load r_1	G	E	C	N_1	Q	Q	Q	Q	Q									
load r_2		G	E	C	N_1	Q	Q	Q	Q	Q								
load r_3			G	E	C	N_1	Q	Q	Q	Q	Q							
load r_4				G	E	C	N_1	Q	Q	Q	Q	Q						
load r_5					G	E	C	N_1	Q	Q	Q	Q	Q					
load r_6						G	E	C	N_1	Q	Q	Q	Q	Q				
load r_7							G	E	C	N_1	Q	Q	Q	Q	Q			
load r_8								G	E	C	N_1	Q	Q	Q	Q	Q		
use r_1									G	E	C	N_1	N_2	N_3	W			

Figure 11. Pipelined loads to the external cache. Shading shows six cycles separating a load from the instruction using its value.

data cache. Thus, if a use immediately follows a load, the grouping stage breaks the group, and a seven-cycle stall occurs (Figure 10).

Because of the high penalty associated with a load miss for code scheduled based on loads hitting in the data cache, UltraSparc provides hardware support for nonblocking loads. It does this through a load buffer that allows code scheduling based on external-cache hits. For applications with a working set too large to fit in the data cache (capacity misses) or applications with data patterns generating many conflict misses, a compiler may schedule the code so that data accesses effectively bypass the data cache. The application then relies on all data to come from the external cache.

A load that misses the data cache does not necessarily stall the pipeline (nonblocking loads): It merely goes to the load buffer where it waits until the data requested from the external cache returns. The pipeline stalls only when an instruction dependent on the load enters the pipeline before the load data arrives.

A load that misses in the data cache goes into the load buffer. The load buffer depth and the interaction of the load buffer with the rest of the pipeline support full throughput (one load per cycle) for an external cache with a three-cycle latency (pin to pin) and one-cycle throughput. As shown in Figure 11, if eight cycles separate the use from the load, no stall occurs, and the program execution achieves full throughput. This scheduling requires six more cycles between the load and the instruction using its value than data cache scheduling.

As Figure 11 shows, the load buffer must be at least seven entries deep to accommodate all pipelined loads in the steady state. The buffer requires two additional entries to hold seven loads and allow two more to issue (without having to stall them). One extra entry is in stage E, the other, in C (loads enter the load buffer in N_1), making the load buffer nine entries deep.

```
.align start 16 bytes
ld          [start],%f0          (D-Cache miss)
ld          [start + 8],%f2      (D-Cache hit)
ld          [start + 16],%f4     (D-Cache miss)
ld          [start + 24],%f6     (D-Cache hit)
```

Figure 12. Interleaved data cache hits and misses to the same subblock.

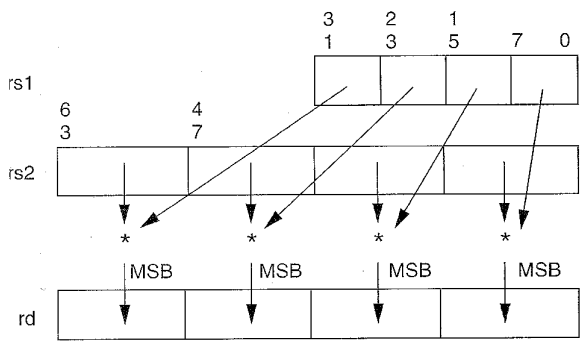


Figure 13. MUL8×16 instruction.

When a load enters the load buffer, the processor compares the memory location the load will access to that of all other (older) loads in the buffer. If other loads are to the same 16-byte subblock, the processor marks the entering load as a hit. This is because by the time it accesses the data cache array, the subblock will be present (Figure 12). The detection of a hit eliminates a transaction in the external cache and makes more slots available for other external-cache bus clients (such as the instruction cache, store buffer, and snoops). It is thus desirable to organize the code to access data sequentially. This may involve interchanging loops so that array subscripts increment by one between each load access.

One of the primary techniques in scheduling loads for the external cache is scheduling the load as early as possible in the instruction stream. Moving instructions to a position in the instruction stream before conditional branches can effectively hide the latencies of long operations; it also increases the number of candidate instructions that the compiler can schedule without conflicts. Sparc Version 9 provides nonfaulting loads (equivalent to silent loads used for Multiflow's Trace and Cydrome's Cydra-5 computers), allowing loads to move ahead of conditional control structures that regulate their use.

Nonfaulting loads execute as any other loads except that catastrophic errors, such as segmentation fault conditions, do not cause program termination. The hardware and software (via a trap handler) cooperate so that the load appears to complete normally with a zero result. To minimize page faults when a speculative load references a Null pointer (address zero), it is desirable to map low addresses (especially address zero) to a page of all zeros and use the nonfaulting-only page attribute bit.

Simulations of commonly used codes on UltraSparc have shown that programs have much to gain by using nonfaulting loads. For integer programs, the average group size sent through the pipeline is 33 percent larger with code motion allowed across one branch (using speculative loads). When we move instructions ahead two branches, the groups become 50 percent larger.

Multimedia support. By far, the most substantial opportunity on UltraSparc I for software to enhance performance is by taking advantage of our new multimedia instruction set, VIS. Graphics speed has a big effect on a workstation user's perception of performance. Graphics functionality is increasingly sophisticated and includes desktop video for teleconferencing and broadcast-quality viewing, 3D visualization and animation, image manipulation for desktop publishing, and so on.

Until now, these applications often required specialized graphics hardware. Typically, one or more graphics cards added functionality to the base machine. For example, one card would add MPEG-1 decompression capabilities, and a system would require a separate card to support 3D visualization. Implementing support for these applications directly on the processor may eliminate the need for additional graphics cards and lead to better overall system cost as well as free valuable I/O slots.

The lack of a standard platform supporting these features has hindered the development of multimedia application software. With UltraSparc, we saw an opportunity to provide a standard multimedia capability for future Sparc systems with only a 3 percent increase in the die area. Programmers can use the 30 new VIS instructions as they do other RISC instructions on UltraSparc. These instructions neither use memory-mapped I/O nor access special I/O devices.

The heart of VIS is a set of instructions optimized for the data types typically used in multimedia algorithms. These data types are 8-, 16-, and 32-bit integer or fixed-point values. Since UltraSparc I already had 64-bit data paths and registers for its execution units, the new instructions often operate on two 32-bit values, four 16-bit, or eight 8-bit values at once. Thus VIS makes full use of resources that would have been wasted by an instruction set with operations not optimized for multimedia data types.

We defined the instructions themselves by examining a variety of graphics and multimedia algorithms. Any potential instruction had to meet three requirements; it must

- execute in a single cycle or be easily pipelined,
- be applicable to several algorithms, and
- not affect the cycle time.

The result was RISC principles applied to multimedia. In other words, RISC-based VIS incorporates the fundamental operations present in most graphics and multimedia algorithms. Microprocessors may implement these instructions with relative ease and in a high-performance, fully pipelined manner.

VIS instructions fall into a few basic categories. First, at its core are instructions that perform various operations on the new data types; for example, the MUL8×16 instruction (Figure 13). This instruction performs pairwise multiplication of four 8-bit values with four 16-bit fixed-point values.

The second class of VIS instructions includes conversion instructions between various data types. The FEXPAND instruction, for instance, takes four 8-bit values and converts them into four 16-bit fixed-point values. Finally, we added instructions that accelerate memory access to meet the demanding requirements of most multimedia and graphics applications. Programmers may use the block load and store instructions that move 64-byte blocks of data into and out of registers to implement a very fast block-copy routine. Kohn et al.[8] give a complete description of VIS.

The various VIS instructions often require several traditional integer RISC instructions to perform the same function. Such RISC instructions are typically integer ALU operations, but UltraSparc I achieves benefits by implementing the VIS instructions on the floating-point side.

We added the VIS execution units to the floating-point unit mainly for four reasons. First, more registers are available because programs can store graphics data in all 32 floating-point registers while storing addresses and loop indices in integer registers. Second, programs do not typically use floating-point units concurrently with VIS instructions, which means that we can devote the issue slots normally used for floating-point instructions to VIS instructions. This achieves the maximum parallelism. Third, some instructions have a three-cycle latency that fits naturally into the floating-point pipeline design.

Fourth, UltraSparc's design bases the basic cycle time of the machine around key data path components dictated by the integer side of the processor (that is, ALUs, data cache access, and so on). Implementing VIS instructions on the integer side would have introduced extra gate levels in the adder (to allow intermediate carries to propagate for normal adds), added new functional units (four signed multipliers), and required more bypasses into critical mutiplexers.

In addition to the benefits discussed earlier, implementing VIS on the processor means that performance scales with frequency upgrades. Typically, processor frequency follows an aggressive curve due to gate shrinkage and/or process shrinkage (that is, moving from 0.5-micron CMOS to 0.35 micron). Such upgrade opportunities are typically not available or do not improve as rapidly on the ASICs common in graphics or multimedia acceleration boards.

Scaling also occurs with multiprocessor systems. Many multimedia applications lend themselves well to multithreading, which often attains speedup that is linear with an increasing number of processors.

Existing software libraries implement commonly used algorithms, such as MPEG-2, using VIS instructions. Thus, for many functions, the effort required to attain performance enhancements is relatively small, because library routines are already available.

One of VIS's primary benefits is its ability to implement a variety of algorithms, and limiting users to the prepared software libraries would inhibit this flexibility. Unfortunately, compiler technology has not advanced to the point where it can automatically detect situations in which VIS instructions might be appropriate. Thus, software developers must spend a bit more effort to take advantage of VIS. A C program can call a set of macros that generate each of the VIS instructions. Compilers can perform register allocation and scheduling as for any other C call, so programmers need not develop a detailed knowledge of UltraSparc's microarchitecture.

A quantitative evaluation of VIS applied to a class of engineering algorithms has demonstrated speedups of 2.5 to 7 times[9] over such algorithms' non-VIS implementations. Zhou et al. also describe the use of VIS in a broadcast-quality MPEG player.[1]

WITH INCREASINGLY POWERFUL optimizing compilers, the interaction between hardware and software becomes more important, and designers must give that interaction a high priority when designing a high-end microprocessor. It was our goal to describe part of this boundary here.

We have applied the concepts presented here to the UltraSparc II microarchitecture to further enhance its performance. An added prefetch instruction allows the compiler to better control when data enters the cache. By scheduling prefetch instructions appropriately, the compiler can eliminate stalls due to the processor waiting for main memory. Data is simply preloaded in a level of the memory hierarchy closer to the processor pipeline. Other extensions contribute to improving UltraSparc II's performance to an estimated 465 SPECint92 and 660 SPECfp92 for a 300-MHz part.

Acknowledgments

We acknowledge the work of other members of the architecture team, specifically L. Kohn and G. Maturana. Key members of the logic design team at Sun are J. Bauman, R. Eltejaein, P. Ferolito, P. Fu, D. Greenhill, D. Greenley, G.P. Grewal, K. Holdbrook, B. Kim, H. Kwan, M. Levitt, C. Narasimhaiah, K. Normoyle, N. Parveen, M. Wong, and R. Yu.

References

1. D.L. Weaver and T. Germond, *The Sparc Architecture Manual, Version 9,* Prentice Hall, Englewood Cliffs, N.J., 1994.
2. J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Ann. Int'l Symp. Computer Architecture,* IEEE Computer Society Press, Los Alamitos, Calif., 1981, pp. 135-148.
3. B. Calder and D. Grunwald, "Next Cache Line and Set Prediction," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1995, pp. 287-297.
4. J.H. Edmondson, P. Rubenfeld, and R. Preston, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro,* Vol. 15, No. 2, Apr. 1995, pp. 33-43.
5. M. Tremblay, B. Joy, and K. Shin, "A Three Dimensional Register File for Superscalar Processors," *Proc. 28th Ann. Hawaii Int'l Conf. Systems Sciences,* IEEE CS Press, 1995, pp. 191-201.
6. R.K. Yu and G.B. Zyner, "167 MHz Radix-4 Floating Point Multiplier," *Proc. 12th Symp. Computer Arithmetic,* IEEE CS Press, 1995, pp. 149-154.
7. J.A. Prabhu and G.B. Zyner, "167 MHz Radix-8 Divide and Squareroot Using Overlapped Radix-2 Stages," *Proc. 12th Symp. Computer Arithmetic,* IEEE CS Press, 1995, pp. 155-162.

8. L. Kohn et al., "The Visual Instruction Set VIS in UltraSPARC," *Proc. Compcon,* IEEE CS Press, 1995, pp. 462-469.
9. M. Tremblay et al., "A Visual Instruction Set VIS for New Media Processors," submitted to *IEEE Micro,* Aug. 1996.
10. C. Zhou et al., "MPEG Video Decoding with the UltraSparc Visual Instruction Set," *Proc. Compcon,* IEEE CS Press, 1995, pp. 470-475.

Marc Tremblay is a computer architect involved in the research and development of high-performance processors at Sun Microsystems. As an architect for UltraSparc I and II, his main contributions have focused on microarchitecture definition and processor performance evaluation. His current work relates to integrating extensive multimedia capabilities directly onto the processor and designing processors that more efficiently execute Java applications. He is also a member of Sun's Architecture Group, which the company chartered to investigate and propose novel processor architectures.

Tremblay holds MS and PhD degrees in computer science from the University of California, Los Angeles, and a BS in physics engineering from Laval University in Canada. He is a member of the IEEE Computer Society.

J. Michael O'Connor is a member of the Architecture Group at Sun Microsystems, where he is involved in the research and development of high-performance microprocessors. He participated in the simulation and performance analysis of the UltraSparc I processor. His interests include computer architecture, hardware-software codesign, and performance analysis.

O'Connor holds a BSEE from Rice University and an MSEE from the University of Texas at Austin. He is a member of the IEEE Computer Society.

Direct questions concerning this article to Marc Tremblay, Sun Microelectronics, Sun Microsystems Inc., USUN02-301, 2550 Garcia Ave., Mountain View, CA 94043; tremblay@eng.sun.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 156 Medium 157 High 158