ULTRASPARC-III:

Designing Third-Generation 64-Bit Performance

EVERY DECISION HAS AT LEAST ONE ASSOCIATED TRADE-OFF. SYSTEM ARCHITECTS ULTIMATELY ARRIVED AT THIS 64-BIT PROCESSOR DESIGN AFTER A CHALLENGING SERIES OF DECISIONS AND TRADE-OFFS.

Tim Horel and Gary Lauterbach
Sun Microsystems

The UltraSPARC-III is the third generation of Sun Microsystems' most powerful microprocessors, which are at the heart of Sun's computer systems. These systems, ranging from desktop workstations to large, mission-critical servers, require the highest performance that the UltraSPARC line has to offer. The newest design permits vendors the scalability to build systems consisting of 1,000+ UltraSPARC processors. Furthermore, the design ensures compatibility with all existing SPARC applications and the Solaris operating system.

The UltraSPARC-III design extends Sun's SPARC Version 9 architecture, a 64-bit extension to the original 32-bit SPARC architecture that traces its roots to the Berkeley RISC-I processor.[1] Table 1 (next page) lists salient microprocessor pipeline and physical attributes. The UltraSPARC-III design target is a 600-MHz, 70-watt, 19-mm die to be built in 0.25-micron CMOS with six metal layers for signals, clocks, and power.

Architecture design goals

In defining the newest microprocessor's architecture, we began with a set of four high-level goals for the systems that would use the UltraSPARC-III processor. These goals were shaped by team members from marketing, engineering, management, and operations in Sun's processor and system groups.

Compatibility

With more than 10,000 third-party applications available for SPARC processors, compatibility—with both application and operating system—is an essential goal and primary feature of any new SPARC processor. Previous SPARC generations required a corresponding, new operating system release to accommodate changes in the privileged interface registers, which are visible to the operating system. This in turn required all third-party applications to be qualified on the new operating system before they could run on the new processor. Maintaining the same privileged register interface in all generations eliminated the delay inherent in releasing a new operating system.

Part of our compatibility goal included increasing application program performance—without having to recompile the application—by more than 90%. Furthermore, this benefit had to apply to all applications, not just those that might be a good match for the new architecture. This goal demanded a sizable microarchitecture performance increase while maintaining the programmer-visible characteristics (such as number of functional units and latencies) from previous generations of pipelines.

Performance

To design high performance into the Ultra-

0272-1732/99/$10.00 © 1999 IEEE

Table 1. UltraSPARC-III pipeline and physical data.

Pipeline feature	Parameter
Instruction issue	4 integer
	2 floating-point
	2 graphics
Level-one (L1) caches	Data: 64-Kbyte,4-way
	Instruction: 32-Kbyte,4-way
	Prefetch: 2-Kbyte,4-way
	Write: 2-Kbyte,4-way
Level-two (L2) cache	Unified (data and instruction):
	4- and 8-Mbyte, 1-way
	On-chip tags
	Off-chip data
Physical feature	**Parameter**
Process	0.25-micron CMOS, 6 metal layers
Clock	600+ MHz
Die size	360 mm^2
Power	760 watts @1.8 volts
Transistor count	RAM: 12 million
	Logic: 4 million
Package	1,200-pin LGA

Table 2. Load latency increases as ILP decreases.

Instruction-level parallelism (instructions per cycle)	Load latency (cycles)
0.75	4
1.00	3
1.50	2
3.00	1

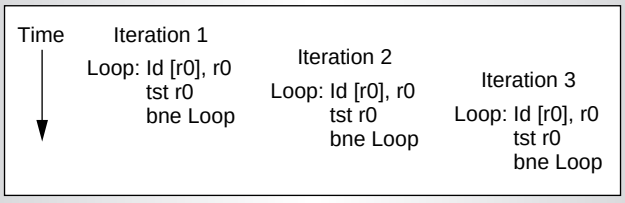

Figure 1. A serially data-dependent algorithm example, which is a simple search for the end of a linked-list data structure.

SPARC-III, we believed we needed—and have—a unique approach. Recent research shows that the trend for system architects is to design ways of extracting more instruction-level parallelism from programs. In considering many aggressive ILP extraction techniques for the UltraSPARC-III, we discovered that they share a common undesirable characteristic—the speedup varies greatly across a set of programs.

Relying on ILP techniques for most of the processor's performance increase would not deliver the desired performance boost. ILP techniques vary greatly from program to program because many programs or program sections use algorithms that are serially data dependent. Figure 1 shows an example of a serially data-dependent algorithm.

In a high-performance processor such as the UltraSPARC-III, several iterations of the loop can concurrently execute. Figure 1 shows three iterations overlapped in time. The time it takes these three iterations to execute depends on the latency of the load instruction. If the load executes with a single-cycle latency, then the maximum overlap occurs, and the processor can execute three instructions each cycle. As the load latency increases, the amount of ILP overlap decreases, as Table 2 shows.

Many ILP studies have assumed latencies of 1 for all instructions, which can cause misleading results. In a nonacademic machine, the load instruction latency is not a constant but depends on the memory system's cache hit rates, resulting in a fractional average latency. The connection between ILP (or achieved ILP, commonly referred to as instructions per cycle—IPC or 1/CPI) and operation latency makes these units cumbersome to analyze for determining processor performance.

One design consideration was an average latency measurement of a data dependency chain (ending at a branch instruction) for the SPEC95 integer suite. The measurement was revealing: The average dependent chain in SPEC95 consisted of a serial-data-dependent chain with one and a half arithmetic or logical operations (on average, half of a load instruction), ending with a branch instruction. A simplified view is that SPEC95 integer code is dominated by load-test-branch data dependency chains.

We realized that keeping the execution latency of these short dependency chains low would significantly affect the UltraSPARC-III's performance. Execution latency is another way to view the clock rate's profound influence on performance. As the clock rate scales, all the bandwidths (in operations per unit time) and latencies (in time per operation) of a processor scale, proportionately.

Bandwidth (ILP) alone cannot provide a speedup for all programs; it's only by scaling both bandwidth and latency that performance can be boosted for all programs.

Our focus thus became to scale up the bandwidths while simultaneously reducing latencies. This goal should not be viewed simply as raising the clock rate. It's possible to simply raise the clock rate by deeper pipelining the stages but at the expense of increased latencies. Each time we insert a pipeline stage, we incur an additional increment in the clocking overhead (flop delay, clock skew, clock jitter). This forces less actual work to be done per cycle, thus leading to increased latency (in absolute nanoseconds). Our goal was to push up the clock rate while at the same time scaling down the execution latencies (in absolute nanoseconds).

Scalability

The UltraSPARC-III is the newest generation of processors that will be based on the design we describe in this article. We designed this processor so that as process technology evolves, it can realize the full potential of future semiconductor processes. Scalability was therefore of major importance. As an example, research at Sun Labs indicates that propagation delay in wiring will pose increasing problems as process geometries decrease.[2] We thus focused on eliminating as many long wires as possible in the architecture. Any remaining long wires are on paths that allowed cycles to be added with minimum performance impact.

Scalability also required designing the on-chip memory system and the bus interface to handle multiprocessor systems to be built with from two to 1,000 UltraSPARC-III processors.

Reliability

A large number of UltraSPARC-III processors will be used in systems such as transaction servers, file servers, and compute servers. These mission-critical systems require a high level of reliability, availability, and serviceability (RAS) to maximize system uptime and minimize the time to repair when a failure does occur.

One of our goal requirements was to detect as many error conditions as possible. In addition, we added three more guidelines to improve system RAS:

- *Don't allow bad data to propagate silently.* For example, when the processor sourcing data on a copy-back operation detects an uncorrectable cache ECC error, it poisons the outgoing data with a unique, uncorrectable ECC syndrome. Any other processor in a multiprocessor system will thus get an error if it touches the data. The sourcing processor also takes a trap when the copy-back error is detected to fulfill the next guideline, below.
- *Identify the source of the error*. To minimize downtime of large multiprocessor systems, the failing replaceable unit must be correctly identified so that a field technician can quickly swap it out. This requires the error's source to be correctly identified.
- *Detect errors as soon as possible*. If errors are not quickly detected, identifying the error's true source can become difficult at best.

Major architectural units

The processor's microarchitecture design has six major functional units that perform relatively independently. The units communicate requests and results among themselves through well-defined interface protocols, as Figure 2 shows.

Instruction issue unit

This unit feeds the execution pipelines with instructions. It independently predicts the control flow through a program and fetches the predicted path from the memory system. Fetched instructions are staged in a queue before forwarding to the two execution units: integer and floating point. The IIU includes a 32-Kbyte, four-way associative instruction cache, the instruction address translation buffer, and a 16 K-entry branch predictor.

Integer execute unit

This unit executes all integer data type instructions: loads, stores, arithmetics, logicals, shifts, and branches. Four independent data paths enable up to four integer instructions to be executed per cycle. The allowable per-cycle integer instruction mix is as follows:

- 2 from (arithmetic, logical, shift), A0/A1 pipelines
- 1 from (load, store), MS pipeline
- 1 from (branch), BR pipeline

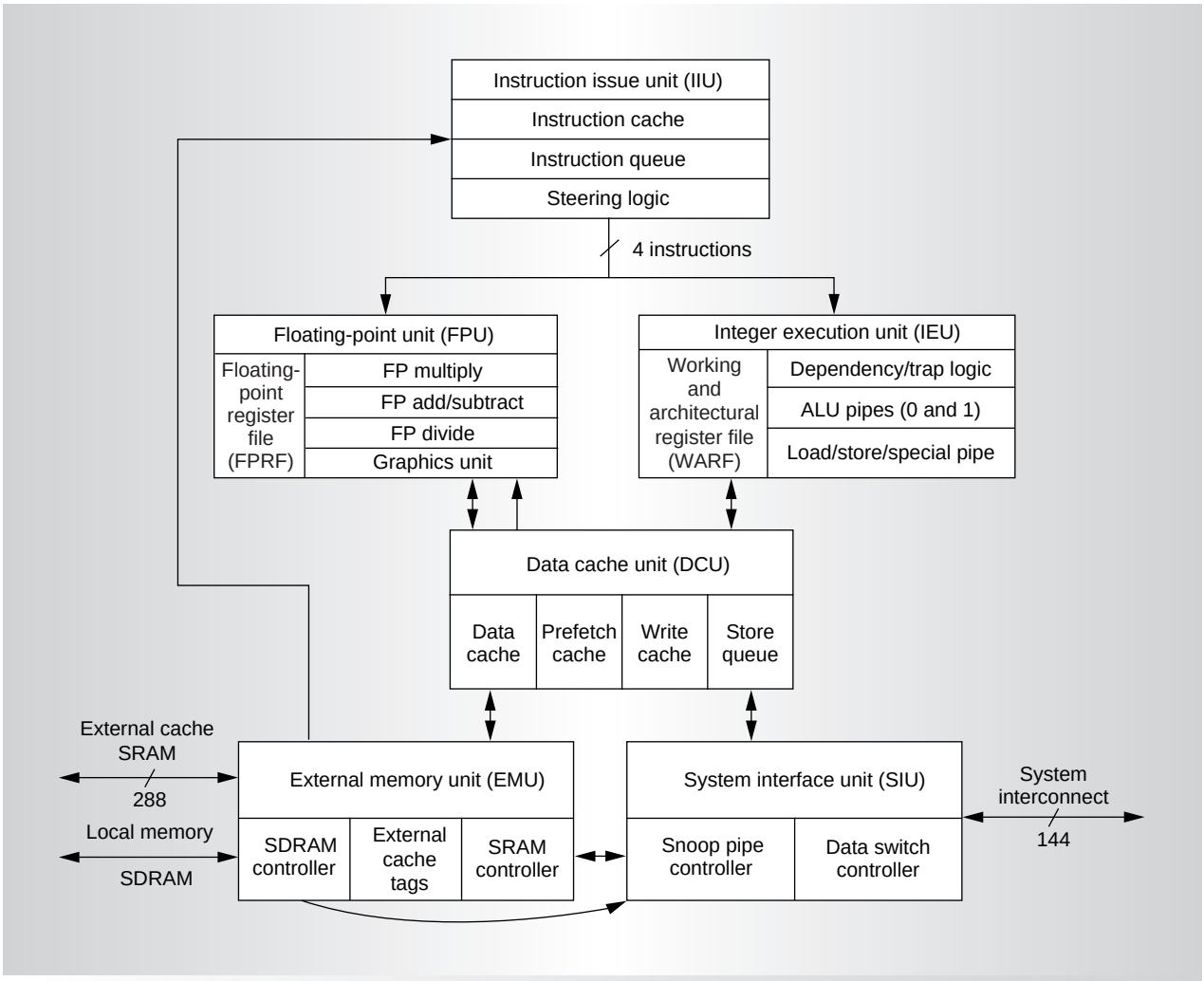

Figure 2. Communication paths between the UltraSPARC-III's six major functional units.

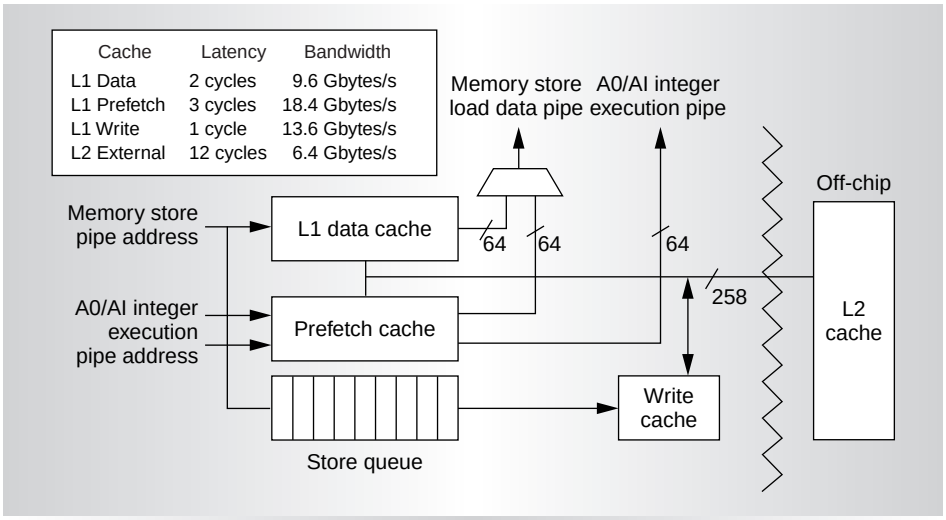

Figure 3. Data cache unit block diagram.

The load/store pipeline also executes floating-point data type memory instructions. A second floating-point data type load instruction can be issued to either of the A0 or A1 pipelines. We describe this instruction in more detail in the prefetch cache discussion, later as part of the on-chip memory section.

Data cache unit (on-chip memory system)

The data cache unit comprises the level-one (L1) on-chip cache memories and the data address translation

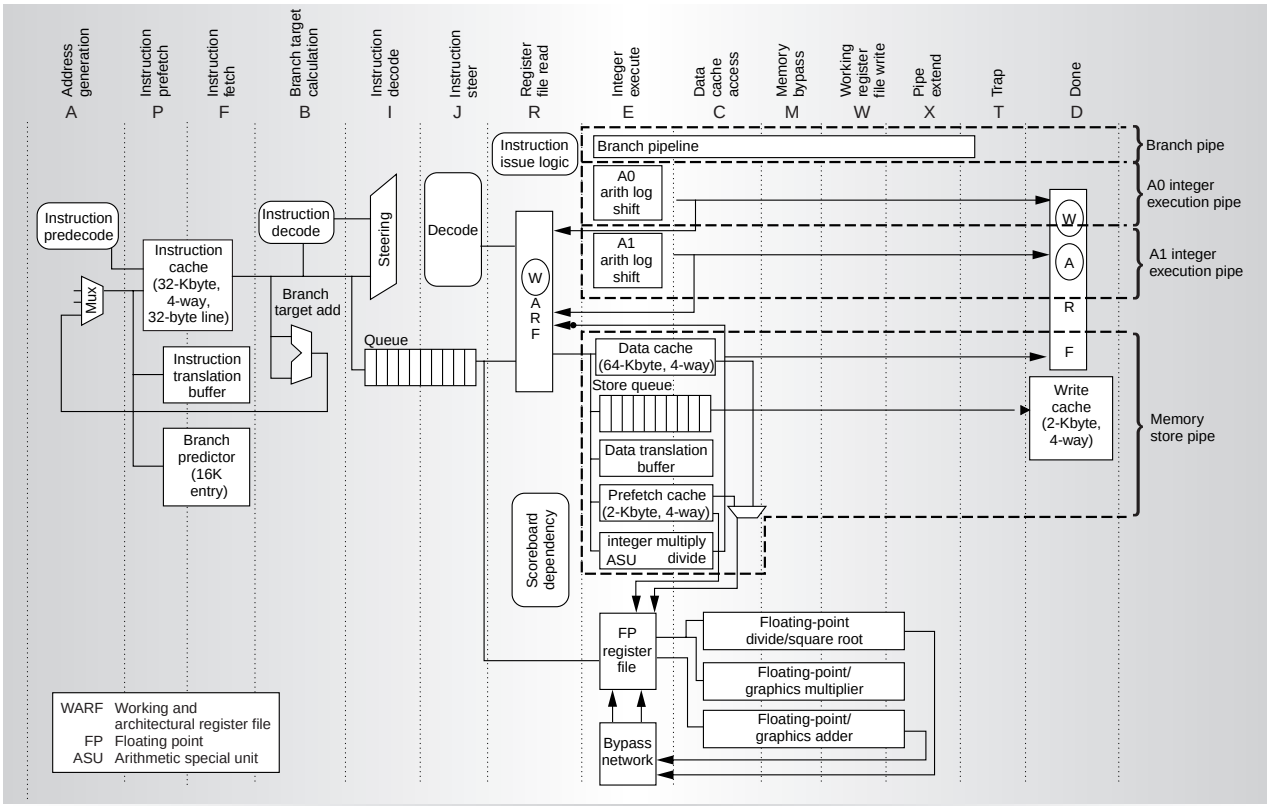

Figure 4. The UltraSPARC-III microprocessor instruction pipeline.

buffer, as Figure 3 shows. There are three first-level, on-chip data caches: data—64-Kbyte, four-way associative, 32-byte line; prefetch—2-Kbyte, four-way associative, 64-byte line; and write—2-Kbyte, four-way associative, 64-byte line.

Floating-point unit

This unit contains the data paths and control logic to execute all floating-point and partitioned fixed-point data type instructions. Three data paths concurrently execute floating-point or graphic (partitioned fixed-point) instructions, one each per cycle from the following classes:

- Divide/multiply (single or double precision or partitioned),
- Add/subtract/compare (single or double precision or partitioned), and
- An independent division data path, which lets a nonpipelined divide proceed concurrently with the fully pipelined multiply and add data paths.

External memory unit

This unit controls the two off-chip memory structures: the level-two (L2) data cache built with off-chip synchronous RAMs (SRAMs), and the main memory system built with off-chip synchronous DRAMs (SDRAMs).

The L2 cache controller includes a 90-Kbyte on-chip tag RAM to support L2 cache sizes up to 8 Mbytes. The main memory controller can support up to four banks of SDRAM memory totaling 4Gbytes of storage.

System interface unit

This unit handles external communication to other processors, memory systems, and I/O devices. The unit can handle up to 15 outstanding transactions to external devices, with support for full out-of-order data delivery on each transaction.

Instruction pipeline

To meet our clock rate and performance goals, we concluded that we needed a deep pipeline. The UltraSPARC-III 14-stage pipeline, as Figure 4 shows, has more stages

How pipeline stages work in the UltraSPARC-III

Stage	Function
A	Generate instruction fetch addresses, generate predecoded instruction bits on cache fill
P	Fetch first cycle of instructions from cache; access first cycle of branch prediction
F	Fetch second cycle of instructions from cache; access second cycle of branch prediction; translate virtual-to-physical address
B	Calculate branch target addresses; decode first cycle of instructions
I	Decode second cycle of instructions; enqueue instructions into the queue
J	Steer instructions to execution units
R	Read integer register file operands; check operand dependencies
E	Execute integers for arithmetic, logical, and shift instructions; read, and check dependency of, first cycle of data cache access floating-point register file
C	Access second cycle of data cache, and forward load data for word and double-word loads; execute first cycle of floating-point instructions
M	Load data alignment for half-word and byte loads; execute second cycle of floating-point instructions
W	Write speculative integer register file; execute third cycle of floating-point instructions
X	Extend integer pipeline for precise floating-point traps; execute fourth cycle of floating-point instructions
T	Report traps
D	Write architectural register file

than any previous UltraSPARC pipeline. The extra pipeline stages must be added in pipeline paths that are infrequently used—for example, trap generation.

Each pipeline stage performs part of the work necessary to execute an instruction, as the box, "How pipeline stages work in the UltraSPARC-III," explains. The instruction issue unit occupies the A through J stages of the pipeline, and the integer execution unit accounts for the R through D stages. The data cache unit occupies the E, C, M, and W stages of the pipe in parallel with integer execution unit stages. The floating-point unit is shown as a side pipeline that parallels the E through D stages of the integer pipeline. The other units of the machine (system interface unit and external memory unit) have internal pipelines but are not considered part of the core processor pipe.

We determined the processor's pipeline depth early in the design process by analyzing several basic paths. We selected the integer execution path to determine the machine cycle time so we would have minimum latency for the basic integer operation. Using an aggressive dynamic adder for this stage resulted in our setting the number of logic gate levels per stage to approximately eight—the exact number depends on circuit style.

Early analysis also showed that with eight gate delays (using a three-input NAND with a fan-out of three as a gate delay) per stage, the overhead due to synchronous clocking (from flip-flop delay, clock skew, jitter, and so on) would consume about 30% of the cycle. If we tried to pipeline the integer execution over two cycles (commonly called super-pipelining), the second 30% clocking overhead would significantly increase latency. As a result, performance would decline in some applications. The on-chip cache memories are pipelined across two stages, but they don't suffer the additional clock overhead because we used a wave-pipeline circuit design.

Another known critical path from previous SPARC designs is the global pipe stall signal. This signal freezes the flip-flops when an unexpected event, such as a data cache miss, occurs. This freeze signal is dominated by wire delay that we knew would have technology scaling problems, so we decided to completely eliminate it by using a nonstalling pipeline. Since the pipeline state couldn't be frozen, we had to use a repair mechanism that could restore the state when an unexpected event occurs. It's handled like a trap: The pipeline is allowed to drain, and its state is restored by refetching instructions that were in the pipeline, starting at the A stage.

One concern with a deep pipeline is the cost of branch misprediction. When a branch is mispredicted, instructions must be refetched starting at the A stage. This incurs a penalty of eight cycles (A through E stages). With recent improvements in branch prediction, a processor incurs this penalty much less frequently, allowing the pipeline to be longer with only a small performance cost. In addition, we designed a small amount of alternate path buffering in the I stage (the miss queue). If a predicted taken branch thus mispredicts (actually not taken), a few instructions are immediately available to start in the I stage. This effectively halves the branch misprediction penalty.

Pipeline stages after the M stage impact performance whenever the pipeline must be drained. We overlapped the new fetch (for a trap target or a refetch) with the back of the

pipeline draining. By doing so, we could add stages to the back of the pipe, as long as we could guarantee that the pipe results were drained before new instructions reached the R stage. To ease the implementation of precise exceptions, we added back-end pipe stages up to this limit.

We pushed back the floating-point execution pipeline by one cycle relative to the integer execution pipe. This allows extra time to the floating-point unit for wire delays. We had to keep the machine's latency-sensitive integer part physically small to minimize wire delays. Moving the floating-point unit away from the integer core was a major step toward achieving this goal.

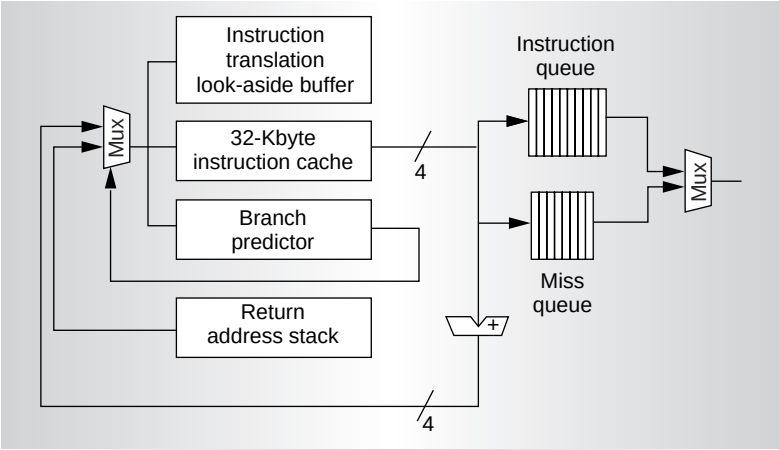

Figure 5. Instruction issue unit (IIU) block diagram.

Instruction issue unit

Experience with previous UltraSPARC pipelines showed our design teams that many critical-timing paths occurred in the instruction issue unit. Consequently, we knew we had to pay particular attention to designing this part of the processor. Our decision to keep UltraSPARC-III a static speculation machine compatible with the previous pipelines paid off in the IIU design. Dynamic speculation machines require very high fetch bandwidths to fill an instruction window and find instruction-level parallelism. In a static speculation machine, the compiler can make the speculated path sequential, resulting in fewer requirements on the instruction fetch unit. We used this static speculation advantage to simplify the fetch unit and minimize the number of critical timing paths. Figure 5 illustrates the IIU's different blocks.

The pipeline's A stage corresponds to the address lines entering the instruction cache. All fetch address generation and selection occurs in this pipe stage. Also at the A stage, a small, 32-byte buffer supports sequential prefetching into the instruction cache. When the instruction cache misses, the cache line requires 32 bytes. But instead of requesting only the 32 bytes needed for the cache, the processor issues a 64-byte request. The first 32 bytes fill the cache line; the second 32 bytes are stored in the buffer. The buffer can then be used to fill the cache if the next sequential cache line also misses.

We distributed the instruction cache access over two cycles (P and F pipeline stages) by using a wave-pipelined SRAM design. In this design, the cache is pipelined without the use of latches or flip-flops. Careful circuit design ensures that the data waves in each cycle do not overtake each other.[3] In parallel with the cache access, this design also allows branch predictor and instruction address translation buffer access. By the time the instructions are available from the cache in the B stage, we also have the physical address from the translator and a prediction for any branch that was fetched. The processor uses all this information in the B stage to determine whether to follow a sequential- or taken-branch path. The processor also determines whether the instruction cache access was a hit or a miss. If the processor predicts a taken branch in the B stage, the processor sends back the target address for that branch to the A stage to redirect the fetch stream.

Waiting until the B stage to redirect the fetch stream lets us use a large, accurate branch predictor. We minimized the wait penalty for branch targets through compiler static speculation and the instruction buffering queues.

The branch predictor uses a Gshare algorithm[4] with 16K 2-bit saturating up/down counters. Since the predictor is large, it needed to be pipelined across two stages. In the original Gshare scheme, this would require the predictor to be indexed with an old or inaccurate copy of the program counter (PC).

We modified the scheme by offsetting the history bits such that the three low-order index bits into the predictor use PC information only. Each time the predictor is accessed, eight counters are read out. Later, one of them is

selected (using the three low-order PC bits) in the pipeline's B stage after the exact position of the first branch in the fetch group is known.

Simulations showed that not XORing the global-history bits with the low-order PC bits does not affect the branch predictor performance. The three-cycle loop through the A, P, and F stages for taken branches lets us keep the global-history register at the predictor's input in an exact state. The register is exact because there can only be one taken branch every three cycles.

We designed two instruction buffering queues into the UltraSPARC-III: the instruction queue and the miss queue. The 20-entry instruction queue decouples the fetch unit from the execution units, allowing each to proceed at its own rate. The fetch unit is allowed to work ahead, predicting the execution path and stuffing instructions into the instruction queue until it's full. When the fetch unit encounters a taken branch delay, we lose two fetch cycles to fill the instruction queue. Our simulations show, however, that there are usually enough instructions already buffered in the instruction queue to occupy the execution units. This two-cycle delay also gives us the opportunity to buffer the sequential instructions that have already been accessed into the four-entry miss queue. If we then find that we mispredicted the taken branch, the instructions from the miss queue are immediately available to send to the execution units.

The last two stages of the instruction issue unit decode the instruction type and steer each instruction to the appropriate execution unit. These two functions must be done in separate pipeline stages to achieve the cycle time goal.

Integer execute unit

We guarantee the minimum logical latency for the most frequent instructions by setting the processor cycle time with the integer execute stage. However, the amount of work we try to fit in one execute stage cycle varies over a wide range. We used several techniques to minimize the cycle time of the E stage. We applied the most aggressive circuit techniques available to design the E stage—the entire integer data path uses dynamic precharge circuits. We had to carefully design the physical data path to minimize wiring lengths; wire delay causes more than 25% of the delay in this stage.

This level of design cannot be applied to the entire processor. It was vital that the microarchitecture clearly showed where this sort of design investment would pay off in performance.

We extended the future file method to help achieve a short cycle time.[5] The working and architectural register file (WARF) let us remove the result bypass buses from most of the integer execution pipeline stages. Without bypass buses, we could shorten the integer data path and narrow the bypass multiplexing. Both contribute to a short cycle time.

The WARF can be regarded as two separate register files. The processor accesses the working register file in the pipeline's R stage and supplies integer operands to the execution unit. The file is also written with integer results as soon as they become available from the execution pipeline. Most integer operations complete in one cycle, with results immediately written into the working register file in the pipeline's C stage. If an exceptional event occurs, the immediately written results must be undone. Undoing results is accomplished with a broadside copy of all integer registers from the architectural register file back into the working register file. By placing the architectural register file at the end of the pipe, we can ensure that we do not commit results into it until we have resolved all exceptional conditions. Copying the architectural register file back into the working register file gives us a simple, fast way to repair the pipeline state when exceptions do occur.

The state copying of the WARF also offers a simple mechanism to implement the SPARC architecture register windows. The architectural register file contains a full eight windows' worth of integer registers. A broadside copy into the working register file of the new window is made when a window must be changed.

We moved the data path for the least frequently executed integer instructions to a separate location to further unburden the core integer execution pipeline from extra wiring. Nonpipelined instructions such as integer divide are executed in this data path, which is called the arithmetic/special unit (ASU). The ASU was decoupled from impacting the machine cycle time by dedicating a full cycle each way to get operands to and from this unit.

On-chip memory system

The performance influence of the memory system becomes increasingly dominant as processor performance and clock rates increase. For this reason, the on-chip memory system was crucial to our delivering UltraSPARC-III's performance and scalability goals. In designing the on-chip memory system, we followed this principle: Achieve uniform performance scaling by scaling both bandwidth and latency. A popular architectural trend is to try to hide the memory latency scaling problem by using program ILP. Not surprisingly, the hiding is not free: Programs with low ILP suffer a performance hit, and ILP that could have been used to speed up the program execution was wasted on "hiding" the lagging memory system. Table 3 summarizes the UltraSPARC-III on-chip memory system.

Table 3. UltraSPARC-III's on-chip memory system parameters.

Cache	Size (Kbytes)	Associativity	Line length (bytes)	Protocol
Instruction	32	4-way, microtag pseudorandom	32	Store coherent
Data	64	4-way, microtag pseudorandom	32	Write-through
Write	2	4-way, LRU	64	Write-validate
Prefetch	2	4-way, LRU	64	Store coherent

The key to scaling memory latency in the UltraSPARC-III is the first-level, sum-addressed memory data cache.[6] Fusing the memory address adder with the word line decoder for the data cache largely eliminates the address adder's latency. This enabled us to increase the data cache size to completely occupy the time available in two processor cycles. The combination of an increased cache size with a scaled clock rate while maintaining a two-cycle access gives us a linear memory latency improvement. We can demonstrate this linear improvement with the following calculation of overall memory latency:

average latency = L1 hit time + L1 miss rate * L1 miss time + L2 miss rate * L2 miss time

Table 4 shows latency trade-offs, which we achieved with some representative values from simulations of the SPEC integer benchmarks to compare the UltraSPARC-II and UltraSPARC-III.

Comparing the 300-MHz UltraSPARC-II with the 600-MHz UltraSPARC-III shows that we were able to scale the average memory latency by more than the clock ratio. We achieved this result through the use of the sum-addressed memory (SAM) cache and improvements in the L2 cache and memory

Table 4. Memory latency trade-offs. US-II is the UltraSPARC-II; US-III is the UltraSPARC-III. SAM is sum-addressed memory. All L2 caches are 4-Mbyte, direct-mapped.

Clock	L1 data cache	L1 cache latency (cycles)	Load use penalty (cycles)	L1 miss rate (fraction per load instruction)	L1 miss rate cost (ns)	L-2 miss rate (fraction per load instruction)	L2 miss cost (ns)	Average memory latency (ns)
US-II								
300 MHz	16-Kbyte, 1-way	2	1	0.10	30	0.01	150	11.16
600 MHz	16-Kbyte, 1-way	2	1	0.10	20	0.01	100	6.33
600 MHz	64-Kbyte, 4-way	3	2	0.04	20	0.01	100	6.80
US-III								
600 MHz	64-Kbyte, 4-way, SAM	2	1	0.04	20	0.01	100 [estimated for design purposes]	5.13

latencies.[6] The alternative UltraSPARC-III approaches could not keep up with the clock rate scaling even with L2 cache and memory latency reductions.

The SAM cache works well for programs having reasonable cache hit rates, but we wanted the performance of all programs to scale. For programs dominated by main memory latency, we use two techniques: a prefetch cache that is accessed in parallel with the L1 data cache, and a low-latency, on-chip memory controller (described later). Analysis showed that many programs dominated by main memory latency shared a common characteristic: the ability to prefetch the memory data well before it's needed by the execution units.

By issuing up to eight in-flight prefetches to main memory, the prefetch cache enables a program to utilize 100% of the available main memory bandwidth without incurring a slowdown due to the main memory latency. The prefetch cache is a 2-Kbyte SRAM organized as 32 entries of 64 bytes and using four-way associativity with an LRU replacement policy. A multiport SRAM design let us achieve a very high throughput. Data can be streamed through the prefetch cache in a manner similar to stream buffers.[7,8] On every cycle, each of two independent read ports supply 8 bytes of data to the pipeline while a third write port fills the cache with 16 bytes.

Other microprocessors, such as the UltraSPARC-II, implement prefetch instructions. Our simulations, however, show that prefetching's full benefit is not realized without the high-bandwidth streaming afforded by the three ports of the prefetch cache. We also included an autonomous stride prefetch engine that tracks the program counters of load instructions and detects when a load instruction is striding through memory. When the prefetch engine detects a striding load, the prefetch engine issues a hardware prefetch independent of any software prefetch. This allows the prefetch cache to be effective even on codes that do not include prefetch instructions.

Our next challenge was to scale the on-chip memory bandwidths. We solved this largely by using two techniques: wave-pipelined SRAM designs for the on-chip caches, and a write cache for store traffic. Wave-pipelining of the caches let us decouple the on-chip memory bandwidth from the latency and independently optimize each characteristic.

Write-caching is an excellent way to reduce the bandwidth due to store traffic.[9] In the UltraSPARC-III we use a write cache to reduce the store traffic bandwidth to the off-chip L2 data cache. The write cache provides other benefits: By being the sole source of on-chip dirty data, the write cache easily handles both multiprocessor and on-chip cache consistency. Error recovery also becomes easier with the write cache, since the write cache keeps all other on-chip caches clean and simply invalidates them when an error is detected.

Sharing a 2-Kbyte SRAM design with the prefetch cache conserved our design effort. Also, it was practical: Simulations showed that the write-back bandwidth of the write cache was relatively insensitive to its size once it was larger than 512 bytes. The bandwidth reduction at 2 Kbytes was equivalent to the store traffic from a write-back, 64-Kbyte, four-way associative data cache. Over 90% of the time the write cache can merge a store into an existing dirty write-cache line.

We use a byte validate policy on the write cache. Rather than reading the data from the L2 cache for the bytes within the line that are not being overwritten, we just keep an individual valid bit for each byte. Not performing the read-on-allocate saves considerable L2 cache bandwidth by postponing a read-modify-write until the write cache evicts a line. Frequently, by eviction time the entire line has been written so the write cache can eliminate the read. We included the write cache in the L2 data cache, and write-cache data can supersede read data from the L2 data cache. We handle this by a byte-merging multiplexer on the incoming L2 cache data bus that can choose either write-cache data or L2 cache data for each byte.

The last benefit of the write cache is in implementing the V9 memory ordering rules. The V9 architecture specifies a memory total store ordering that simplifies the writing of high-performance multiprocessor programs. This model requires that store operations be made visible to all other processors in a multiprocessor system in the original program order. The write cache provides the point of global store visibility in UltraSPARC-III systems. Generally, keeping the requirements of stores (bandwidth, error correction, ordering,

consistency) in a separate cache lets us independently optimize both parts of the on-chip memory system.

Floating-point unit

To meet the cycle time goals for the UltraSPARC-III, we made a concession to latency scaling in the floating-point execution units. Early circuit analysis showed that by using advanced dynamic circuit design, we need add only one additional latency cycle to the floating-point add and multiply units. For numerical floating-point programs, the impact of additional execution latency concerned us less. We were less concerned because previous UltraSPARC generations encouraged unrolled loops to be scheduled for the L2 cache latency, which was eight cycles. Since the previous pipelines had modulo scheduled loops at a multiple of our new latencies, the code schedules would be compatible.

We scaled the floating-point divide latency (in absolute nanoseconds) by using a multiplicative iteration algorithm. Table 5 summarizes the characteristics of the UltraSPARC-III floating-point execution units and compares them to the UltraSPARC-II latencies.

Table 5. UltraSPARC-III (US-III) floating-point units compared to UltraSPARC-II (US-II) latencies.

Operation	600-MHz US-III latency (cycles, ns)	US-III issue rate	300-MHz US-II latency (cycles, ns)
Add/subtract	4, 6.66	1 per cycle	3, 9.99
Multiply	4, 6.66	1 per cycle	3, 9.99
Divide	20, 33.4	1 per 17 cycles	22, 72.6
Square root	24, 40.0	1 per 21 cycles	23, 76.0

External memory and system bus interface

The UltraSPARC-III external memory system includes a large L2 data cache and the main memory system. Integrating, on chip, the control of both these external memory systems was essential in achieving our performance, scalability, and reliability goals.

We built the L2 data cache with eight industry-standard, register-to-register, pipelined static memory chips cycling at one-third of the processor clock rate. The cache controller allows programmable support of 4 or 8 Mbytes of L2 cache. The L2 cache controller accesses off-chip L2 cache SRAMs with a 12-cycle latency to supply a 32-byte cache line to the L1 caches. A 256-bit-wide data bus between the off-chip SRAMs and the microprocessor delivers the full 32 bytes of data needed for an L1 miss in a single SRAM cycle. By placing the tags for the L2 cache on chip, we reduced the latency to main memory with early detection of L2 misses. On-chip tags also enable derivative future designs to build associative L2 caches without a latency penalty. The L2 cache controller accesses on-chip tags in parallel with the start of the off-chip SRAM access and can provide a way-select signal to a late-select address pin on the off-chip data SRAMs.

Dedicating every other cycle of the on-chip L2 cache tags to coherency snoops from other processors provides excellent coherency bandwidth, since the tag SRAM is wave-pipelined at the full 600-MHz target clock rate.

Moving the main memory DRAM controller on chip reduces memory latency, compared to the previous generation, and scales memory bandwidth with the number of processors. The memory controller supports up to 4 Gbytes of SDRAM memory organized as four independent banks. In a multiprocessor system, the SDRAM banks can be interleaved across the per-processor memory controllers. By sizing the SDRAM data bus to be the same as the coherency unit (512 bits), we can minimize the latency to complete a data transfer from memory. This can have a significant performance effect since misses from large caches tend to cluster, and contention for the memory bus from adjacent misses can impact performance. The memory controller has a peak 3.2-Gbyte/sec transfer rate.

We maximized the bandwidth of the processor interface to the system bus by allowing up to 15 outstanding bus transactions from each processor. The outstanding transactions can complete out of order with respect to the original request order. This enables the memory banks in a multiprocessor system to service requests as soon as a bank is available. The processor's bus interface takes care of reordering the data delivery to the pipeline to meet the requirements of the SPARC V9 memory programming model.

The system bus interface architecture was key to meeting the reliability goals we have stated. All processor interfaces use error detection

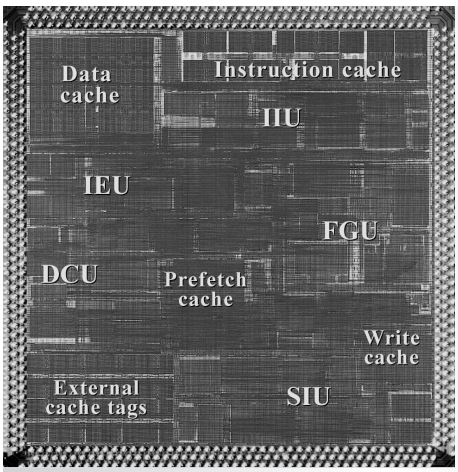

Figure 6. The UltraSPARC-III's major functional units.

and/or correction codes to detect errors as soon as possible. The processor performs error detection on every external chip-to-chip hop to correctly isolate any fault to its source. We also designed an 8-bit-wide "back-door" bus that runs independently from the main system bus. If the system bus has an error, each processor can boot up and run diagnostic programs over the back-door bus to diagnose the problem.

To enable the scaling of multiprocessor systems based on UltraSPARC-III up to 1,000 processors, we included support for in-memory coherency directories on chip. Checking ECC over a 144-bit memory word instead of 72 bits freed up 7 bits of each 144-bit memory word for use by the in-memory directory. The processor must examine the directory state only on each memory access to see if an external agent is required to intervene to complete the memory access. Placing the directory in main memory allows it to automatically expand as memory is added to the system.

Physical design

Physical design grows more challenging with each new processor generation. As the logic complexity grows, circuit counts and related interconnection increases. With interconnections becoming increasingly dominant in design, the block designs must buffer their inputs and outputs, very much like I/O cells for ASIC designs but without the level translation. As clock rates rise faster than available speed increases in base CMOS technology, increasing individual gate complexity without increasing gate delay becomes more urgent. With clock rates, gate count, and total wiring increasing, chip power increases, forcing additional changes in thermal management and electrical distribution schemes. A chip plot outlining the major functional units is shown in Figure 6.

The UltraSPARC-III is flip-chip (solder bump) attached to a multilayered ceramic land grid array package having 750 I/O signals and 450 power bumps. The package has a new cap to mate with an air-cooled heat sink containing a heat pipe structure to control the die temperature. A continuous double grid system on metal layers 5 and 6 provides all this power. This paired grid reduces the power supply loop inductance on the die and provides return current paths for long signal wiring. The grid elements in the sixth metal layer are strapped with what amount to bus bars on metal layer 7. This evens out the power supply resistance drops that would otherwise be seen by the blocks.

The heavy-duty grid concept extends to clock distribution as well. A single distributed clock tree contains a completely shielded grid to reduce both jitter-inducing injected noise and global skew. Each block also has its own shorted, shielded, and buffered clock, further reducing the blocks' local skews.

The circuit methodology we employed is primarily fully static CMOS to simplify the verification requirements for much of the design. Only where speed requirements dictate higher performance did we use dynamic, or domino, designs. Also for verification ease, we placed the dynamic signals only within fully shielded custom cells. To improve the speed enhancement obtained with the dynamic circuits without further increasing their power, we used an overlapping, multiphased, nonblocking clock scheme, similar to that described by Klass.[10]

With clock rates continuing to increase, the part of the cycle time allocated to flip-flops comes under great pressure. To improve this, we designed a new edge-triggered flip-flop.[11] This partially static output, dynamic input flip-flop does not require setup time and is one of the lowest D-to-Q delays for the power and area in use today. The flip-flop design's dynamic input stage effectively allows us to tuck in a full logic stage without increasing the D-to-Q

delay. The noise immunity increases by an input shutoff mechanism that reduces the effective sample time, allowing the design to be used as though it were fully static.

To improve our ability to wire the processor globally, we used an area-based router. This enabled reuse of any block area not needed in the design's lower level for additional top-level wiring. Cost functions for global routing based on noise and timing analysis let us define a specific wire group's width and spacing. Similarly, signal routing within the blocks lets us improve timing and noise margins. MICRO

The UltraSPARC-III is the newest generation of 64-bit SPARC processors to be used in a wide range of Sun systems. The chip will go into production in the fourth quarter of 1999. The architecture enables multiprocessor systems with more than 1,000 processors to be easily built and still achieve excellent performance. Starting at 600-MHz target clock rates, we plan to extend the UltraSPARC-III architecture to achieve clock rates in excess of 1 GHz with UltraSPARC-IV.

References

1. D. Weaver and T. Germond, *The SPARC Architecture Manual*, Version 9, Prentice Hall, Englewood Cliffs, N.J., 1994.
2. N. Wilhelm, "Why Wire Delays Will No Longer Scale for VLSI Chips," Tech. Report TR-95-44, Sun Laboratories, Mountain View, Calif., 1995.
3. K.J. Nowka and M.J. Flynn, *Wave Pipelining of High-Performance CMOS Static RAM*, Tech. Report CSL-TR-94-615, Stanford University, Stanford, Calif., 1994.
4. S.-T. Pan, K. So, and J. Rameh, "Improving Branch Prediction Accuracy using Branch Correlation," *Proc. Fifth Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1992, pp. 76–84.
5. J.E. Smith and J.E. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. Computer Architecture*, ACM Press, p. 36–44.
6. R. Heald et al., "64-KByte Sum-Addressed-Memory Cache with 1.6-ns Cycle and 2.6ns Latency," *IEEE J. Solid-State Circuits*, Nov. 1998, pp. 1,682–1,689.
7. N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 364–373.
8. J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proc. Supercomputing 91*, IEEE Computer Soc. Press, 1991, pp. 176–186.
9. N. Jouppi, "Cache Write Policies and Performance," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1992, pp. 191–201.
10. F. Klass, "A Non-Blocking Multiple-Phase Clocking Scheme for Dynamic Logic," *IEEE Int'l Workshop on Clock Distribution Networks Design, Synthesis, and Analysis*, IEEE Press, Piscataway, N.J., 1997.
11. F. Klass, "Semi-dynamic and Dynamic Flip-flops with Embedded Logic," Digest of Tech. Papers, *1998 Symp. VLSI Circuits*, IEEE Press, 1998, pp. 108–109.

Tim Horel is Megacell group manager for the UltraSPARC-III development team at Sun Microsystems Inc. He previously held product development and engineering positions at AMCC and IBM. Horel has a BSEE from the State University of New York at Buffalo.

Gary Lauterbach is a distinguished engineer at Sun Microsystems Inc. and chief architect of the UltraSPARC-III. In addition to microprocessor design, he has worked on operating system design, CAE tools, process control systems and microwave communication systems. He has a BSEE from the New Jersey Institute of Technology. Lauterbach is a past member of the ACM and the IEEE.

Contact the authors about this article at Sun Microsystems Inc., {tim.horel, gary.lauterbach}@eng.sun.com.

You may find information concerning UltraSPARC-III's comparison to its major competitors in an interview with author Gary Lauterbach in the June 1999 issue of Computer *magazine in the Computing Practices section.*