A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers

The Runway bus, a synchronous, 64-bit, split-transaction, time multiplexed address and data bus, is a new processor-memory-I/O interconnect optimized for one-way to four-way symmetric multiprocessing systems. It is capable of sustained memory bandwidths of up to 768 megabytes per second in a four-way system.

by William R. Bryg, Kenneth K. Chan, and Nicholas S. Fiduccia

The HP 9000 K-class servers and J-class workstations are the first systems to introduce a low-cost, high-performance bus structure named the Runway bus. The Runway bus is a new processor-memory-I/O interconnect that is ideally suited for one-way to four-way symmetric multiprocessing for high-end workstations and midrange servers. It is a synchronous, 64-bit, split-transaction, time multiplexed address and data bus. The HP PA 7200 processor and the Runway bus have been designed for a bus frequency of 120 MHz in a four-way multiprocessor system, enabling sustained memory bandwidths of up to 768 Mbytes per second without external interface or "glue" logic.

The goals for the design of the Runway protocol were to provide a price/performance-competitive bus for one-way to four-way multiprocessing, to minimize interface complexity, and to support the PA 7200 and future processors. The Runway bus achieves these goals by maximizing bus frequency, pipelining multiple operations as much as possible, and using available bandwidth very efficiently, while keeping complexity and pin count low enough so that the bus interface can be integrated directly on the processors, memory controllers, and I/O adapters that connect to the bus.

Overview

The Runway bus features multiple outstanding split transactions from each bus module, predictive flow control, an efficient distributed pipelined arbitration scheme, and a snoopy coherency protocol,[1] which allows flexible coherency check response time.

The design center application for the Runway protocol is the HP 9000 K-class midrange server. Fig. 1 shows a Runway bus block diagram of the HP 9000 K-class server. The Runway bus connects one to four PA 7200 processors with a dual I/O adapter and a memory controller through a shared address and data bus. The dual I/O adapter is logically two separate Runway modules packaged on a single chip. Each I/O adapter interfaces to the HP HSC I/O bus. The memory controller acts as Runway host, taking a central role in arbitration, flow control, and coherency through use of a special client-OP bus.

The shared bus portion of the Runway bus includes a 64-bit address and data bus, master IDs and transaction IDs to tag all transactions uniquely, address valid and data valid signals to specify the cycle type, and parity protection for data and control. The memory controller specifies what types of transactions can be started by driving the special client-OP bus, which is used for flow control and memory arbitration. Distributed arbitration is implemented with unidirectional wires from each module to other modules. Coherency is maintained by having all modules report coherency on dedicated unidirectional wires to the memory controller, which calculates the coherency response and sends it with the data.

Each transaction has a single-cycle header of 64 bits, which minimally contains the transaction type (TTYPE) and the physical address. Each transaction is identified or tagged with the issuing module's master ID and a transaction ID, the combination of which is unique for the duration of the transaction. The master ID and transaction ID are transmitted in parallel to the main address and data bus, so no extra cycles are necessary for the transmission of the master ID and transaction ID.

The Runway bus is a split-transaction bus. A read transaction is initiated by transmitting the encoded header, which includes the address, along with the issuer's master ID and a unique transaction ID, to all other modules. The issuing module then relinquishes control of the bus, allowing other modules to issue their transactions. When the data is available, the module supplying the data, typically memory, arbitrates for the bus, then transmits the data along with the master ID and transaction ID so that the the original issuer can match the data with the particular request.

Write transactions are not split, since the issuer has the data that it wants to send. The single-cycle transaction header is followed immediately by the data being written, using the issuer's master ID and a unique transaction ID.

Fig. 1. *HP 9000 K-class server Runway bus block diagram.*

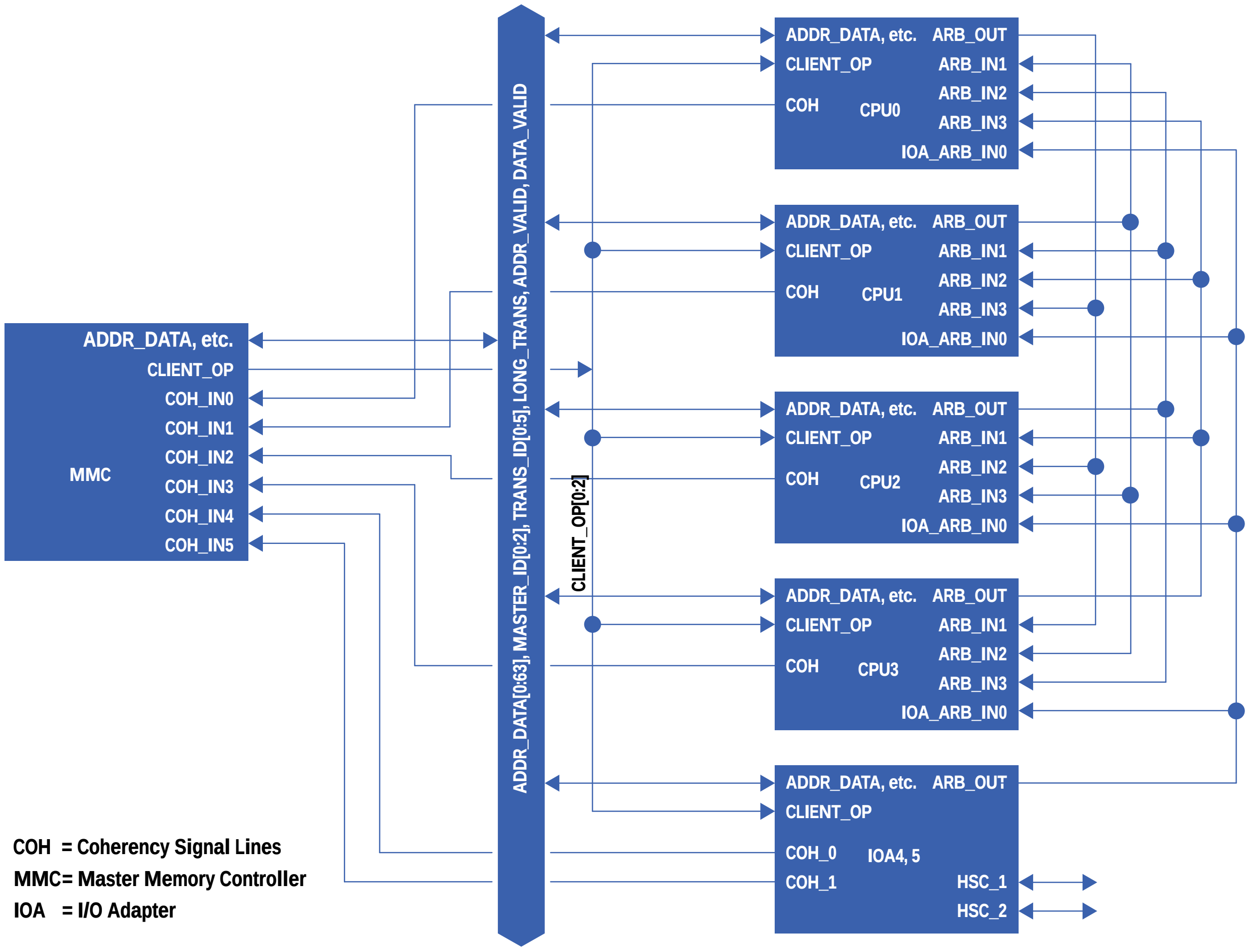
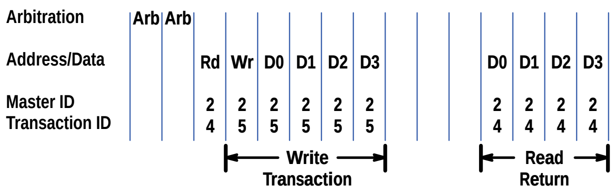


Fig. 2 shows a processor issuing a read transaction followed immediately by a write transaction. Each transaction is tagged with the issuing module's master ID as well as a transaction ID. This combination allows the data response, tagged with the same information, to be directed back to the issuing module without the need for an additional address cycle. Runway protocol allows each module to have up to 64 transactions in progress at one time.

Fig. 2. *A processor issuing a read transaction followed immediately by a write transaction on the Runway bus.*



Arbitration

To minimize arbitration latency without decreasing maximum bus frequency, the Runway bus has a pipelined, two-state arbitration scheme in which the determination of the arbitration winner is distributed among all modules on the bus. Each module drives a unique arbitration request signal and receives other modules' arbitration signals. On the first arbitration cycle, all interested parties assert their arbitration signals, and the memory controller drives the client-OP control signals (see Table I) indicating flow control information or whether all modules are going to be preempted by a memory data return. During the second cycle, all modules evaluate the information received and make the unanimous decision about who has gained ownership of the bus. On the third Runway cycle, the module that won arbitration drives the bus.

With distributed arbitration instead of centralized arbitration, arbitration information only needs to flow once between bus requesters. Using a centralized arbitration unit would require information to flow twice, first between the requester and the arbiter and then between the arbiter and the winner, adding extra latency to the arbitration.

Distributed arbitration on the Runway bus allows latency between arbitration and bus access to be as short as two cycles. Once a module wins arbitration, it may optionally assert a special *long transaction* signal to extend bus ownership for a limited number of cycles for certain transactions. To maximize bus utilization, arbitration is pipelined: while arbitration can be asserted on any cycle, it is effective for the selection of the next bus owner two cycles before the current bus owner releases the bus.

Table I
Client-OP Bus Signals

ANY_TRANS	Any transaction allowed
NO_IO	Any transaction allowed except CPU to I/O
RETURNS_ONLY	Return or response transactions allowed
ONE_CYCLE	Only one-cycle transactions allowed
NONE_ALLOWED	No transactions allowed
MEM_CONTROL	Memory module controls bus
SHARED_RETURN	Shared data return
ATOMIC	Atomic owner can issue any transaction; other modules can only issue response transactions.

Arbitration priority is designed to maintain fairness while delivering optimal performance. The highest arbitration priority is always given to the current bus owner through use of the long transaction signal, so that the current owner can finish whatever transaction it started. The second highest priority is given to the memory controller for sending out data returns, using the client-OP bus to take control of the Runway bus. Since the data return is the completion of a previous split read request, it is likely that the requester is stalled waiting for the data, and the data return will allow the requester to continue processing. The third highest arbitration priority goes to the I/O adapter, which requests the bus relatively infrequently, but needs low latency when it does. Lowest arbitration priority is the processors, which use a round-robin algorithm to take turns using the bus.

The arbitration protocol is implemented in such a way that higher-priority modules do not have to look at the arbitration request signals of lower-priority modules, thus saving pins and reducing costs. A side effect is that low-priority modules can arbitrate for the bus faster than high-priority modules when the bus is idle. This helps processors, which are the main consumers of the bus, and doesn't bother the memory controller since it can predict when it will need the bus for a data return and can start arbitrating sufficiently early to account for the longer delay in arbitration.

Predictive Flow Control

To make the best use of the available bandwidth and greatly reduce complexity, transactions on the Runway bus are never aborted or retried. Instead, the client-OP bus is used to communicate what transactions can safely be initiated, as shown in Table I.

Since the Runway bus is heavily pipelined, there are queues in the processors, memory controllers, and I/O adapters to hold transactions until they can be processed. The client-OP bus is used to communicate whether there is sufficient room in these queues to receive a particular kind of transaction. Through various means, the memory controller keeps track of how much room is remaining in these queues and restricts new transactions when a particular queue is critically full, meaning that the queue would overflow if all transactions being started in the pipeline plus one more all needed to go into that queue. Since the memory controller "predicts" when a queue needs to stop accepting new transactions to avoid overflow, this is called *predictive flow control*.

Predictive flow control increases the cost of queue space by having some queue entries that are almost never used, but the effective cost of queue space is going down with greater integration. The primary benefit of predictive flow control is greatly reduced complexity, since modules no longer have to design in the capability of retrying a transaction that got aborted. This also improves bandwidth since each transaction is issued on the bus exactly once.

A secondary benefit of predictive flow control is faster completion of transactions that must be issued and received in order, particularly writes to I/O devices. If a transaction is allowed to be aborted, a second serially dependent transaction cannot be issued until the first transaction is guaranteed not to be aborted. Normally, this is after the receiving module has had enough time to look at the transaction and check the state of its queues for room, which is at least several cycles into the transaction. With predictive flow control, the issuing module knows when it wins arbitration that the first transaction will issue successfully, and the module can immediately start arbitrating for the second transaction.

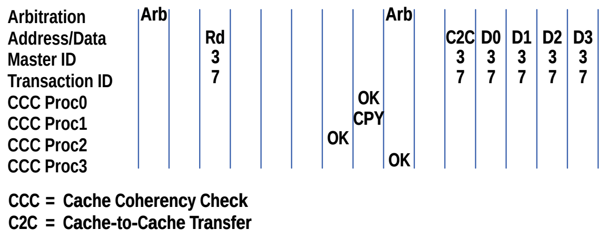
Coherency

The Runway bus provides cache and TLB (translation lookaside buffer) coherence with a snoopy protocol. The protocol maintains cache coherency among processors and I/O modules with a minimum amount of bus traffic while also minimizing the processor complexity required to support snoopy multiprocessing, sometimes at the expense of memory controller complexity.

The Runway bus supports processors with four-state caches: a line may be invalid, shared, private-clean, or private-dirty. An invalid line is one that is not present in cache. A line is shared if it is present in two or more caches. A private line can only be present in one cache; it is private-dirty if it has been modified, private-clean otherwise.

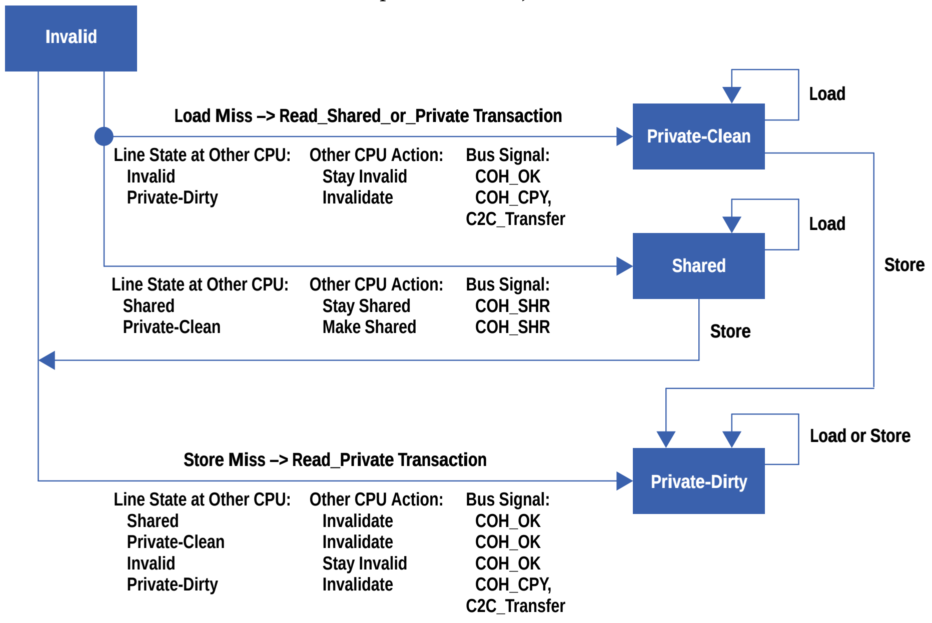
Whenever a coherent transaction is issued on the bus, each processor or I/O device (acting as a third party) performs a snoop, or coherency check, using the virtual index and physical address. Each module then sends its coherency check status directly to the memory controller on dedicated COH signal lines. Coherency status may be COH_OK, which means that either the line is absent or the line has been invalidated. A coherency status of COH_SHR means that the line is either already shared or has changed to shared after the coherency check. A third possibility is COH_CPY, which means the third party has a modified copy of the line and will send the line directly to the requester. Fig. 3 shows a coherent read transaction that hits a dirty line in a third party's cache.

Fig. 3. *A coherent read transaction hits a dirty line in a third party's cache.*



After the memory controller has received coherency status from every module, it will return memory data to the requester if the coherency status reports consist of only COH_OK or COH_SHR. If any module signaled COH_SHR, the memory controller will inform the requester to mark the line shared on the client-OP lines during the data return. If any module signals COH_CPY, however, the memory controller will discard the memory data and wait for the third party to send the modified cache line directly to the requester in a C2C_WRITE transaction. The memory controller will also write the modified data in memory so that the requester can mark the line clean instead of dirty, freeing the requester (and the bus) from a subsequent write transaction if the line has to be cast out. Fig. 4 shows cache state transitions resulting from CPU instructions. Fig. 5 shows transitions resulting from bus snoops.

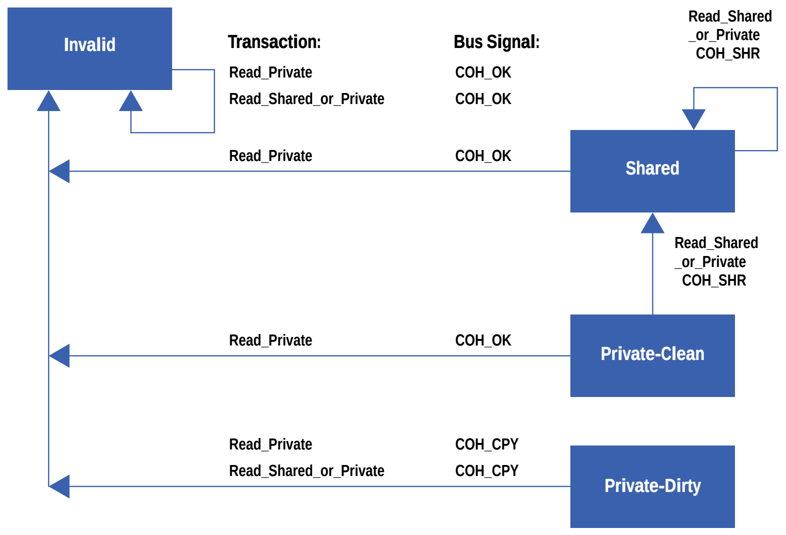
Fig. 4. *Cache state transitions resulting from CPU instructions. The lines with arrows show the transitions of cache state at the originating CPU. The text near each line describes the conditions at other CPUs that caused the transition, as well as the effect on the other CPU's state. For example, from the invalid state, a load miss will always cause a* Read_Shared_or_Private *transaction. The final state for the load miss will be either private-clean (if another CPU had the cache line invalid or private-dirty) or shared (if another CPU had the cache line shared or private-clean).*



Invalid
Load Miss –> Read_Shared_or_Private Transaction
Line State at Other CPU: Invalid Private-Dirty
Other CPU Action: Stay Invalid Invalidate
Bus Signal: COH_OK COH_CPY, C2C_Transfer
Private-Clean
Load
Shared
Load
Store
Line State at Other CPU: Shared Private-Clean
Other CPU Action: Stay Shared Make Shared
Bus Signal: COH_SHR COH_SHR
Store
Load or Store
Store Miss –> Read_Private Transaction
Private-Dirty
Line State at Other CPU: Shared Private-Clean Invalid Private-Dirty
Other CPU Action: Invalidate Invalidate Stay Invalid Invalidate
Bus Signal: COH_OK COH_OK COH_OK COH_CPY, C2C_Transfer

The Runway coherency protocol supports multiple outstanding coherency checks and allows each module to signal coherency status at its own rate rather than at a fixed latency. Each module maintains a queue of coherent transactions received from the bus to be processed in FIFO order at a time convenient for the module. As long as the coherency response is signaled before data is available from the memory controller, delaying the coherency check will not increase memory latency. This flexibility allows CPUs to implement simple algorithms to schedule their coherency checks so as to minimize conflicts with the instruction pipeline for cache access.

Fig. 5. *Cache state transitions resulting from bus coherency checks (snoops).*



Virtual Cache Support

Like all previous HP multiprocessor buses, virtually indexed caches are supported by having all coherent transactions also transmit the virtual address bits that are used to index the processors' caches. The twelve least-significant address bits are the offset within a virtual page and are never changed when translating from a virtual to a physical address. Ten virtual cache index bits are transmitted; these are added to the twelve page-offset bits so that virtual caches up to 4M bytes deep (22 address bits) can be supported.

Coherent I/O Support

Runway I/O adapters take part in cache coherency, which allows more efficient DMA transfers. Unlike previous systems, no cache flush loop is needed before a DMA output and no cache purge is needed before DMA input can begin. The Runway bus protocol defines DMA write transactions that both update memory with new data lines and cause other modules to invalidate data that may still reside in their caches.

The Runway bus supports coherent I/O in a system with virtually indexed caches. I/O adapters have small caches and both generate and respond to coherent transactions. The I/O adapters have a lookup table (I/O TLB) to attach virtual index information to I/O reads and writes, for both DMA accesses and control accesses. For more information see *Article 6*.

Coherent I/O also reduces the overhead associated with the load-and-clear semaphore operation. Since all noninstruction accesses in the system are coherent, semaphore operations are performed in the processors' and I/O adapters' caches. The processor or I/O adapter gains exclusive ownership and atomically performs the semaphore operation in its own cache. If the line is already private in the requester's cache, no bus transaction is generated to perform the operation, greatly improving performance. The memory controller is also simplified because it does not need to support semaphore operations in memory.

The Runway bus has both full-line (32-byte) and half-line (16-byte) DMA input transactions, called WRITE_PURGE and WRITE16_PURGE. Both transactions write the specified amount of data into memory at the specified address, then invalidate any copies of the entire line that may be in a processor's cache. The full-line WRITE_PURGE is the accepted method for DMA input on systems that have coherent I/O, if the full line is being written. The half-line WRITE16_PURGE is used for 16-byte writes if enabled via the fast DMA attribute in the I/O TLB. Software programs the I/O TLB with the fast attribute if it knows that both halves of the line will be overwritten by the DMA. Otherwise, if the I/O TLB does not specify the fast attribute, the I/O adapter uses the slower read private, merge, write back transaction, which will safely merge the DMA data with any dirty processor data. The use of WRITE16_PURGE greatly increases DMA input bandwidth for older, legacy I/O cards that use 16-byte blocks.

Design Trade-offs

To get the best performance from a low-cost interconnect, the bus designers chose a time multiplexed bus, so that the same pins and wires can be used for both address and data. Separate address and data buses would have increased the number of pins needed by about 50%, but would have increased usable bandwidth by only 20%. Since the number of pins on a chip has a strong impact on chip cost, the time multiplexed address and data bus gives us the best trade-off. A smaller number of pins is also important to allow the bus interface to be included on the processor chip instead of requiring one or more separate transceiver chips.

To get the best bandwidth, the designers targeted for the highest bus frequency that could be achieved without requiring dead cycles for bus turnaround. The use of dead cycles would have allowed a higher nominal frequency, but dead cycles would have consumed 20 to 30 percent of the bandwidth, for a net loss.

Bandwidth Efficiency

The Runway bus has a rated raw bandwidth of 960 Mbytes/second, which is derived by taking the width of the bus and multiplying by the frequency: 64 bits $\times$ 120 MHz $\div$ 8 bits/byte = 960 megabytes/second. However, raw bandwidth is an almost meaningless measure of the bus, since different buses use the raw bandwidth with greatly differing amounts of efficiency. Instead, buses should be compared on effective or usable bandwidth, which is the amount of data that is transferred over time using normal transactions.

To deliver as much of the raw bandwidth as possible as usable bandwidth, the designers minimized the percentage of the cycles that were not delivering useful data. Transaction headers, used to initiate a transaction, are designed to fit within a single cycle. Data returns are tagged with a separate transaction ID field, so that data returns do not need a return header. Finally, electrically, the bus is designed so that dead cycles are not necessary for master changeover. The only inherent overhead is the one-cycle transaction header.

For 32-byte lines, both read and write transactions take exactly five cycles on the bus: one cycle for the header and four cycles for data. It doesn’t matter that the read transaction is split. Thus, for the vast majority of the transactions issued on the bus, 80% of the cycles are used to transmit data. The effective bandwidth is 960 Mbytes/s $\times$ 80% = 768 Mbytes/s, which is very efficient compared to competitive buses.

In addition, the Runway bus is able to deliver its bandwidth to the processors that need the bandwidth. Traditional buses typically allow each processor to have only a single outstanding transaction at a time, so that each processor can only get at most about a quarter of the available bandwidth. Runway protocol allows each module—processor or I/O adapter—to have up to 64 transactions outstanding at a time. The PA 7200 processor uses this feature to have multiple outstanding instruction and data prefetches, so that it has fewer stalls as a result of cache misses. When a processor really needs the bandwidth, it can actually get the vast majority of the available 768-Mbyte/s bandwidth.

High Frequency

The Runway protocol is designed to allow the highest possible bus frequency for a given implementation. The protocol is designed so that no logic has to be performed in the same cycle that data is transmitted from one chip to another chip. Any logic put into the transmission cycle would add to the propagation delay and reduce the maximum frequency of the bus. From a protocol standpoint, for this to work, each chip will receive bus signals at the end of one cycle, evaluate those signals in a second cycle and decide what to transmit, then transmit the response in the third cycle.

To maximize implementation frequency, the Runway bus project took a system-level design approach. All modules on the bus and the bus itself were designed together for optimal performance, instead of designing an interface specification permitting any new module to be plugged in as long as it conforms to the specification. We achieved a higher-performance system with the system approach than we could have achieved with an interface specification.

The I/O driver cells for the different modules were designed together and SPICE-simulated iteratively to get the best performance. Since short distances are important, the pinouts of the modules are coordinated to minimize unnecessary crossings and to minimize the worst-case bus paths. See the sidebar: *Runway Bus Design Considerations* for more information on the electrical design of the Runway bus.

The bus can be faster if there are fewer modules on it, since there is less total length of bus and less capacitance to drive. The maximum configuration is limited to six modules—four processors, a dual I/O adapter, and a memory controller—to achieve the targeted frequency of 120 MHz.

Another optimization made to achieve high bus frequency is the elimination of wire-ORs. By requiring that only one module drive a signal in any cycle, some traditionally bused signals that require fast response, such as cache coherency check status, are duplicated, one set per module. Other bused signals that do not require immediate response (e.g., error signals) are more cost-effectively transformed into broadcast transactions. Adapting the Runway protocol to eliminate wire-ORs allowed us to boost the bus frequency by 10 to 20 percent.

Acknowledgments

The authors would like to acknowledge the contributions of many individuals who participated in the definition of the Runway bus protocol: Robert Brooks, Steve Chalmers, Barry Flahive, David Fotland, Craig Frink, Hani Hassoun, Tom Hotchkiss, Larry McMahan, Bob Naas, Helen Nusbaum, Bob Odineal, John Shelton, Tom Spencer, Brendan Voge, John Wickeraad, Jim Williams, John Wood, and Mike Ziegler. In addition, we would like to thank the various engineers and managers of the Computer Technology Laboratory, the General Systems Laboratory, the Chelmsford Systems Laboratory, and the Engineering Systems Laboratory who helped design, verify, build, and test the processors, I/O adapters, and memory controllers necessary to make the Runway bus a reality.

Reference

1. P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, Vol. 23, no. 6, June 1990, pp. 12-25.

▶ Go to Sidebar: *Runway Bus Electrical Design Considerations*
▶ Go to *Article 3*
▶ Go to Table of Contents
▶ Go to HP Journal Home Page