
The AlphaServer 4100 Cached Processor Module Architecture and Design

The DIGITAL AlphaServer 4100 processor module uses the Alpha 21164 microprocessor series combined with a large, module-level backup cache (B-cache). The cache uses synchronous cache memory chips and includes a duplicate tag store that allows CPU modules to monitor the state of each other's cache memories with minimal disturbance to the microprocessor. The synchronous B-cache, which can be easily synchronized with the system bus, permits short B-cache access times for the DIGITAL AlphaServer 4100 system. It also provides a smooth transition from accessing the B-cache to transferring data to or from main memory, without the need for re-synchronization or data buffering.

**Maurice B. Steinman
George J. Harris
Andrej Kocev
Virginia C. Lamere
Roger D. Pannell**

The DIGITAL AlphaServer 4100 series of servers represents the third generation of Alpha microprocessor-based, mid-range computer systems. Among the technical goals achieved in the system design were the use of four CPU modules, 8 gigabytes (GB) of memory, and partial block writes to improve I/O performance.

Unlike the previous generation of mid-range servers, the AlphaServer 4100 series can accommodate four processor modules, while retaining the maximum memory capacity. Using multiple CPUs to share the workload is known as symmetric multiprocessing (SMP). As more CPUs are added, the performance of an SMP system increases. This ability to increase performance by adding CPUs is known as scalability. To achieve perfect scalability, the performance of four CPUs would have to be exactly four times that of a single CPU system. One of the goals of the design was to keep scalability as high as possible yet consistent with low cost. For example, the AlphaServer 4100 system achieves a scalability factor of 3.33 on the Linpack 1000 × 1000, a large, parallel scientific benchmark. The same benchmark achieved 3.05 scalability on the previous-generation platform.¹

The 8-GB memory in the AlphaServer 4100 system represents a factor of four improvement compared with the previous generation of mid-range servers.² The new memory is also faster in terms of the data volume flowing over the bus (bandwidth) and data access time (latency). Again, compared with the previous generation, available memory bandwidth is improved by a factor of 2.7 and latency is reduced by a factor of 0.6.

In systems of this class, memory is usually addressed in large blocks of 32 to 64 bytes. This can be inefficient when one or two bytes need to be modified because the entire block might have to be read out from memory, modified, and then written back into memory to achieve this minor modification. The ability to modify a small fraction of the block without having to extract the entire block from memory results in partial block writes. This capability also represents an advance over the previous generation of servers.

To take full advantage of the Alpha 21164 series of microprocessors, a new system bus was needed. The bus used in the previous generation of servers was not fast

enough, and the cost and size of the bus used in high-end servers was not adaptable to mid-range servers.

Three separate teams worked on the project. One team defined the system architecture and the system bus, and designed the bus control logic and the memory modules.³ The second team designed the peripheral interface (I/O), which consists of the Peripheral Component Interconnect (PCI) and the Extended Industry Standard Architecture (EISA) buses, and its interface to the system bus (I/O bridge).⁴ The third team designed the CPU module. The remainder of this paper describes the CPU module design in detail. Before delving into the discussion of the CPU module, however, it is necessary to briefly describe how the system bus functions.

The system bus consists of 128 data bits, 16 check bits with the capability of correcting single-bit errors, 36 address bits, and some 30 control signals. As many as 4 CPU modules, 8 memory modules, and 1 I/O module plug into the bus. The bus is 10 inches long and, with all modules in place, occupies a space of 11 by 13 by 9 inches. With power supplies and the console, the entire system fits into an enclosure that is 26 by 12 by 17.5 inches in dimension.

CPU Module

The CPU module is built around the Alpha 21164 microprocessor. The module's main function is to provide an extended cache memory for the microprocessor and to allow it to access the system bus.

The microprocessor has its own internal cache memory consisting of a separate primary data cache (D-cache), a primary instruction cache (I-cache), and a second-level data and instruction cache (S-cache). These internal caches are relatively small, ranging in size from 8 kilobytes (KB) for the primary caches to 96 KB for the secondary cache. Although the internal cache operates at microprocessor speeds in the 400-megahertz (MHz) range, its small size would limit performance in most applications. To remedy this, the microprocessor has the controls for an optional external cache as large as 64 megabytes (MB) in size. As implemented on the CPU module, the external cache, also known as the backup cache or B-cache, ranges from 2 MB to 4 MB in size, depending on the size of the memory chips used. In this paper, all references to the cache assume the 4-MB implementation.

The cache is organized as a physical, direct-mapped, write-back cache with a 144-bit-wide data bus consisting of 128 data bits and 16 check bits, which matches the system bus. The check bits protect data integrity by providing a means for single-bit-error correction and double-bit-error detection. A physical cache is one in which the address used to address the cache memory is translated by a table inside the microprocessor that converts software addresses to physical memory

locations. Direct-mapped refers to the way the cache memory is addressed, in which a subset of the physical address bits is used to uniquely place a main memory location at a particular location in the cache. When the microprocessor modifies data in a write-back cache, it only updates its local cache. Main memory is updated later, when the cache block needs to be used for a different memory address. When the microprocessor needs to access data not stored in the cache, it performs a system bus transaction (fill) that brings a 64-byte block of data from main memory into the cache. Thus the cache is said to have a 64-byte block size.

Two types of cache chips are in common use in modern computers: synchronous and asynchronous. The synchronous memory chips accept and deliver data at discrete times linked to an external clock. The asynchronous memory elements respond to input signals as they are received, without regard to a clock. Clocked cache memory is easier to interface to the clock-based system bus. As a result, all transactions involving data flowing from the bus to the cache (fill transactions) and from the cache to the bus (write microprocessor-based system transactions) are easier to implement and faster to execute.

Across the industry, personal computer and server vendors have moved from the traditional asynchronous cache designs to the higher-performing synchronous solutions. Small synchronous caches provide a cost-effective performance boost to personal computer designs. Server vendors push synchronous-memory technology to its limit to achieve data rates as high as 200 MHz; that is, the cache provides new data to the microprocessor every 5 nanoseconds.^{5,6} The AlphaServer 4100 server is DIGITAL's first product to employ a synchronous module-level cache.

At power-up, the cache contains no useful data, so the first memory access the microprocessor makes results in a miss. In the block diagram shown in Figure 1, the microprocessor sends the address out on two sets of lines: the index lines connected to the cache and the address lines connected to the system bus address transceivers. One of the cache chips, called the TAG, is not used for data but instead contains a table of valid cache-block addresses, each of which is associated with a valid bit. When the microprocessor addresses the cache, a subset of the high-order bits addresses the tag table. A miss occurs when either of the following conditions has been met.

1. The addressed valid bit is clear, i.e., there is no valid data at that cache location.
2. The addressed valid bit is set, but the block address stored at that location does not match the address requested by the microprocessor.

Upon detection of a miss, the microprocessor asserts the READ MISS command on a set of four command lines. This starts a sequence of events

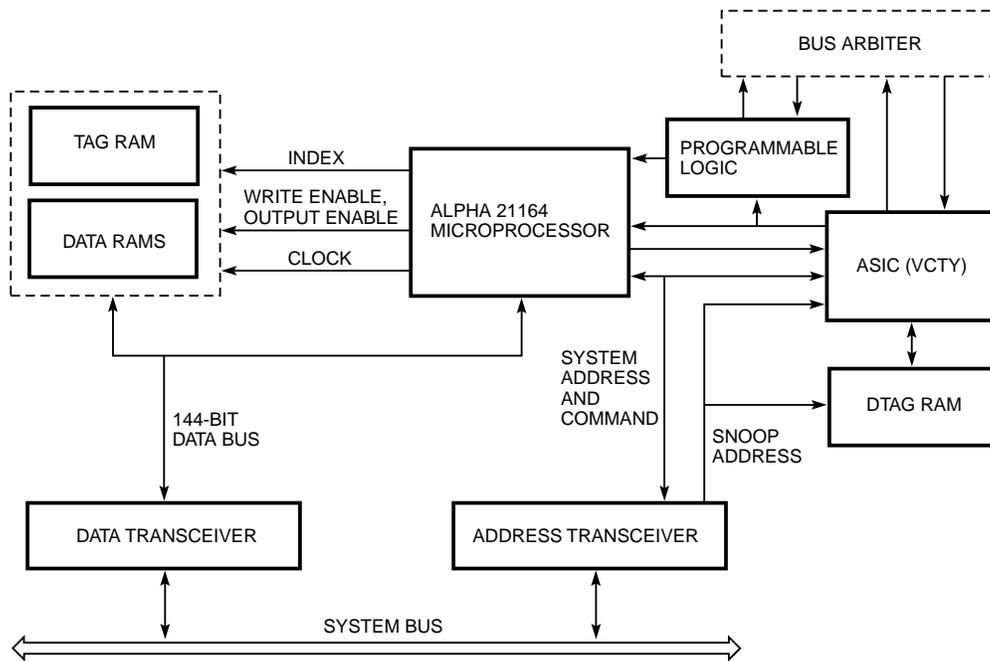


Figure 1
CPU Module

that results in the address being sent to the system bus. The memory receives this address and after a delay (memory latency), it sends the data on the system bus. Data transceivers on the CPU module receive the data and start a cache fill transaction that results in 64 bytes (a cache block) being written into the cache as four consecutive 128-bit words with their associated check bits.

In an SMP system, two or more CPUs may have the same data in their cache memories. Such data is known as shared, and the shared bit is set in the TAG for that address. The cache protocol used in the AlphaServer 4100 series of servers allows each CPU to modify entries in its own cache. Such modified data is known as dirty, and the dirty bit is set in the TAG. If the data about to be modified is shared, however, the microprocessor resets the shared bit, and other CPUs invalidate that data in their own caches. The need is thus apparent for a way to let all CPUs keep track of data in other caches. This is accomplished by the process known as snooping, aided by several dedicated bus signals.

To facilitate snooping, a separate copy of the TAG is maintained in a dedicated cache chip, called duplicate tag or DTAG. DTAG is controlled by an application-specific integrated circuit (ASIC) called VCTY. VCTY and DTAG are located next to each other and in close proximity to the address transceivers. Their timing is tied to the system bus so that the address associated with a bus transaction can easily be applied to the DTAG, which is a synchronous memory device, and the state of the cache at that address can be read out. If that cache location is valid and the address that is stored in the DTAG matches that of the system bus

command (a hit in DTAG), the signal MC_SHARED may be asserted on the system bus by VCTY. If that location has been modified by the microprocessor, then MC_DIRTY is asserted. Thus each CPU is aware of the state of all the caches on the system. Other actions also take place on the module as part of this process, which is explained in greater detail in the section dealing specifically with the VCTY.

Because of the write-back cache organization, a special type of miss transaction occurs when new data needs to be stored in a cache location that is occupied by dirty data. The old data needs to be put back into the main memory; otherwise, the changes that the microprocessor made will be lost. The process of returning that data to memory is called a victim write-back transaction, and the cache location is said to be victimized. This process involves moving data out of the cache, through the system bus, and into the main memory, followed by new data moving from the main memory into the cache as in an ordinary fill transaction. Completing this fill quickly reduces the time that the microprocessor is waiting for the data. To speed up this process, a hardware data buffer on the module is used for storing the old data while the new data is being loaded into the cache. This buffer is physically a part of the data transceiver since each bit of the transceiver is a shift register four bits long. One hundred twenty-eight shift registers can hold the entire cache block (512 bits) of victim data while the new data is being read in through the bus receiver portion of the data transceiver chip. In this manner, the microprocessor does not have to wait until the victim data is transferred along the system bus and into the main memory

before the fill portion of the transaction can take place. When the fill is completed, the victim data is shifted out of the victim buffer and into the main memory. This is known as an exchange, since the victim write-back and fill transactions execute on the system bus in reverse of the order that was initiated by the microprocessor. The transceiver has a signal called BYPASS; when asserted, it causes three of the four bits of the victim shift register to be bypassed. Consequently, for ordinary block write transactions, the transceiver operates without involving the victim buffer.

B-Cache Design

As previously mentioned, the B-cache uses synchronous random-access memory (RAM) devices. Each device requires a clock that loads signal inputs into a register. The RAM operates in the registered input, flow-through output mode. This means that an input flip-flop captures addresses, write enables, and write data, but the internal RAM array drives read output data directly as soon as it becomes available, without regard to the clock. The output enable signal activates RAM output drivers asynchronously, independently of the clock.

One of the fundamental properties of clocked logic is the requirement for the data to be present for some defined time (setup time) before the clock edge, and to remain unchanged for another interval following the clock edge (hold time). Obviously, to meet the setup time, the clock must arrive at the RAM some time after the data or other signals needed by the RAM. To help the module designer meet this requirement, the microprocessor may delay the RAM clock by one internal microprocessor cycle time (approximately 2.5 nanoseconds). A programmable register in the microprocessor controls whether or not this delay is invoked. This delay is used in the AlphaServer 4100 series CPU modules, and it eliminates the need for external delay lines.

For increased data bandwidth, the cache chips used on CPU modules are designed to overlap portions of successive data accesses. The first data block becomes available at the microprocessor input after a delay equal to the BC_READ_SPEED parameter, which is preset at power-up. The following data blocks are latched after a shorter delay, BC_READ_SPEED—WAVE. The BC_READ_SPEED is set at 10 microprocessor cycles and the WAVE value is set to 4, so that BC_READ_SPEED—WAVE is 6. Thus, after the first delay of 10 microprocessor cycles, successive data blocks are delivered every 6 microprocessor cycles. Figure 2 illustrates these concepts.

In Figure 2, the RAM clock at the microprocessor is delayed by one microprocessor cycle. The RAM clock at the RAM device is further delayed by clock buffer and network delays on the module. The address at the microprocessor is driven where the clock would have

occurred had it not been delayed by one microprocessor cycle, and the address at the RAM is further delayed by index buffer and network delays. Index setup at the RAM satisfies the minimum setup time required by the chip, and so does address hold. Data is shown as appearing after data access time (a chip property), and data setup at the microprocessor is also illustrated.

VCTY

As described earlier, a duplicate copy of the microprocessor's primary TAG is maintained in the DTAG RAM. If DTAG were not present, each bus address would have to be applied by the microprocessor to the TAG to decide if the data at this address is present in the B-cache. This activity would impose a very large load on the microprocessor, thus reducing the amount of useful work it could perform. The main purpose of the DTAG and its supporting logic contained in the VCTY is to relieve the microprocessor from having to examine each address presented by the system bus. The microprocessor is only interrupted when its primary TAG must be updated or when data must be provided to satisfy the bus request.

VCTY Operation

The VCTY contains a system bus interface consisting of the system bus command and address signals, as well as some system bus control signals required for the VCTY to monitor each system bus transaction. There is also an interface to the microprocessor so that the VCTY can send commands to the microprocessor (system-to-CPU commands) and monitor the commands and addresses issued by the microprocessor. Last but not least, a bidirectional interface between the VCTY and the DTAG allows only those system bus addresses that require action to reach the microprocessor.

While monitoring the system bus for commands from other nodes, the VCTY checks for matches between the received system bus address and the data from the DTAG lookup. A DTAG lookup is initiated anytime a valid system bus address is received by the module. The DTAG location for the lookup is selected by using system bus Address<21:6> as the index into the DTAG. If the DTAG location had previously been marked valid, and there is a match between the received system bus Address<38:22> and the data from the DTAG lookup, then the block is present in the microprocessor's cache. This scenario is called a cache hit.

In parallel with this, the VCTY decodes the received system bus command to determine the appropriate update to the DTAG and determine the correct system bus response and CPU command needed to maintain system-wide cache coherency. A few cases are illustrated here, without any attempt at a comprehensive discussion of all possible transactions.

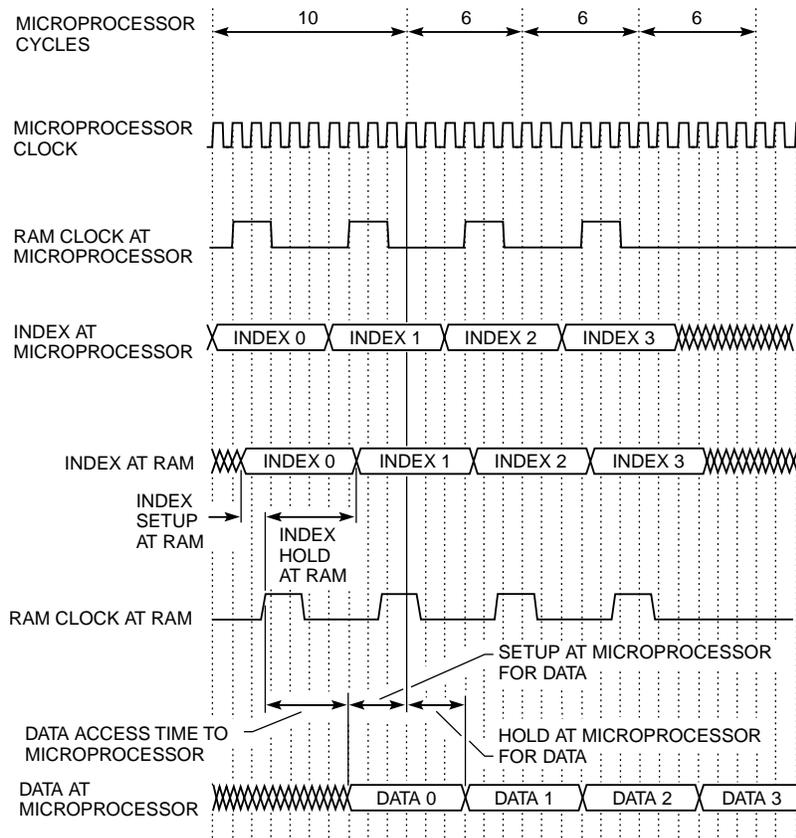


Figure 2
Cache Read Transaction Showing Timing

Assume that the DTAG shared bit is found to be set at this address, the dirty bit is not set, and the bus command indicates a write transaction. The DTAG valid bit is then reset by the VCTY, and the microprocessor is interrupted to do the same in the TAG.

If the dirty bit is found to be set, and the command is a read, the MC_DIRTY_EN signal is asserted on the system bus to tell the other CPU that the location it is trying to access is in cache and has been modified by this CPU. At the same time, a signal is sent to the microprocessor requesting it to supply the modified data to the bus so the other CPU can get an up-to-date version of the data.

If the address being examined by the VCTY was not shared in the DTAG and the transaction was a write, the valid bit is reset in the DTAG, and no bus signals are generated. The microprocessor is requested to reset the valid bit in the TAG. However, if the transaction was not a write, then shared is set in the DTAG, MC_SHARED is asserted on the bus, and a signal is sent to the microprocessor to set shared in the TAG.

From these examples, it becomes obvious that only transactions that change the state of the valid, shared, or dirty TAG bits require any action on the part of the

microprocessor. Since these transactions are relatively infrequent, the DTAG saves a great deal of microprocessor time and improves overall system performance.

If the VCTY detects that the command originated from the microprocessor co-resident on the module, then the block is not checked for a hit, but the command is decoded so that the DTAG block is updated (if already valid) or allocated (i.e., marked valid, if not already valid). In the latter case, a fill transaction follows and the VCTY writes the valid bit into the TAG at that time. The fill transaction is the only one for which the VCTY writes directly into the TAG.

All cycles of a system bus transaction are numbered, with cycle 1 being the cycle in which the system bus address and command are valid on the bus. The controllers internal to VCTY rely on the cycle numbering scheme to remain synchronized with the system bus. By remaining synchronized with the system bus, all accesses to the DTAG and accesses from the VCTY to the microprocessor occur in fixed cycles relative to the system bus transaction in progress.

The index used for lookups to the DTAG is presented to the DTAG in cycle 1 of the system bus transaction. In the event of a hit requiring an update of the

DTAG and primary TAG, the microprocessor interface signal, EV_ABUS_REQ, is asserted in cycles 5 and 6 of that system bus transaction, with the appropriate system-to-CPU command being driven in cycle 6. The actual update to the DTAG occurs in cycle 7, as does the allocation of blocks in the DTAG.

Figure 3 shows the timing relationship of a system bus command to the update of the DTAG, including the sending of a system-to-CPU command to the microprocessor. The numbers along the top of the diagram indicate the cycle numbering. In cycle 1, when the signal MC_CA_L goes low, the system bus address is valid and is presented to the DTAG as the DTAG_INDEX bits. By the end of cycle 2, the DTAG data is valid and is clocked into the VCTY where it is checked for good parity and a match with the upper received system bus address bits. In the event of a hit, as is the case in this example, the microprocessor interface signal EV_ABUS_REQ is asserted in cycle 5 to indicate that the VCTY will be driving the microprocessor command and address bus in the next cycle. In cycle 6, the address that was received from the system bus is driven to the microprocessor along with the SETSHARED command. The microprocessor uses this command and address to update the primary tag control bits for that block. In cycle 7, the control signals DTAG_OE_L and DTAG_WE1_L are asserted low to update the control bits in the DTAG, thus indicating that the block is now shared by another module.

DTAG Initialization

Another important feature built into the VCTY design is a cursory self-test and initialization of the DTAG. After system reset, the VCTY writes all locations of the DTAG with a unique data pattern, and then reads the entire DTAG, comparing the data read versus what was written and checking the parity. A second write-read-compare pass is made using the inverted data pattern. This inversion ensures that all DTAG data bits are written and checked as both a 1 and a 0. In addition, the second pass of the initialization leaves each block of the DTAG marked as invalid (not present in the B-cache) and with good parity. The entire initialization sequence takes approximately 1 millisecond per megabyte of cache and finishes before the microprocessor completes its self-test, avoiding special handling by firmware.

Logic Synthesis

The VCTY ASIC was designed using the Verilog Hardware Description Language (HDL). The use of HDL enabled the design team to begin behavioral simulations quickly to start the debug process.

In parallel with this, the Verilog code was loaded into the Synopsys Design Compiler, which synthesized the behavioral equations into a gate-level design. The use of HDL and the Design Compiler enabled the designers to maintain a single set of behavioral models for the ASIC, without the need to manually enter

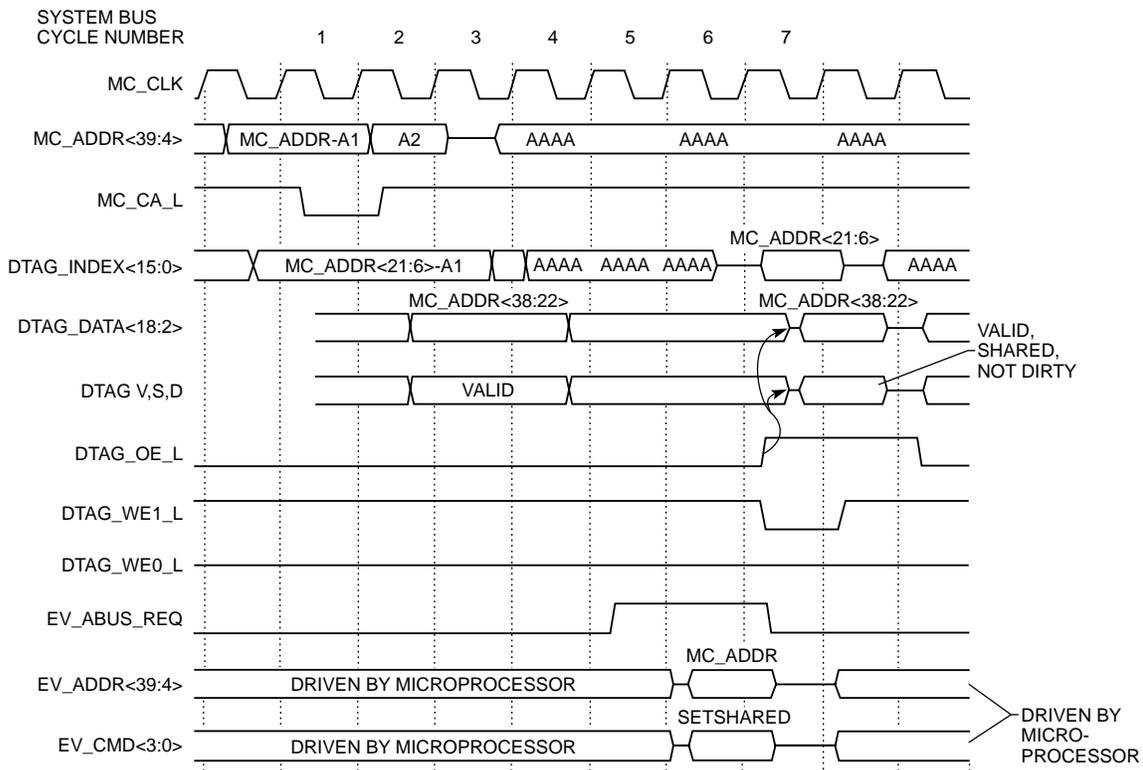


Figure 3
DTAG Operation

schematics to represent the gate-level design. The synthesis process is shown in a flowchart form in Figure 4. Logic verification is an integral part of this process, and the flowchart depicts both the synthesis and verification, and their interaction.

Only the synthesis is explained at this time. The verification process depicted on the right side of the flowchart is covered in a later section of this paper.

As shown on the left side of the flowchart, the logic synthesis process consists of multiple phases, in which the Design Compiler is invoked repeatedly on each subblock of the design, feeding back the results from the previous phase. The Synopsys Design Compiler was supplied with timing, loading, and area constraints to synthesize the VCTY into a physical design that met technology and cycle-time requirements. Since the ASIC is a small design compared to technology capabilities, the Design Compiler was run without an area constraint to facilitate timing optimization.

The process requires the designer to supply timing constraints only to the periphery of the ASIC (i.e., the

I/O pins). The initial phase of the synthesis process calculates the timing constraints for internal networks that connect between subblocks by invoking the Design Compiler with a gross target cycle time of 100 nanoseconds (actual cycle time of the ASIC is 15 nanoseconds). At the completion of this phase, the process analyzes all paths that traverse multiple hierarchical subblocks within the design to determine the percentage of time spent in each block. The process then scales this data using the actual cycle time of 15 nanoseconds and assigns the timing constraints for internal networks at subblock boundaries. Multiple iterations may be required to ensure that each subblock is mapped to logic gates with the best timing optimization.

Once the Design Compiler completes the subblock optimization phase, an industry-standard electronic design interchange format (EDIF) file is output. The EDIF file is postprocessed by the SPIDER tool to generate files that are read into a timing analyzer, Topaz. A variety of industry-standard file formats can be input into SPIDER to process the data. Output files can then

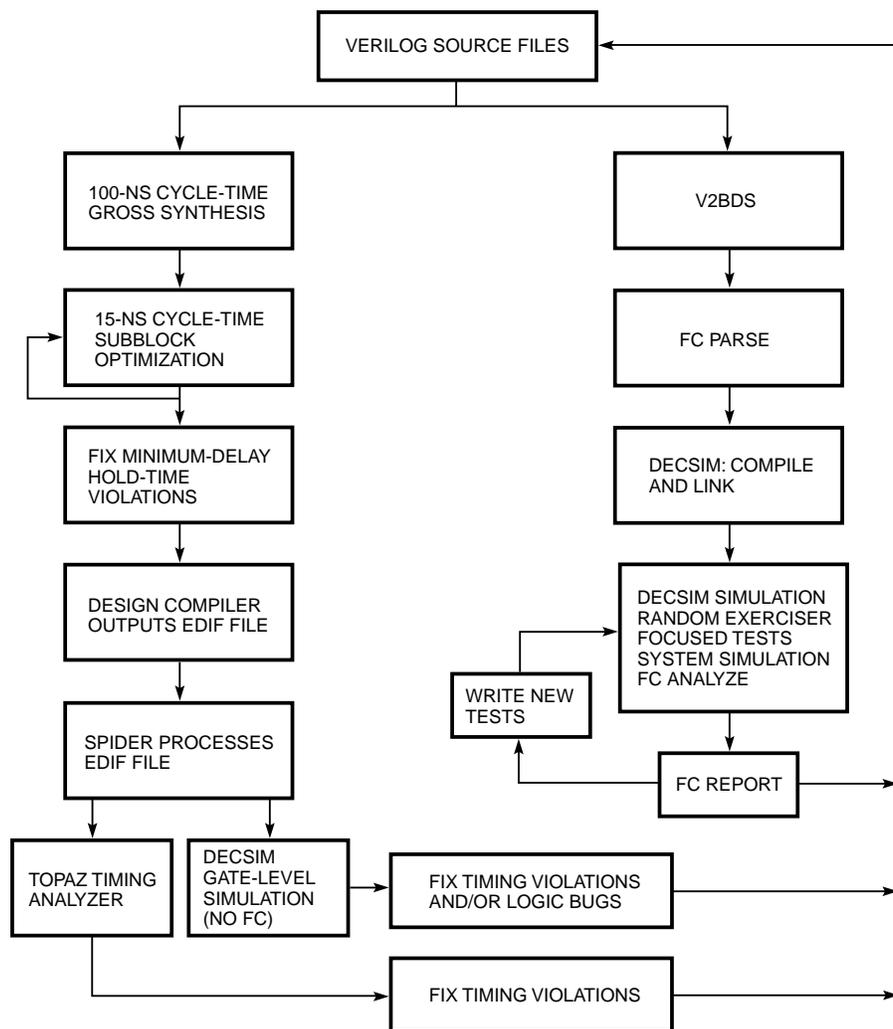


Figure 4
ASIC Design Synthesis and Verification Flow

be generated and easily read by internal CAD tools such as the DECSIM logic simulator and the Topaz timing analyzer.

Topaz uses information contained in the ASIC technology library to analyze the timing of the design as it was mapped by the Design Compiler. This analysis results in output data files that are used to constrain the ASIC layout process and obtain the optimal layout. Logic paths are prioritized for placement of the gates and routing of the connections based on the timing margins as determined by Topaz. Those paths with the least timing margin are given the highest priority in the layout process.

Logic Verification

This section of the paper discusses logic verification and focuses on the use of behavioral model simulation. It should also be noted that once the Design Compiler had mapped the design to gates, SPIDER was also used to postprocess the EDIF file so that DECSIM simulation could be run on the structural design. This process allowed for the verification of the actual gates as they would be built in the ASIC.

The right-hand side of Figure 4 illustrates the logic verification process using a behavioral simulation model. To verify the logic, the system must be performing transactions that exercise all or most of its logic. Ideally, the same software used in physical systems should be run on the design, but this is not practical because of the long run times that would be required. Therefore, specialized software tools are used that can accomplish the task in a shorter time. The verification team developed two such tools: the Random Exerciser and the Functional Checker. They are described in detail in this section.

Random Exerciser

Verification strategy is crucial to the success of the design. There are two approaches to verification testing, directed and random. Directed or focused tests require short run times and target specific parts of the design. To fully test a complex design using directed tests requires a very large number of tests, which take a long time to write and to run. Moreover, a directed test strategy assumes that the designer can foresee every possible system interaction and is able to write a test that will adequately exercise it. For these reasons, random testing has become the preferred methodology in modern logic designs.⁷ Directed tests were not completely abandoned, but they compose only a small portion of the test suite.

Random tests rely on a random sequence of events to create the failing conditions. The goal of the Random Exerciser was to create a framework that would allow the verification team to create random

tests quickly and efficiently without sacrificing flexibility and portability. It consisted of three parts: the test generator, the exerciser code, and the bus monitor.

Test Generator This collection of DECSIM commands randomly generates the test data consisting of addresses (both I/O space and memory space) and data patterns. The user controls the test data generator by setting test parameters. For example, to limit the range of working address space to the uppermost 2 MB of a 4-MB memory space, the working address space parameter is defined as [200000, 400000]. It tells the test generator to choose addresses within that range only—greater than 2 MB and less than 4 MB.

Exerciser Code This code is a collection of routines or sequences of Alpha macrocode instructions to be executed by the microprocessors. Each routine performs a unique task using one of the addresses supplied by the test generator. For example, routine 1 performs a read-verify-modify-write sequence. Routine 2 is similar to routine 1, but it reads another address that is 8 MB away from the original address, before writing to the cache. Since the B-cache is one-way associative, the original address is then evicted from the cache. Lastly, routine 3 performs a lock operation.

Most routines were of the type described above; they used simple load and store instructions. A few routines were of a special type: one generated inter-processor interrupts, others serviced interrupts, another routine generated errors (using addresses to nonexistent memory and I/O space) and checked that the errors were handled properly, and another routine exercised lock-type instructions more heavily.

The activity on the system bus generated by the CPUs was not enough to verify the logic. Two additional system bus agents (models of system bus devices) simulating the I/O were needed to simulate a full system-level environment. The I/O was modeled using so-called commander models. These are not HDL or DECSIM behavioral models of the logic but are written in a high-level language, such as C. From the perspective of the CPU, the commander models behave like real logic and therefore are adequate for the purpose of verifying the CPU module. There were several reasons for using a commander model instead of a logic/behavioral model. A complete I/O model was not yet available when the CPU module design began. The commander model was an evolution of a model used in a previous project, and it offered much needed flexibility. It could be configured to act as either an I/O interface or a CPU module and was easily programmable to flood the system bus with even more activity: memory reads and writes; interrupts to the CPUs by randomly inserting stall cycles in the pipeline; and assertion of system bus signals at random times.

Bus Monitor The bus monitor is a collection of DECSIM simulation watches that monitor the system bus and the CPU internal bus. The watches also report when various bus signals are being asserted and deasserted and have the ability to halt simulation if they encounter cache incoherency or a violation.

Cache incoherency is a data inconsistency, for example, a piece of nondirty data residing in the B-cache and differing from data residing in main memory. A data inconsistency can occur among the CPU modules: for example, two CPU modules may have different data in their caches at the same memory address. Data inconsistencies are detected by the CPU. Each one maintains an exclusive (nonshared) copy of its data that it uses to compare with the data it reads from the test addresses. If the two copies differ, the CPU signals to the bus monitor to stop the simulation and report an error.

The bus monitor also detects other violations:

1. No activity on the system bus for 1,000 consecutive cycles
2. Stalled system bus for 100 cycles
3. Illegal commands on the system bus and CPU internal bus
4. Catastrophic system error (machine check)

The combination of random CPU and I/O activity flooded the system bus with heavy traffic. With the help of the bus monitor, this technique exposed bugs quickly.

As mentioned, a few directed tests were also written. Directed tests were used to re-create a situation that occurred in random tests. If a bug was uncovered using a random test that ran three days, a directed test was written to re-create the same failing scenario. Then, after the bug was fixed, a quick run of the directed test confirmed that the problem was indeed corrected.

Functional Checker

During the initial design stages, the verification team developed the Functional Checker (FC) for the following purposes:

- To functionally verify the HDL models of all ASICs in the AlphaServer 4100 system
- To assess the test coverage

The FC tool consists of three applications: the parser, the analyzer, and the report generator. The right-hand side of Figure 4 illustrates how the FC was used to aid in the functional verification process.

Parser Since DECSIM was the chosen logic simulator, the first step was to translate all HDL code to BDS, a DECSIM behavior language. This task was

performed using a tool called V2BDS. The parser's task was to postprocess a BDS file: extract information and generate a modified version of it. The information extracted was a list of control signals and logic statements (such as logical expressions, if-then-else statements, case statements, and loop constructs). This information was later supplied to the analyzer. The modified BDS was functionally equivalent to the original code, but it contained some embedded calls to routines whose task was to monitor the activity of the control signals in the context of the logic statements.

Analyzer Written in C, the analyzer is a collection of monitoring routines. Along with the modified BDS code, the analyzer is compiled and linked to form the simulation model. During simulation, the analyzer is invoked and the routines begin to monitor the activity of the control signals. It keeps a record of all control signals that form a logic statement. For example, assume the following statement was recognized by the parser as one to be monitored.

(A XOR B) AND C

The analyzer created a table of all possible combinations of logic values for A, B, and C; it then recorded which ones were achieved. At the start of simulation, there was zero coverage achieved.

ABC	Achieved
000	No
001	No
010	No
011	No
100	No
101	No
110	No
111	No

Achieved coverage = 0 percent

Further assume that during one of the simulation tests generated by the Random Exerciser, A assumed both 0 and 1 logic states, while B and C remained constantly at 0. At the end of simulation, the state of the table would be the following:

ABC	Achieved
000	Yes
001	No
010	No
011	No
100	Yes
101	No
110	No
111	No

Achieved coverage = 25 percent

Report Generator The report generator application gathered all tables created by the analyzer and generated a report file indicating which combinations were not achieved. The report file was then reviewed by the verification team and by the logic design team.

The report pointed out deficiencies in the verification tests. The verification team created more tests that would increase the “yes” count in the “Achieved” column. For the example shown above, new tests might be created that would make signals B and C assume both 0 and 1 logic states.

The report also pointed out faults in the design, such as redundant logic. In the example shown, the logic that produces signal B might be the same as the logic that produces signal C, a case of redundant logic.

The FC tool proved to be an invaluable aid to the verification process. It was a transparent addition to the simulation environment. With FC, the incurred degradation in compilation and simulation time was negligible. It performed two types of coverage analysis: exhaustive combinatorial analysis (as was described above) and bit-toggle analysis, which was used for vectored signals such as data and address buses. Perhaps the most valuable feature of the tool was the fact that it replaced the time-consuming and compute-intensive process of fault grading the physical design to verify test coverage. FC established a new measure of test coverage, the percentage of achieved coverage. In the above example, the calculated coverage would be two out of eight possible achievable combinations, or 25 percent.

For the verification of the cached CPU module, the FC tool achieved a final test coverage of 95.3 percent.

Module Design Process

As the first step in the module design process, we used the Powerview schematic editor, part of the Viewlogic CAD tool suite, for schematic capture. An internally developed tool, V2LD, converted the schematic to a form that could be simulated by DECSIM. This process was repeated until DECSIM ran without errors.

During this time, the printed circuit (PC) layout of the module was proceeding independently, using the ALLEGRO CAD tools. The layout process was partly manual and partly automated with the CCT router, which was effective in following the layout engineer’s design rules contained in the DO files.

Each version of the completed layout was translated to a format suitable for signal integrity modeling, using the internally developed tools ADSconvert and MODULEX. The MODULEX tool was used to extract a module’s electrical parameters from its physical description. Signal integrity modeling was performed with the HSPICE analog simulator. We selected HSPICE because of its universal acceptance by the

industry. Virtually all component vendors will, on request, supply HSPICE models of their products. Problems detected by HSPICE were corrected either by layout modifications or by schematic changes. The module design process flow is depicted in Figure 5.

Software Tools and Models

Three internally developed tools were of great value. One was MSPG, which was used to display the HSPICE plots; another was MODULEX, which automatically generated HSPICE subcircuits from PC layout files and performed cross-talk calculations. Cross-talk amplitude violations were reported by MODULEX, and the offending PC traces were moved to reduce coupling. Finally, SALT, a visual PC display tool, was used to verify that signal routing and branching conformed to the design requirements.

One of the important successes was in data line modeling, where the signal lengths from the RAMs to the microprocessor and the transceivers were very critical. By using the HSPICE .ALTER statement and MODULEX subcircuit generator command, we could configure a single HSPICE deck to simulate as many as 36 data lines. As a result, the entire data line group could be simulated in only four HSPICE runs. In an excellent example of synergy between tools, the script capability of the MSPG plotting tool was used to extract, annotate, and create PostScript files of waveform plots directly from the simulation results, without having to manually display each waveform on the screen. A mass printing command was then used to print all stored PostScript files.

Another useful HSPICE statement was .MEASURE, which measured signal delays at the specified threshold levels and sent the results to a file. From this, a separate program extracted clean delay values and calculated the maximum and minimum delays, tabulating the results in a separate file. Reflections crossing the threshold levels caused incorrect results to be reported by the .MEASURE statement, which were easily seen in the tabulation. We then simply looked at the waveform printout to see where the reflections were occurring. The layout engineer was then asked to modify those signals by changing the PC trace lengths to either the microprocessor or the transceiver. The modified signals were then resimulated to verify the changes.

Timing Verification

Overall cache timing was verified with the Timing Designer timing analyzer from Chronology Corporation. Relevant timing diagrams were drawn using the waveform plotting facility, and delay values and controlling parameters such as the microprocessor cycle interval, read speed, wave, and other constants were entered into the associated spreadsheet. All

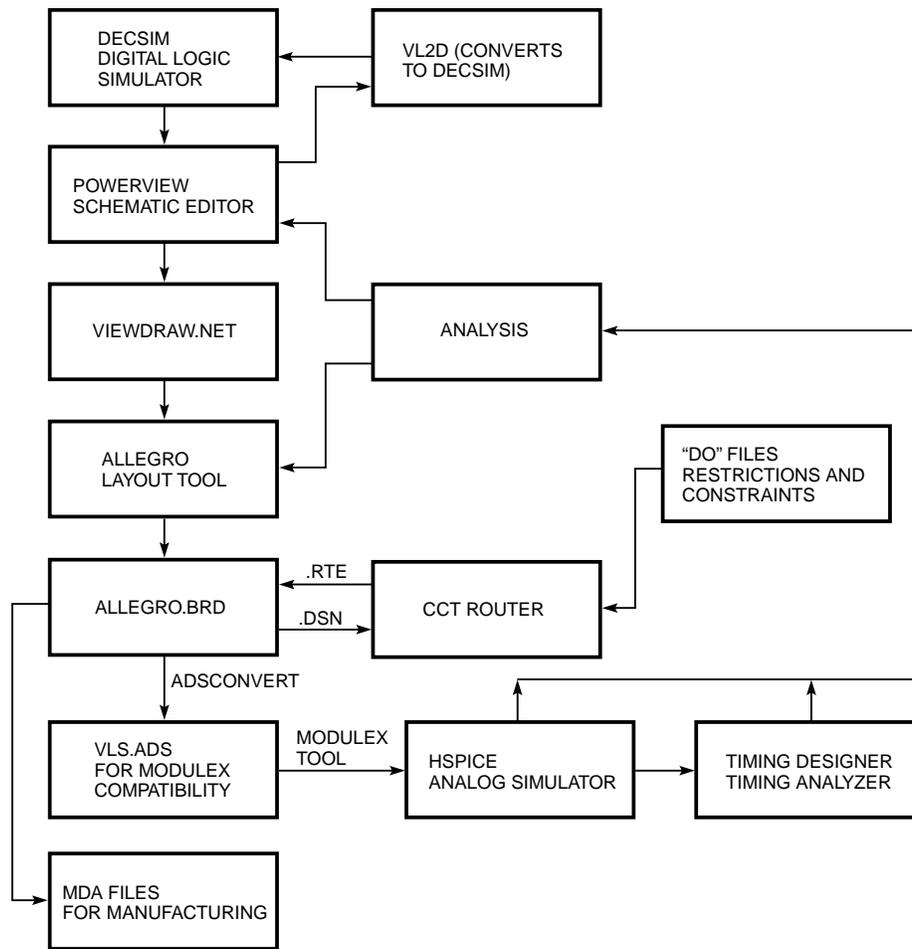


Figure 5
Design Process Flow

delays were expressed in terms of HSPICE-simulated values and those constants, as appropriate. This method simplified changing parameters to try various “what if” strategies. The timing analyzer would instantly recalculate the delays and the resulting margins and report all constraint violations. This tool was also used to check timing elsewhere on the module, outside of the cache area, and it provided a reasonable level of confidence that the design did not contain any timing violations.

Signal Integrity

In high-speed designs, where signal propagation times are a significant portion of the clock-to-clock interval, reflections due to impedance mismatches can degrade the signal quality to such an extent that the system will fail. For this reason, signal integrity (SI) analysis is an important part of the design process. Electrical connections on a module can be made following a direct

point-to-point path, but in high-speed designs, many signals must be routed in more complicated patterns. The most common pattern involves bringing a signal to a point where it branches out in several directions, and each branch is connected to one or more receivers. This method is referred to as treeing.

The SI design of this module was based on the principle that component placement and proper signal treeing are the two most important elements of a good SI design. However, ideal component placement is not always achievable due to overriding factors other than SI. This section describes how successful design was achieved in spite of less than ideal component placement.

Data Line Length Optimization

Most of the SI work was directed to optimizing the B-cache, which presented a difficult challenge because of long data paths. The placement of major module

data bus components (microprocessor and data transceivers) was dictated by the enclosure requirements and the need to fit four CPUs and eight memory modules into the system box. Rather than allowing the microprocessor heat-sink height to dictate module spacing, the system designers opted for fitting smaller memory modules next to the CPUs, filling the space that would have been left empty if module spacing were uniform. As a consequence, the microprocessor and data transceivers had to be placed on opposite ends of the module, which made the data bus exceed 11 inches in length. Figure 6 shows the placement of the major components.

Each cache data line is connected to four components: the microprocessor chip, two RAMs, and the bus transceiver. As shown in Table 1, any one of these components can act as the driver, depending on the transaction in progress.

The goal of data line design was to obtain clean signals at the receivers. Assuming that the microprocessor, RAMs, and the transceiver are all located in-line without branching, with the distance between the two RAMs near zero, and since the positions of the microprocessor and the transceivers are fixed, the only variable is the location of the two RAMs on the data line. As shown in the waveform plots of Figures 7 and 8, the quality of the received signals is strongly affected by this variable. In Figure 7, the reflections are so large that they exceed threshold levels. By contrast, the reflections in Figure 8 are very small, and their waveforms show signs of cancellation. From this it can be inferred that optimum PC trace lengths cause the reflections to cancel. A range of acceptable RAM positions was found through HSPICE simulation. The results of these simulations are summarized in Table 2.

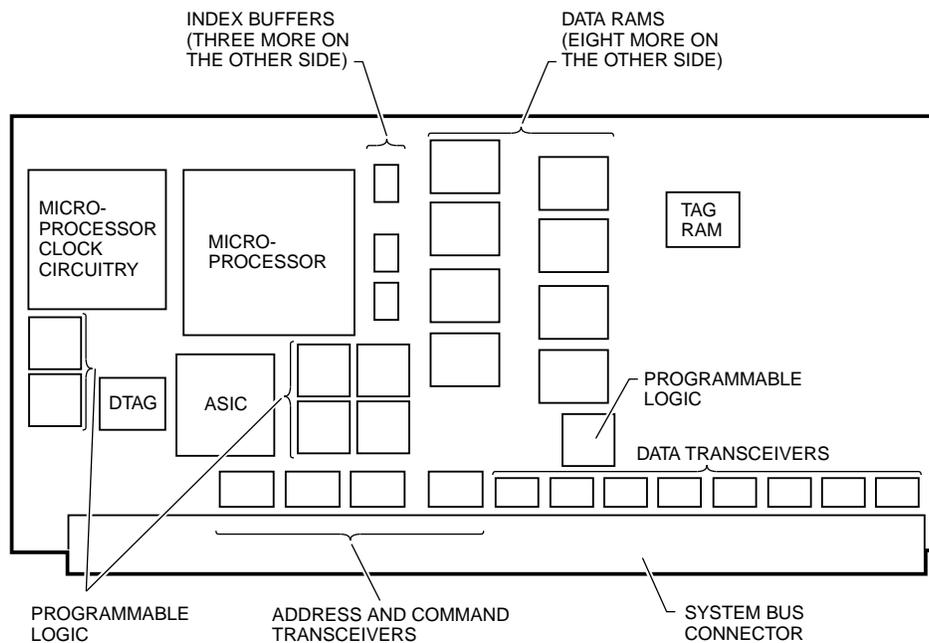


Figure 6
Placement of Major Components

Table 1
Data Line Components

Transaction	Driver	Receiver
Private cache read	RAM	Microprocessor
Private cache write	Microprocessor	RAM
Cache fill	Transceiver	RAM and microprocessor
Cache miss with victim	RAM	Transceiver
Write block	Microprocessor	RAM and transceiver

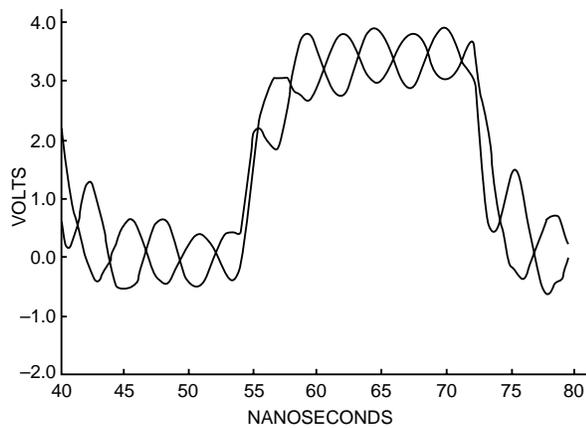


Figure 7
Private Cache Read Showing Large Reflections Due to Unfavorable Trace Length Ratios

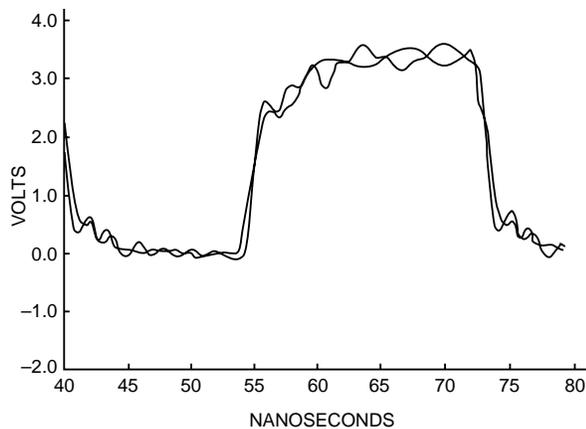


Figure 8
Private Cache Read Showing Reduced Reflections with Optimized Trace Lengths

In the series of simulations given in Table 2, the threshold levels were set at 1.1 and 1.8 volts. This was justified by the use of perfect transmission lines. The lines were lossless, had no vias, and were at the lowest impedance level theoretically possible on the module (55 ohms). The entries labeled SR in Table 2 indicate unacceptably large delays caused by signal reflections recrossing the threshold levels. Discarding these entries leaves only those with microprocessor-to-RAM distance of 3 or more inches and the RAM-to-transceiver distance of at least 6 inches, with the total microprocessor-to-transceiver distance not exceeding 11 inches. The layout was done within this range, and all data lines were then simulated using the network subcircuits generated by MODULEX with threshold levels set at 0.8 and 2.0 volts. These subcircuits included the effect of vias and PC traces run on several signal planes. That simulation showed that all but 12 of the 144 data- and check-bit lines had good signal integrity and did not recross any threshold levels. The failing lines were recrossing the 0.8-volt threshold at the transceiver. Increasing the length of the RAM-to-transceiver segment by 0.5 inches corrected this problem and kept signal delays within acceptable limits.

Approaches other than placing the components in-line were investigated but discarded. Extra signal lengths require additional signal layers and increase the cost of the module and its thickness.

RAM Clock Design

We selected Texas Instruments' CDC2351 clock drivers to handle the RAM clock distribution network. The CDC2351 device has a well-controlled input-to-output delay (3.8 to 4.8 nanoseconds) and 10 drivers in each package that are controlled from one input. The fairly

Table 2
Acceptable RAM Positions Found with HSPICE Simulations

Microprocessor to RAM	PC Trace Length (Inches)		Write Delay (Nanoseconds)		Read Delay (Nanoseconds)			
	RAM to Transceiver		Microprocessor to RAM		RAM to Microprocessor		RAM to Transceiver	
			Rise	Fall	Rise	Fall	Rise	Fall
2	7		0.7	2.3	0.9	SR	1.1	1.4
2	8		0.7	2.7	SR	SR	1.5	1.4
2	9		0.6	3.1	SR	SR	1.7	1.5
3	6		0.9	2.1	1.2	1.1	0.9	1.0
3	7		0.9	2.4	1.0	1.1	1.4	1.3
3	8		0.9	2.9	1.0	1.3	1.5	1.3
4	5		1.1	1.8	1.2	1.4	0.9	SR
4	6		1.3	2.2	1.4	1.4	0.9	1.0
4	7		1.2	2.6	1.3	1.4	1.2	1.2
5	4		1.5	1.7	1.5	1.7	SR	SR
5	5		1.4	2.1	1.8	1.7	SR	SR
5	6		1.6	2.4	1.7	1.4	0.9	1.2

Note: Signal reflections recrossing the threshold levels caused unacceptable delays; these entries were discarded.

long delay through the part was beneficial because, as shown in Figure 2, clock delay is needed to achieve adequate setup times. Two CDC2351 clock drivers, mounted back to back on both sides of the PC board, were required to deliver clock signals to the 17 RAMs.

The RAMs were divided into seven groups based on their physical proximity. As shown in Figure 9, there are four groups of three, two groups of two, and a single RAM. Each of the first six groups was driven by two clock driver sections connected in parallel through resistors in series with each driver to achieve good load sharing. The seventh group has only one load, and one CDC2351 section was sufficient to drive it. HSPICE simulation showed that multiple drivers were needed to adequately drive the transmission line and the load. The load connections were made by short equal branches of fewer than two inches each. The length of the branches was critical for achieving good signal integrity at the RAMs.

Data Line Damping

In the ideal world, all signals switch only once per clock interval, allowing plenty of setup and hold time. In the real world, however, narrow pulses often precede valid data transitions. These tend to create multiple reflections superimposed on the edges of valid signals. The reflections can recross the threshold levels and increase the effective delay, thus causing data errors.

Anticipating these phenomena, and having seen their effects in previous designs, designers included

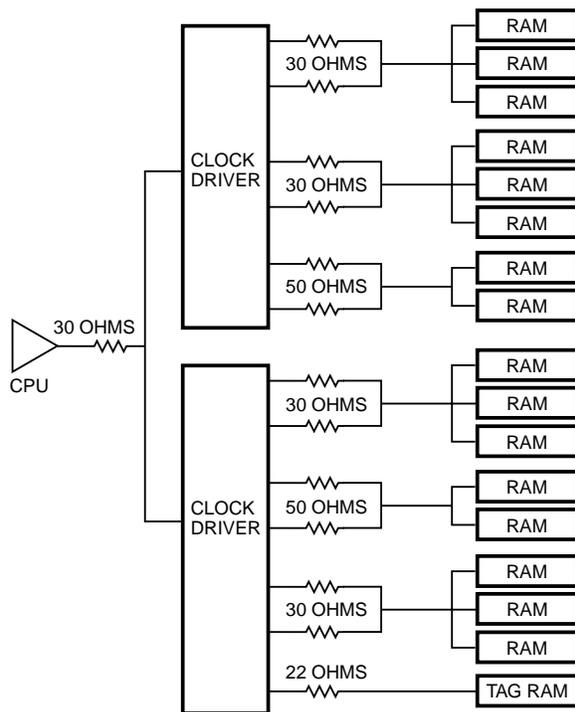


Figure 9
RAM Clock Distribution

series-damping resistors in each cache data line, as shown in Figure 10. Automatic component placement machines and availability of resistors in small packages made mounting 288 resistors on the module a painless task, and the payoff was huge: nearly perfect signals even in the presence of spurious data transitions caused by the microprocessor's architectural features and RAM characteristics. Figure 11 illustrates the handling of some of the more difficult waveforms.

Performance Features

This section discusses the performance of the AlphaServer 4100 system derived from the physical aspects of the CPU module design and the effects of the duplicate TAG store.

Physical Aspects of the Design

As previously mentioned, the synchronous cache was chosen primarily for performance reasons. The architecture of the Alpha 21164 microprocessor is such that its data bus is used for transfers to and from main memory (fills and writes) as well as its B-cache.⁸ As system cycle times decrease, it becomes a challenge to manage memory transactions without requiring wait cycles using asynchronous cache RAM devices. For example, a transfer from the B-cache to main memory (victim transaction) has the following delay components:

1. The microprocessor drives the address off-chip.
2. The address is fanned out to the RAM devices.
3. The RAMs retrieve data.
4. The RAMs drive data to the bus interface device.
5. The bus interface device requires a setup time.

Worst-case delay values for the above items might be the following:

1. 2.6 nanoseconds⁸
2. 5.0 nanoseconds
3. 9.0 nanoseconds
4. 2.0 nanoseconds
5. 1.0 nanoseconds

Total: 19.6 nanoseconds

Thus, for system cycle times that are significantly shorter than 20 nanoseconds, it becomes impossible

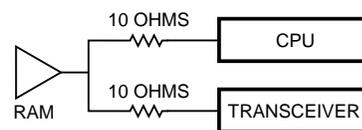


Figure 10
RAM Driving the Microprocessor and Transceiver through 10-ohm Series Resistors



Figure 11
Handling of Difficult Waveforms

to access the RAM without using multiple cycles per read operation, and since the full transfer involving memory comprises four of these operations, the penalty mounts considerably. Due to pipelining, the synchronous cache enables this type of read operation to occur at a rate of one per system cycle, which is 15 nanoseconds in the AlphaServer 4100 system, greatly increasing the bandwidth for data transfers to and from memory. Since the synchronous RAM is a pipeline stage, rather than a delay element, the window of valid data available to be captured at the bus interface is large. By driving the RAMs with a delayed copy of the system clock, delay components 1 and 2 are hidden, allowing faster cycling of the B-cache.

When an asynchronous cache communicates with the system bus, all data read out from the cache must be synchronized with the bus clock, which can add as many as two clock cycles to the transaction. The synchronous B-cache avoids this performance penalty by cycling at the same rate as the system bus.²

In addition, the choice of synchronous RAMs provides a strategic benefit; other microprocessor vendors are moving toward synchronous caches. For example, numerous Intel Pentium microprocessor-based systems employ pipeline-burst, module-level caches using synchronous RAM devices. The popularity of these systems has a large bearing on the RAM industry.⁹ It is in DIGITAL's best interest to follow the synchronous RAM trend of the industry, even for Alpha-based systems, since the vendor base will be larger. These vendors will also be likely to put their efforts into improving the speeds and densities of the best-selling synchronous RAM products, which will facilitate improving the cache performance in future variants of the processor modules.

Effect of Duplicate Tag Store (DTAG)

As mentioned previously, the DTAG provides a mechanism to filter irrelevant bus transactions from the

Alpha 21164 microprocessor. In addition, it provides an opportunity to speed up memory writes by the I/O bridge when they modify an amount of data that is smaller than the cache block size of 64 bytes (partial block writes).

The AlphaServer 4100 I/O subsystem consists of a PCI mother board and a bridge. The PCI mother board accepts I/O adapters such as network interfaces, disk controllers, or video controllers. The bridge provides the interface between PCI devices and between the CPUs and system memory. The I/O bridge reads and writes memory in much the same way as the CPUs, but special extensions are built into the system bus protocol to handle the requirements of the I/O bridge.

Typically, writes by the I/O bridge that are smaller than the cache block size require a read-modify-write sequence on the system bus to merge the new data with data from main memory or a processor's cache. The AlphaServer 4100 memory system typically transfers data in 64-byte blocks; however, it has the ability to accept writes to aligned 16-byte locations when the I/O bridge is sourcing the data. When such a partial block write occurs, the processor module checks the DTAG to determine if the address hits in the Alpha 21164 cache hierarchy. If it misses, the partial write is permitted to complete unhindered. If there is a hit, and the processor module contains the most recently modified copy of the data, the I/O bridge is alerted to replay the partial write as a read-modify-write sequence. This feature enhances DMA write performance for transfers smaller than 64 bytes since most of these references do not hit in the processor cache.⁴

Conclusions

The synchronous B-cache allows the CPU modules to provide high performance with a simple architecture, achieving the price and performance goals of the AlphaServer 4100 system. The AlphaServer 4100

CPU design team pioneered the use of synchronous RAMs in an Alpha microprocessor-based system design, and the knowledge gained in bringing a design from conception to volume shipment will benefit future upgrades in the AlphaServer 4100 server family, as well as products in other platforms.

Acknowledgments

The development of this processor module would not have been possible without the support of numerous individuals. Rick Hetherington performed early conceptual design and built the project team. Pete Bannon implemented the synchronous RAM support features in the CPU design. Ed Rozman championed the use of random testing techniques. Norm Plante's skill and patience in implementing the often tedious PC layout requirements contributed in no small measure to the project's success. Many others contributed to firmware design, system testing, and performance analysis, and their contributions are gratefully acknowledged. Special thanks must go to Darrel Donaldson for supporting this project throughout the entire development cycle.

References

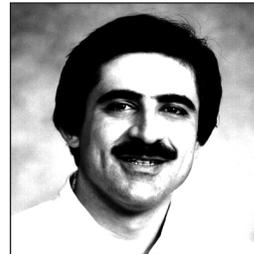
1. DIGITAL AlphaServer Family DIGITAL UNIX Performance Flash (Maynard, Mass.: Digital Equipment Corporation, 1996), <http://www.europe.digital.com/info/performance/sys/unix-svr-flash-9.abs.html>.
2. Z. Cvetanovic and D. Donaldson, "AlphaServer 4100 Performance Characterization," *Digital Technical Journal*, vol. 8, no. 4 (1996, this issue): 3–20.
3. G. Herdeg, "Design and Implementation of the AlphaServer 4100 CPU and Memory Architecture," *Digital Technical Journal*, vol. 8, no. 4 (1996, this issue): 48–60.
4. S. Duncan, C. Keefer, and T. McLaughlin, "High Performance I/O Design in the AlphaServer 4100 Symmetric Multiprocessing System," *Digital Technical Journal*, vol. 8, no. 4 (1996, this issue): 61–75.
5. "Microprocessor Report," *MicroDesign Resources*, vol. 8, no. 15 (1994).
6. *IBM Personal Computer Power Series 800 Performance* (Armonk, N.Y.: International Business Machines Corporation, 1995), <http://ike.engr.washington.edu/news/whitep/ps-perf.html>.
7. L. Saunders and Y. Trivedi, "Testbench Tutorial," *Integrated System Design*, vol. 7 (April and May 1995).
8. *DIGITAL Semiconductor 21164 (366 MHz Through 433 MHz) Alpha Microprocessor Hardware Reference Manual* (Hudson, Mass.: Digital Equipment Corporation, 1996).

9. J. Handy, "Synchronous SRAM Roundup," *Dataquest* (September 11, 1995).

General Reference

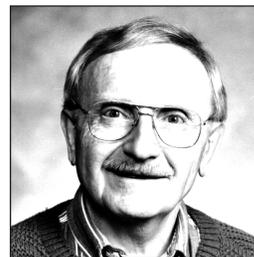
R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, Mass.: Digital Press, 1992).

Biographies



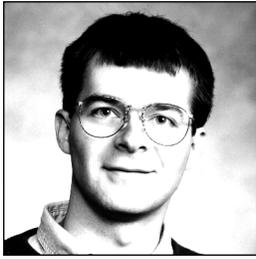
Maurice B. Steinman

Maurice Steinman is a hardware principal engineer in the Server Product Development Group and was the leader of the CPU design team for the DIGITAL AlphaServer 4100 system. In previous projects, he was one of the designers of the AlphaServer 8400 CPU module and a designer of the cache control subsystem for the VAX 9000 computer system. Maurice received a B.S. in computer and systems engineering from Rensselaer Polytechnic Institute in 1986. He was awarded two patents related to cache control and coherence and has two patents pending.



George J. Harris

George Harris was responsible for the signal integrity and cache design of the CPU module in the AlphaServer 4100 series. He joined DIGITAL in 1981 and is a hardware principal engineer in the Server Product Development Group. Before joining DIGITAL, he designed digital circuits at the computer divisions of Honeywell, RCA, and Ferranti. He also designed computer-assisted medical monitoring systems using PDP-11 computers for the American Optical Division of Warner Lambert. He received a master's degree in electronic communications from McGill University, Montreal, Quebec, and was awarded ten patents relating to computer-assisted medical monitoring and one patent related to work at DIGITAL in the area of circuit design.



Andrej Kocev

Andrej Kocev joined DIGITAL in 1994 after receiving a B.S. in computer science from Rensselaer Polytechnic Institute. He is a senior hardware engineer in the Server Product Development Group and a member of the CPU verification team. He designed the logic verification software described in this paper.



Virginia C. Lamere

Virginia Lamere is a hardware principal engineer in the Server Product Development Group and was responsible for CPU module design in the DIGITAL AlphaServer 4100 series. Ginny was a member of the verification teams for the AlphaServer 8400 and AlphaServer 2000 CPU modules. Prior to those projects, she contributed to the design of the floating-point processor on the VAX 8600 and the execution unit on the VAX 9000 computer system. She received a B.S. in electrical engineering and computer science from Princeton University in 1981. Ginny was awarded two patents in the area of the execution unit design and is a co-author of the paper "Floating Point Processor for the VAX 8600" published in this *Journal*.



Roger D. Pannell

Roger Pannell was the leader of the VCTY ASIC design team for the AlphaServer 4100 system. He is a hardware principal engineer in the Server Product Development Group. Roger has worked on several projects since joining Digital in 1977. Most recently, he has been a module/ASIC designer on the AlphaServer 8400 and VAX 7000 I/O port modules and a bus-to-bus I/O bridge. Roger received a B.S. in electronic engineering technology from the University of Lowell.