

# The Alpha Demonstration Unit: A High-performance Multiprocessor for Software and Chip Development

By Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart

## Abstract

*Digital's first RISC system built using the 64-bit Alpha AXP architecture is the prototype known as the Alpha demonstration unit or ADU. It consists of a backplane containing 14 slots, each of which can hold a CPU module, a 64MB storage module, or a module containing two 50MB/s I/O channels. A new cache coherence protocol provides each processor and I/O channel with a consistent view of shared memory. Thirty-five ADU systems were built within Digital to accelerate software development and early chip testing.*

There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things.

– Niccolo Machiavelli, *The Prince*

Introducing a new, 64-bit computer architecture posed a number of challenges for Digital. In addition to developing the architecture and the first integrated implementations, an enormous amount of software had to be moved from the VAX and MIPS (MIPS Computer Systems, Inc.) architectures to the Alpha AXP architecture. Some software was originally written in higher-level languages and could be recompiled with a few changes. Some could be converted using binary translation tools.[1] All software, however, was subject to testing and debugging.

It became clear in the early stages of the program that building an Alpha demonstration unit (ADU) would be of great benefit to software developers. Having a functioning hardware system would motivate software developers and reduce the overall time to market considerably. Software development, even in the most disciplined organizations, proceeds much more rapidly when real hardware is available for programmers. In addition, hardware engineers could exercise early implementations of the processor on the ADU, since a part as complex as the DECchip 21064 CPU is difficult to test using conventional integrated circuit testers.

For these reasons, a project was started in early 1989 to build a number of prototype systems as rapidly as possible. These systems did not require the high levels of reliability and availability typical of Digital products, nor did they need to have low cost, since only a few would be built. They did need to be ready at the same time as the first chips, and they had to be sufficiently robust that their presence would accelerate the overall program.

Digital's Systems Research Center (SRC) in Palo Alto, CA had had experience in building similar prototype systems. SRC had designed and built much of its computing equipment.[2] Being located in Silicon Valley, SRC could employ the services of a number of local medium-volume fabrication and assembly companies without impeding the mainstream Digital engineering and manufacturing groups, which were developing AXP product systems.

The project team was deliberately kept small. Two designers were located at SRC, one was with the Semiconductor Engineering Group's Advanced Development Group in Hudson, MA, and one was a member of Digital's Cambridge Research Laboratory in Cambridge, MA. Although the project team was separated both geographically and organizationally, communication flowed smoothly because the individuals had collaborated on similar projects in the past. The team used a common set of design tools, and Digital's global network made it possible to exchange design information between sites easily. As the project moved from the design phase to production of the systems, the group grew, but at no point did the entire team exceed ten people.

Since multiprocessing capability is central to the Alpha AXP architecture, we decided that the ADU had to be a multiprocessor. We chose to implement a bus-based memory coherence protocol. A high-speed bus connects three types of modules: The CPU module contains one microprocessor chip, its external cache, and an interface to the bus. A storage module contains two 32-megabyte (MB) interleaved banks of dynamic random-access memory (DRAM). The I/O

module contains two 50MB per second (MB/s) I/O channels that are connected to one or two DECstation 5000 workstations, which provide disk and network I/O as well as a high-performance debugging environment. Most of the logic, with the exception of the CPU chip, is emitter-coupled logic (ECL), which we selected for its high speed and predictable electrical characteristics. Modules plug into a 14-slot card cage. The card cage and power supplies are housed in a 0.5-meter (m) by 1.1-m cabinet. A fully loaded cabinet dissipates approximately 4,000 watts and is cooled by forced air. Figures 1 and 2 are photographs of the system and the modules.

### NOTE

Figure 1 (The Alpha Demonstration Unit) is a photograph and is unavailable.

### NOTE

Figure 2 (ADU Modules (a) CPU Module (b) Storage Module (c) I/O Module) is a photograph and is unavailable.

In the remaining sections of this paper, we discuss the backplane interconnect and cache coherence protocol used in the ADU. We then describe the system modules and discuss the design choices. We also present some of the uses we have found for the ADU in addition to its original purpose as a software development vehicle. We conclude with an assessment of the project and its impact on the overall Alpha AXP program.

## Backplane Interconnect

The choice of a backplane interconnect has more impact on the overall design of a multiprocessor than any other decision. Complexity, cost, and performance are the factors that must be balanced to produce a design that is adequate for the intended use. Given the overall purpose of the project, we chose to minimize complexity and maximize performance. System cost is important in a high-volume product, but is not important when only a few systems are produced.

To minimize complexity, we chose a pipelined bus design in which all operations take place at fixed times relative to the time at which a request is issued. To maximize performance, we defined the operations so that two independent transactions can be in progress at once, which fully utilizes the bus.

We designed the bus to provide high bandwidth, which is suitable for a multiprocessor system, and to offer minimal latency. As the CPU cycle time becomes very small, 5 nanoseconds (ns) for the DEC-chip 21064 chip, the main memory latency becomes

an important component of system performance. The ADU bus can supply 320MB/s of user data, but still is able to satisfy a cache read miss in just 200 ns.

### Bus Signals

The ADU backplane bus uses ECL 100K voltage levels. Fifty-ohm controlled-impedance traces, terminated at both ends, provide a well-characterized electrical environment, free from the reflections and noise often present in high-speed systems.

Table 1 lists the signals that make up the bus. The data portion consists of 64 data signals, 14 error correction code (ECC) signals, and 2 parity bits. The ECC signals are stored in the memory modules, but no checking or correction is done by the memories. Instead, the ECC bits are generated and checked only by the ultimate producers and consumers of data, the I/O system and the CPU chip. Secondary caches, the bus, and main memory treat the ECC as uninterpreted data. This arrangement increases performance, since the memories do not have to check data before delivering it. The memory modules would have been less expensive had we used an ECC code that protected a larger block. Since the CPU caches are large enough to require ECC and since the CPU requires ECC over 32-bit words, we chose to combine the two correction mechanisms into one. This decision was consistent with our goal of simplifying the design and improving performance at the expense of increased cost. The parity bits are provided to detect bus errors during address and data transfers. All modules generate and check bus parity.

The module identification signals are used only during system initialization. Each module type is assigned an 8-bit *type code*, and each backplane slot is wired to provide the slot number to the module it contains. Each module in the system reports its type code serially on the nType line during the  $8 \times$  slot number nTypeClk cycles after the deassertion of system reset. A configuration process running on the console processor toggles nTypeClk cycles and observes the nType line to determine the type of module in each backplane slot.

The 100-megahertz (MHz) system clock is distributed radially to each module from a clock generator on the backplane. Constant-length wiring and a strictly specified fan-out path on each module controls clock skew. Since a bus cycle takes two clocks, the phase signal is used to identify the first clock period.

**Table 1**  
**Bus Signals**

Signal Name	Pins	Use
~Data[63..00]	64	Data
~ECC0[6..0]	7	ECC on Data[31..00]
~ECC1[6..0]	7	ECC on Data[63..32]
~P[0]	1	Even Parity over Data[31..00], ECC0[6..0]
~P[1]	1	Even Parity over Data[63..32], ECC1[6..0]
B-shared	1	Cache coherence
B-dirty	1	Cache coherence
Retry	1	Storage module busy
Error	1	Data or address parity error
ArbRequest	8	Arbitration for the bus
Clock	2	100 MHz differential clock
Phase	1	50 MHz Reset 1
nTypeClk	1	Module identification
nType	1	Module identification
nId	4	Module slot number (0..13) set by backplane wiring

*Addressing*

The bus supports a physical address space of 64 gigabytes ( $2^{36}$  bytes). The resolution of a bus address is a 32-byte cache block, which is the only unit of transfer supported; consequently, 31 address bits suffice. One-quarter of the address space is reserved for control registers rather than storage. Accesses to this region are treated specially: CPUs do not store data from this region in their caches, and the target need not supply correct ECC bits.

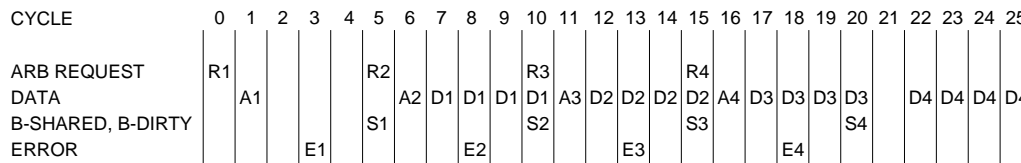
The method used to select the target module of a bus operation is geographic. The initiator sends the target module's slot number with the address during a request cycle. In addition to the 4-bit slot number, the initiator supplies a 3-bit *subnode identifier* with the address. Subnodes are the unit of memory interleaving. The 64MB storage module, for example, contains two independent 32MB subnodes that can operate concurrently.

The geographic selection of the target means that a particular subnode only needs to compare the requested slot and subnode bits with its own slot and subnode numbers to decide whether it is the target. This reduces the time required for the decision compared to a scheme in which the target inspects the address field, but it means that each initiator must maintain a mapping between physical addresses and slot and subnode numbers. This mapping is performed by a RAM in each initiator. For

CPU modules, the RAM lookup does not reduce performance, since the access is done in parallel with the access of the module's secondary cache. The slot-mapping RAMs in each initiator are loaded at system initialization time by the configuration process described previously.

*Bus Operation*

The timing of addresses and data is shown in Figure 3. All data transfers take place at fixed times relative to the start of an operation. Eight of the backplane slots can contain modules capable of initiating requests. These slots are numbered from 0 to 7, but are located at the center of the backplane to reduce the transit time between initiators and targets.



This figure shows the contents of the bus during four read cycles. If requests are made at full rate, the bus is fully occupied with addresses and data. B-shared and B-dirty are sent in the fifth cycle after the arbitration request. If any module detects a parity error during an address cycle, it asserts error two cycles later.

Figure 3 Bus Timing

A bus cycle starts when one of the initiators arbitrates for the bus. The arbitration method guarantees that no initiator can be starved. Each initiator monitors all bus operations and must request only those cycles that it knows the target can accept. Initiators are allowed to arbitrate for a particular target nine or more cycles after that target has started a read, or ten or more cycles after the target has started a write. To arbitrate, an initiator asserts the ArbRequest line corresponding to its current priority. Priorities range from 0 (lowest) to 7 (highest). If a module is the highest priority requester (i.e., no higher priority ArbRequest line than its own is asserted), that module wins the arbitration, and it transmits an address and a command in the next cycle. The winning module sets its priority to zero, and all initiators with priority less than the initial priority of the winner increment their priority *regardless of whether they made a request during the arbitration cycle*. Initially, each initiator's priority is set to its slot number. Priorities are thus distinct initially and remain so over time. This algorithm favors initiators that have not made a recent request, since the priority of such an initiator increases even if it does not make requests. If all initiators make continuous requests, the algorithm provides round-robin servicing, but the implementation is simpler than round robin.

An arbitration cycle is followed by a request cycle. The initiator places an address, node and subnode numbers, and a command on the bus. There are only three commands. A read command requests a 32-byte cache block from memory. The target memory or a cache that contains a more recent copy supplies the data after a five-cycle delay. A write command transmits a 32-byte block to memory, using the same cycles for the data transfer as the read command. Other caches may also take the block and update their contents. A victim write is issued by a CPU module when a block is evicted from the secondary cache. When such an eviction occurs, any other caches that contain the block are guaranteed to contain the same value, so they need not participate in the transfer at all. The block is stored in memory, as in a normal write.

### Cache Coherence

In a multiprocessor system with caches, it is essential that writes done by one processor be made available to the other processors in the system in a timely fashion. A number of approaches to the *cache coherence* problem have appeared in the literature. These approaches fall into two categories, depending on the way in which they handle processor writes. *Invalidation or ownership* protocols require that a processor's cache must acquire an exclusive copy of the block before the write can be done.[3] If another cache contains a copy of the block, that copy is invalidated. On the other hand, *update* protocols maintain coherence by performing write-through operations to other caches that share the block.[2] Each cache maintains enough state to determine whether any other cache shares the block. If the data is not present in another cache, then write through is unnecessary and is not done.

The two protocols have quite different performances, depending on system activity.[4] An update protocol performs better than an invalidation protocol in an application in which data is shared (and written) by multiple processors (e.g., a parallel algorithm executing on several processors). In an invalidation protocol, each time a processor writes a location, the block is invalidated in all other caches that share it. All caches require an expensive miss to retrieve the block when it is next referenced. On the other hand, an update protocol performs poorly in a system in which processes can migrate between processors. With migration, data appears in both caches, and each time a processor writes a location, a write-through operation updates the other cache, even though its CPU is no longer interested in the block. Larger caches with long block lifetimes exacerbate this problem.

### *Coherence Protocol*

The coherence protocol used in the ADU is a hybrid of an update and an invalidation protocol, and like many hybrids, it combines the good features of both parents. The protocol depends on the fact that the CPU chips contain an on-chip cache backed by a much larger secondary cache that monitors all bus operations. Initially, the secondary caches use an update protocol. Caches that contain shared data perform a write-through operation to update the blocks in other caches whenever the associated CPU performs a write. If no other cache shares a block, this write through is unnecessary and is not done. When a secondary cache receives an update (i.e., it observes a write on the bus directed to a block it contains), it has two options. It can invalidate the block and report to the writer that it has done so. If it is the only cache sharing the block, subsequent write-through operations will not occur. Alternatively, it can accept the update and report that it did so, in which case the cache that performed the write-through operation continues to send updates whenever its CPU writes the block.

The actions taken by a cache that receives an update are determined by whether the block is in the CPU's on-chip cache. The secondary cache contains a table that allows it to determine this without interfering with the CPU. If the block is in the on-chip cache, the secondary cache accepts the update and invalidates the block in the on-chip cache. If the block is not in the on-chip cache, the secondary cache block is invalidated. If the block is being actively shared, it will be reloaded by the CPU before the next update arrives, and the block will continue to be shared. If not, the block will be invalidated when the second update arrives.

### *Implementation of the Protocol*

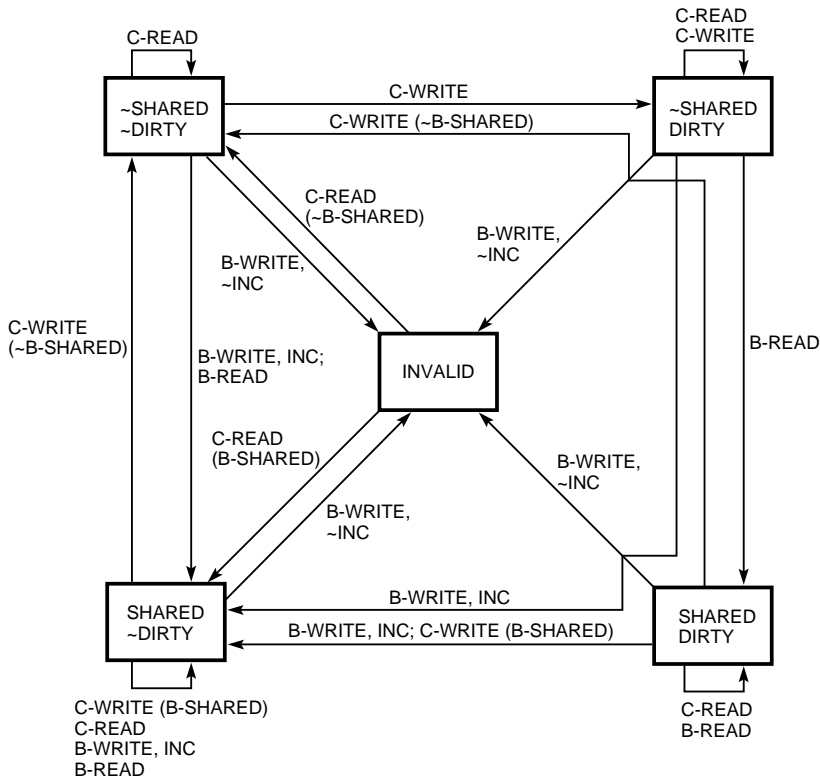
The implementation of the coherence protocol is not complex. The five possible states of a secondary cache block are shown in Figure 4. Initially, all blocks in the cache are marked invalid. Misses in the CPU's on-chip cache cause a bus read to be issued if the block is not in the secondary cache. If the cache block is assigned to another memory location and is dirty (i.e., has been written since it was read from memory), a victim write is issued to evict the block, then a read is issued. Other caches monitor operations on the bus and assert the block-shared (B-shared) signal if they contain the block. If a cache contains a dirty block and it observes a bus read, it asserts B-shared and B-dirty, and supplies the data. B-dirty inhibits the memory's delivery of data.

The CPU's on-chip cache uses a write-through strategy. A CPU write to a shared block in the secondary cache initiates a bus write to update the contents of other caches that share the block. Memory is written, so the block becomes clean. If another cache takes the update, it asserts B-shared, and the initiator's state becomes Shared not (~) Dirty. If no other cache takes the update, either because it does not contain the block or because it decides to invalidate it, then the B-shared signal is not asserted, and the initiator's state becomes ~Shared ~Dirty. The B-shared and B-dirty signals may be asserted by several modules during cycle five of bus operations. The responses are ORed by the open-emitter ECL backplane drivers. More than one cache can contain a block with Shared = true, but only one cache at a time can contain a block with Dirty = true.

Designing the bus interconnect and coherence protocol was an experiment in specification. The informal description required approximately 15 pages of prose to describe the bus. The *real* specification was a multithreaded program that represented the various interfaces at a level of detail sufficient to describe every signal, but, when executed, simulated the components at a higher level. By running this program with sequences of simulated memory requests, we were able to refine the design rapidly and measure the performance of the system before designing any logic. Most design errors were discovered at this time, and prototype system debugging took much less time than usual.

### **System Modules**

In this section, we describe the system modules and the packaging of the ADU. We discuss the design choices made to produce the CPU module, storage modules, and I/O module on schedule. We also discuss applications of the ADU beyond its intended use as a vehicle for software development.



Transitions occur as a result of CPU reads and writes (C-read, C-write) and bus operations initiated by other caches or I/O controllers (B-read, B-write). A C-read or C-write to an invalid block causes a B-read; a C-write to a shared block causes a B-write. The B-shared response indicates that some other cache contains the block. INC indicates that the block is in the CPU's on-chip cache.

Figure 4 Secondary Cache Line States

**CPU Module**

The ADU CPU module consists of a single CPU chip, a 256-kilobyte (KB) secondary cache, and an interface to the system bus. All CPU modules in the system are identical. The CPU modules are not self-sufficient; they must be initialized by the console workstation before the CPU can be enabled.

The CPU module contains extensive test access logic that allows other bus agents to read and write most of the module's internal state. We implemented this logic because we knew these modules would be used to debug CPU chips. Test access logic would help us determine the cause of a CPU chip malfunction and would make it possible for us to introduce errors into the secondary cache to test the error detection and correction capabilities of the CPU chip. This logic was used to perform almost all initialization of the CPU module and was also used to troubleshoot CPU modules after they were fabricated.

The central feature of the CPU module (shown in Figure 5) is the secondary cache, built using 16K by 4 BiCMOS static RAMs. Each of the 16K half-blocks in the data store is 156 bits wide (4 long-words of data, each protected by 7 ECC bits). Each of the 8K entries in the tag store is an 18-bit address (protected by parity) and a 3-bit control field (valid/shared/dirty, also protected by parity). In addition, a secondary cache duplicate tag store, consisting of an 18-bit address and a valid bit (protected by parity), is used as a hint to speed processing of reads and writes encountered on the system bus. Finally, a CPU chip data cache duplicate tag store (protected by parity) functions as an invalidation filter and selects between update and invalidation strategies.

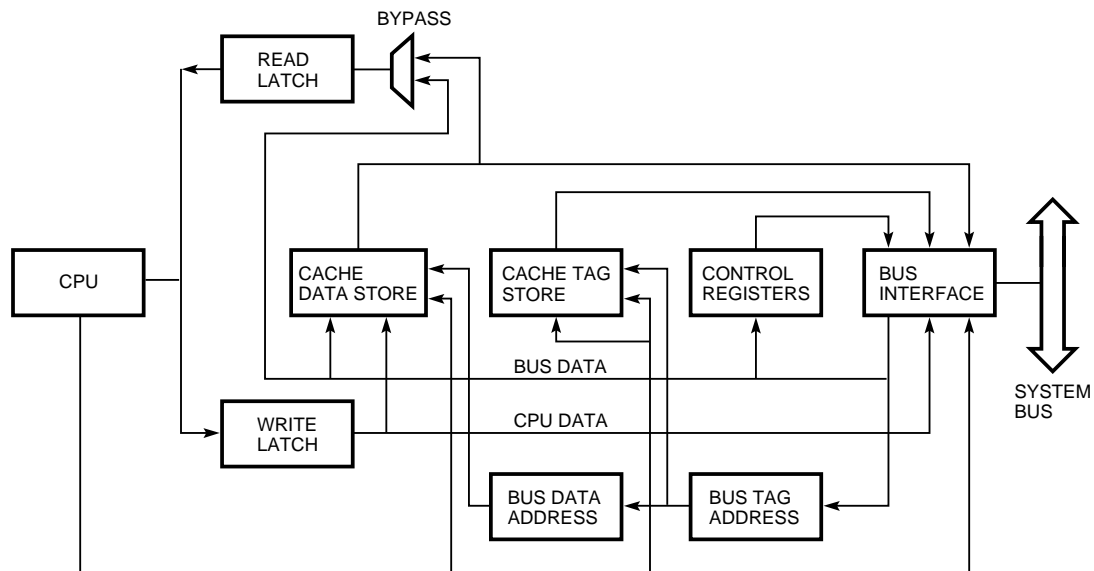


Figure 5 CPU Module

The system bus interface watches for reads and writes on the bus, and looks up each address in the secondary cache. On read hits, it asserts B-shared on the bus, and, if the block is dirty in the secondary cache, it asserts B-dirty and supplies read data to the bus. On write hits, it selects between the invalidate and update strategies, modifies the control field in the secondary cache tag store appropriately, and, if the update strategy is selected, it accepts data from the system bus.

Unlike most bus devices, the CPU module's system bus interface must accept a new address every five cycles. To do this, it is implemented as two independent finite state machines connected together in a pipelined fashion.

The tag state machine, which operates during bus cycles 1 through 5, watches for addresses, performs all tag store reads (in bus cycle 4, just in time to assert B-shared and B-dirty in bus cycle 5), and performs any needed tag store writes (in bus cycle 5). If the tag state machine determines that bus data must be supplied or accepted, it enables the data state machine, and, at the same time, begins processing the next bus request.

The data state machine, which operates during bus cycles 6 through 10, moves data to and from the bus and handles the reading and writing of the secondary cache data store. The highly pipelined nature of the system bus makes reading and writing the data store somewhat tricky. Figure 6a shows

a write hit that has selected the update strategy immediately followed by a read hit that must supply data to the bus. High performance mandates the use of clocked transceivers, which means the secondary cache data store must read one cycle ahead of the bus and must write one cycle behind the bus, resulting in a conflict in bus cycle 11. However, the bus transfers data in a fixed order, so the read will always access quadword 0 of the block, and the write will always access quadword 3 of the block. By implementing the data store as two 64-bit-wide banks, it is possible to handle these back-to-back transactions without creating any special cases, as shown in Figure 6b. This example is typical of the style of design used in the ADU, which eliminates extra mechanisms wherever possible.

The CPU interface handles the arbitration for the secondary cache and generates the necessary reads and writes on the system bus when the CPU secondary cache misses.

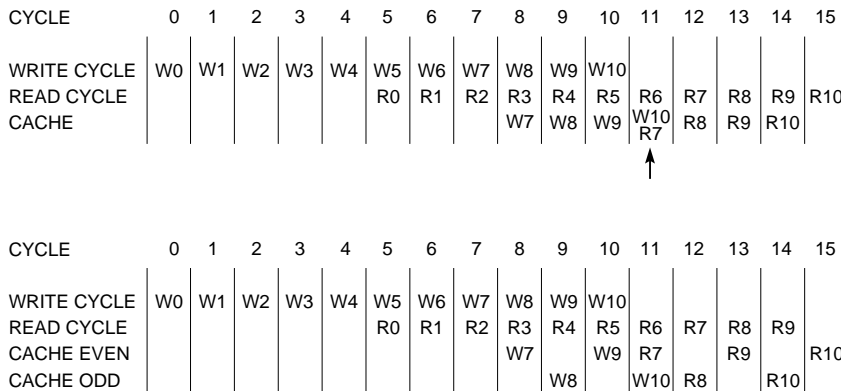


Figure 6 CPU Timing

The CPU chip is supplied with a clock that is not related to the system clock in frequency or phase. This factor made it easier to use both the 100-MHz frequency of the DC227 prototype chip and the 200-MHz frequency of the DECchip 21064 CPU. It also allowed us to vary the operating frequency during CPU chip debugging. However, the data buses connecting the CPU chip to the rest of the CPU module must cross a clock-domain boundary. Perhaps more significant, the secondary cache tag and data stores have two asynchronous sources of control, since the CPU contains an integrated secondary cache controller.

The bidirectional data bus of the CPU chip is converted into the unidirectional data buses used by the rest of the CPU module by transparent cut-off latches. These latches, which are located in a ring surrounding the CPU, also convert the quasi-ECL levels generated by the CPU chip into true ECL levels for the rest of the CPU module. These latches are normally held open, so the CPU chip is, in effect, connected directly to the secondary cache tag and data RAMs. Control signals from the CPU chip's integrated secondary cache controller are simply ORed into the appropriate secondary cache RAM drivers.

These latches are also used to pass data across the two-clock-domain boundary. Normally all latches are open. On reads, logic in the CPU chip clock domain closes all the latches and sends a read request into the bus clock domain. Logic in the bus clock domain obtains the data, writes both the secondary cache and the read latches, and sends an acknowledgment back into the CPU chip clock domain. Logic in the CPU chip clock domain accepts the first half-block of the data, opens the first read latch, accepts the second half-line of the data, and opens all remaining latches. Writes are similar. Logic in the

CPU chip clock domain writes the first half-line into the write latch, makes the second half-line valid (behind the latch), and sends a write request into the bus clock domain. Logic in the bus clock domain accepts the first half-line of data, opens the write latch, accepts the second half-block of data, and sends an acknowledgment back into the CPU chip clock domain.

Logic in the CPU chip clock domain controls all latches. Only two signals pass through synchronizers: a single request signal passes from the CPU chip clock domain to the bus clock domain, and a single acknowledge signal passes from the bus clock domain to the CPU chip clock domain.

The secondary cache arbitration scheme is unconventional because the system bus has no stall mechanism. If a read or a write appears on the system bus, the bus interface must have unconditional access to the secondary cache; it cannot wait for the CPU to finish its current cycle. In fact, the bus interface cannot detect if a cycle is in progress in the CPU chip's integrated cache controller.

Nevertheless, all events in the system bus interface occur at fixed times with respect to bus arbitration cycles. As a result, the system bus interface can supply a busy signal to the CPU interface, which allows it to predict the bus interface's use of the secondary cache in the immediate future. The CPU interface, therefore, waits until the secondary cache can be accessed without conflict and then performs its cycle without additional checking. This waiting is performed by the CPU chip's integrated secondary cache controller for some cycles, and by logic in the CPU interface running in the bus clock domain for other cycles. To reduce latency, the CPU reads the secondary cache while waiting, and ignores the data if it is not yet valid.

All operations use ownership of the system bus as an interlock. For example, if the CPU writes to a location in the secondary cache that is marked as shared, the CPU interface acquires the system bus, and then updates the secondary cache at the same time as it broadcasts the write. This does not eliminate all race conditions; in particular, it allows a dirty secondary cache block to be invalidated by a system bus write while the CPU interface is waiting to acquire the bus to write the block to memory. This is easily handled, however, by having the CPU interface generate a signal (`always_update`) that insists that the system bus interface select the update strategy.

The combination of arbitration by predicting future events and the use of the system bus as an interlock makes the CPU module's control logic extremely simple. The bus interface and the CPU interface have no knowledge of one another beyond the busy and `always_update` signals. Since no complicated interactions between the CPU and the bus exist, no time-consuming simulations of the interactions needed to be performed, and we had none of the difficult-to-track-down bugs that are usually associated with multiprocessor systems.

The CPU module contains a number of control registers. The bus cycles that read and write these registers are processed by the system bus interface as ordinary, but somewhat degenerate, cases. The local CPU accesses its local registers over the system bus, using ordinary system bus reads and writes, so no special logic is needed to resolve race conditions.

To keep pace with our schedule, we arranged for most of the system to be debugged before the CPU chip arrived. By using a suitably wired integrated circuit test clip, we could place commands onto the CPU chip's command bus and verify the control signals with an oscilloscope. The results of these tests left us fairly confident that the system worked before the first chip arrived.

We resumed testing the CPU module after the CPU chip was installed. We placed short (three to five instructions) programs into main memory, enabled the CPU chip for a short time, then inspected the secondary cache (using the CPU module's test access logic) to examine the results.

Eventually we connected an external pulse generator to the CPU chip's clock and an external power supply to the CPU chip. These modifications permitted us to vary both the operating frequency and the operating voltage of the CPU chip. By using a pulse generator and a power supply that could be remotely controlled by another computer, we were able to write simple programs that could run CPU chip diagnostics, without manual intervention, over

a wide range of operating conditions. This greatly simplified the task of collecting the raw data needed by the chip designers to verify the critical paths in the chip.

#### *Storage Modules*

The ADU's storage modules must provide high bandwidth, both to service cache misses and to support demanding I/O devices. More important, they must provide low latency, since in the case of a cache miss, the processor is stalled until the miss is satisfied. It is also important to provide a modest amount of memory interleaving. Although the bus protocol allows only two memory subnodes to be active at once, higher interleave increases the probability that a module will be free when a memory request is issued.

Each storage module is organized as two independent bus subnodes, so that even in a system with one module, memory is two-way interleaved. Each of the subnodes consists of four banks, each of which stores two longwords of data and their associated error correction bits. With 1-megabit (Mb) RAM chips, the capacity of each module is 64MB. Figure 7 shows the organization of the storage module. The module consists of two independent subnodes, each with four banks of storage. Control signals are pipelined through the banks so that the module can deliver or accept a 64-bit data word (plus ECC) every 20 ns. With the exception of the DRAM interface signals, all signals are ECL levels. The G014 gallium arsenide (GaAs) driver chip improves performance by allowing parallel termination of the DRAM address lines.

A memory cycle consists of a five-bus-cycle access period followed by four bus cycles of data transfer. Each data transfer cycle moves two 39-bit longwords between the module and the backplane bus, for a total of 32 data bytes per memory cycle. This is the size of a CPU module cache block. A read operation takes 10 bus cycles to complete, but a write requires 11 cycles.

Since a data rate of 1 word every 20 ns is beyond the capabilities of even the fastest nibble-mode RAMs, we needed an approach that did not require each RAM to provide more than 1 bit per access. We chose to pipeline the four banks of each subnode. Each of the four banks contributes only one 78-bit word to the block. The banks are started sequentially, with a one-cycle delay between each bank.

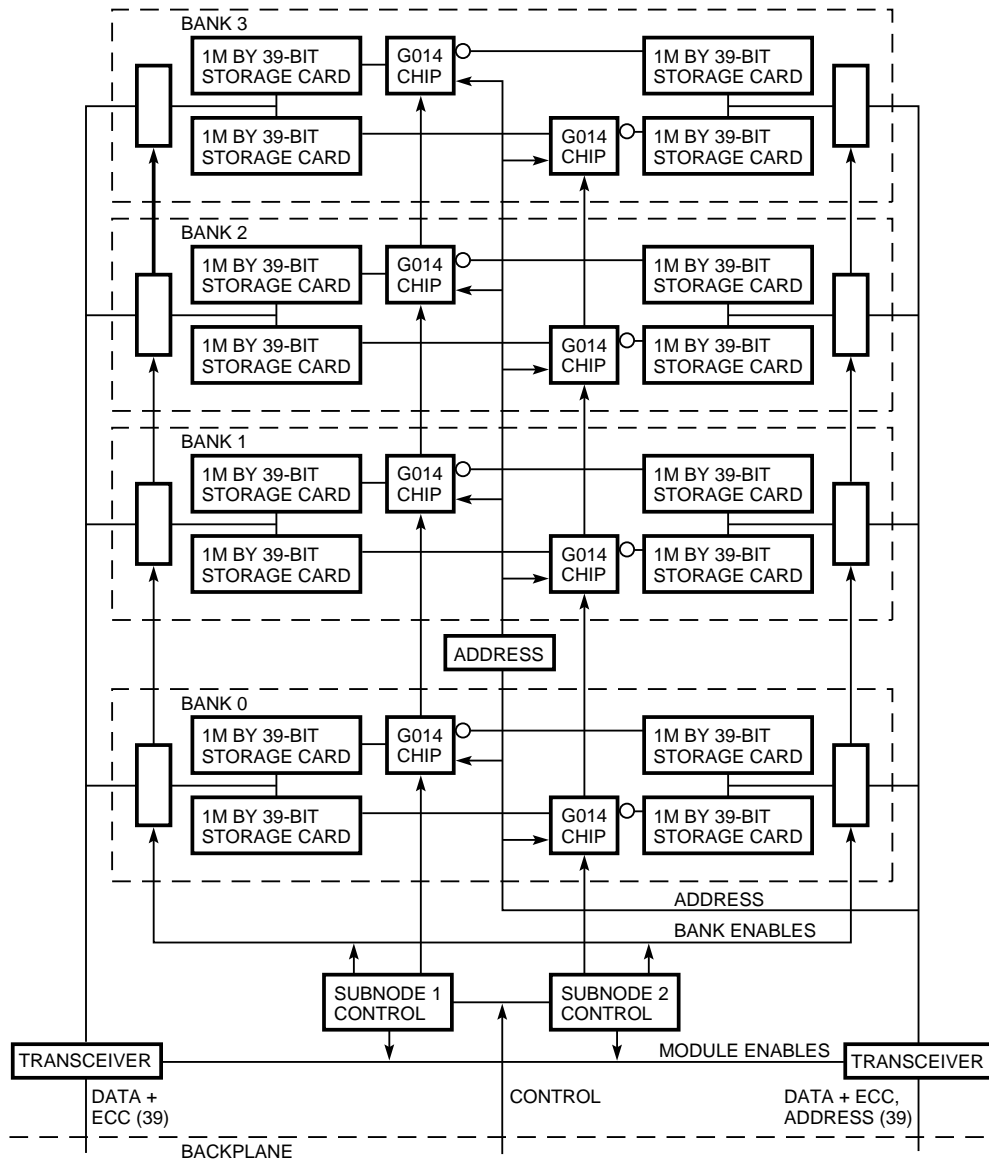


Figure 7 ADU Storage Module

The high performance of the storage module is achieved by maintaining ECL levels and using ECL 100K components wherever possible. The RAM I/O pin levels are converted to ECL levels by latching transceivers associated with each bank. Fortunately, the timing of accesses to the two subnodes of a module makes it possible to share these transceivers between the same banks of the module's two subnodes.

The DRAM chips are packaged on small daughter cards that plug into connectors on both sides of the

main array module. There are 2 daughter cards for each bank within a subnode, for a total of 16 daughter cards per module. The DRAM address and control lines are carried on controlled impedance traces. Since each of the 39 DRAMs on an address line represents a capacitive load of approximately 8 picofarads, the loaded impedance of the line is about 30 ohms.

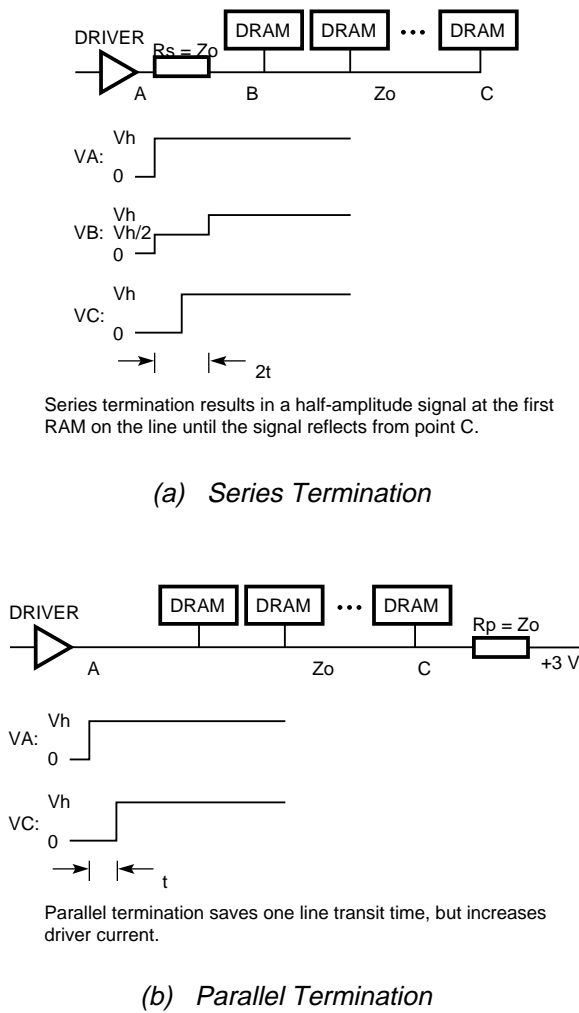


Figure 8 Address Line Termination

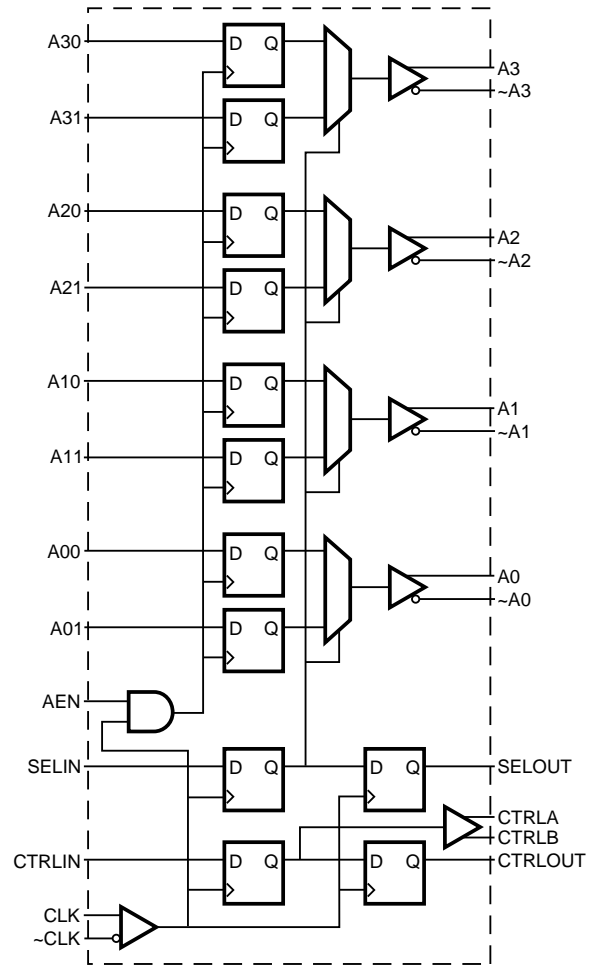


Figure 9 Address and Control Driver

The usual approach to driving the address and control lines of a RAM array uses series termination, as shown in Figure 8a. This arrangement has the advantage that the driver current is reduced, since the load impedance seen by the driver ( $R_s + Z_0$ ) is twice that of the loaded transmission line ( $Z_0$ ). Unfortunately, the RAM access time is increased, because the signal from the driver ( $V_b$ ) must propagate to the far end of the line, be reflected, and return to the driver before the first RAM on the line sees a full-amplitude signal. Since the capacitive loading added by the RAM pins lowers the signal propagation velocity in addition to reducing the impedance, the added delay can be a significant fraction of the overall cycle time.

Since low latency was a primary design goal, we chose parallel termination of the RAM address and control lines, as shown in Figure 8b. Each address line is terminated to +3 volts with a series resistor ( $R_s$ ) of 33 ohms, slightly higher than the line impedance. In this configuration, each line's driver must sink a current of almost 0.1 ampere. Since no commercial chip could meet this requirement at the needed speed, we commissioned a semicustom GaAs chip.[5]

As shown in Figure 9, each GaAs chip contains a register for eight address bits, row/column address multiplexing and high current drivers for the RAM address lines, and a driver for one of the three RAM control signals (RAS, CAS, Write). To reduce the current switched by each chip, each address bit drives two output pins. One pin carries true data, and the other is complemented. The total current is therefore constant. Each pin drives one of the two RAM modules of a bank. A total of three GaAs chips is required per bank. In the present module, with 1M- by 1-bit RAM chips, only 10 of the 12 address drivers are used, so the system can be easily expanded to make use of 16M RAMs.

The storage module contains only a small amount of control logic. This logic generates the control signals for the RAMs and the various transceivers that route data from the backplane to each bank. This logic also generates the signals needed to refresh the RAMs and to assert the retry signal if another node attempts to access the module while it is refreshing itself.

### *I/O Module*

The I/O module for the ADU contains two 50MB/s I/O channels and a local CPU subsystem. The I/O channels connect to one or two DECstation 5000 workstations, which act as I/O front-end processors and also provide console and diagnostic functions. The local CPU subsystem is used to provide interval

timer and time-of-day clock services to ADU processors.

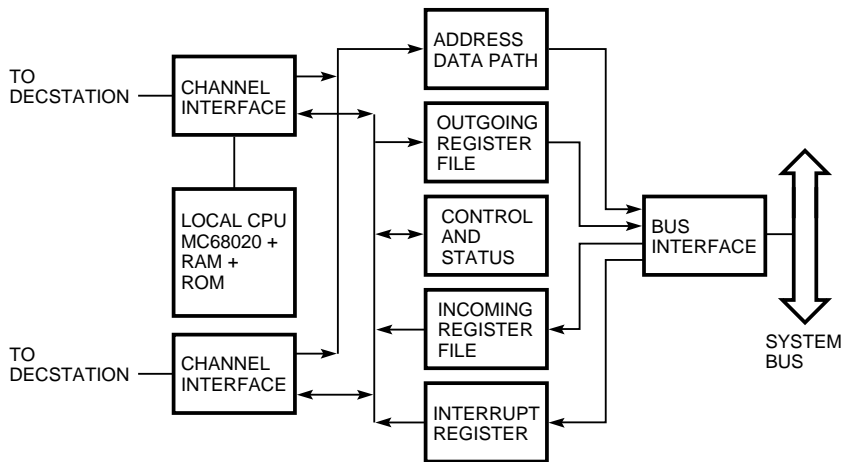
The original specification for the ADU I/O system required support only for serial line, small computer systems interface (SCSI) disk, and Ethernet I/O devices. We knew that the ADU would be used to exercise new CPU chips and untested software. With this in mind, we organized the I/O system around a DECstation 5000 workstation as a front-end and console processor. This reduced our work considerably, as all I/O is done by the workstation. A TURBOchannel module connects the DECstation 5000 over a 50MB/s cable to the I/O module in the ADU. We selected 50MB/s in order to support the simultaneous, peak-bandwidth operation of two SCSI disk strings, an Ethernet, and a fiber distributed data interface (FDDI) network adapter. The I/O module contains two of these channels, which allows two DECstation 5000 workstations to be attached.

At the hardware level, the I/O system supports block transfers of data from the main memory of the workstation to and from ADU memory. In addition, the I/O module includes command and doorbell registers, which are used by ADU processors to attract the attention of the I/O system.

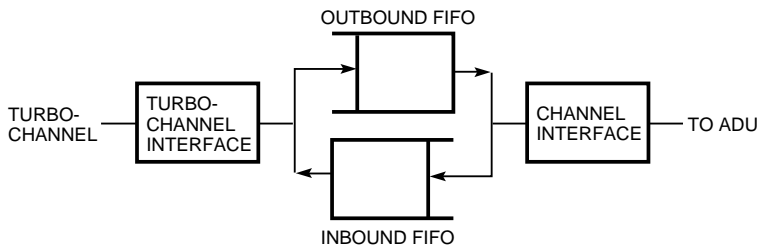
In software, I/O requests are placed by ADU processors into command rings in ADU memory. The memory address of a command ring is placed into an I/O control register, and the associated doorbell is rung. The doorbell causes a hardware interrupt on the front-end DECstation 5000, which alerts the I/O server process that action is needed. The I/O server reads the command ring from ADU memory and performs the requested I/O. I/O completion status is stored into ADU memory, and an interrupt is sent to the requesting ADU processor.

In addition to its role as an I/O front-end processor, the DECstation 5000 workstation acts as a diagnostic and console processor. When an ADU is powered on, diagnostic software is run from the workstation. First, the correct functioning of the I/O module is tested. Then the ADU module identification process determines the types and locations of all CPU and storage modules in the system. Diagnostics are then run for each module.

Once diagnostic software has run, the console software is given control. This software is responsible for loading privileged architecture library (PAL) and operating system software. Once the operating system is running, the workstation becomes an I/O server.



(a) ADU I/O Module



(b) TURBOchannel I/O Module

Figure 10 I/O Module

The presence of the DECstation 5000 gave the chip team and operating system developers a stable place to stand while checking out their own components. In addition, the complete diagnostic capability and error checking coverage of the ADU hardware helped to isolate faults.

The central features of the I/O module, shown in Figure 10, are two 1K- by 80-bit register files built from 5-ns ECL RAMs. These memories are cycled every 10 ns to simulate dual-ported memories at the 20-ns bus cycle rate. One memory is used as a staging RAM for block transfers from the I/O processors to ADU memory. The other memory is shared between use as command register space for the I/O system and a staging RAM for transfers from ADU memory to the I/O system.

On the bus side, the register files are connected directly to the backplane bus transceivers. On the I/O side, the register files are connected to a shared

40-ns bus that connects to the two I/O channels.

The buses are time-slotted to eliminate the need for arbitration logic. As a consequence, the I/O module control logic is contained in a small number of programmable array logic chips that implement the I/O channel controllers and a block-transfer state machine that handles bus transfers.

Each I/O channel carries 32 bits of data plus 7 bits of ECC in parallel on a 50-pair cable. Each data word also carries a 3-bit tag that specifies the destination of the data. The cable is half-duplex, with the direction of data flow under the control of software on the DECstation. Data arriving from the DECstation is buffered in 1K FIFOs. These FIFOs carry data across the clock-domain boundary between the I/O system and the ADU and permit both I/O channels to run at full speed simultaneously.

Each I/O channel interface also has an address counter and a slot-mapping RAM, which are loaded

from the workstation. The slot-mapping function sets the correspondence between ADU bus addresses and the geographically addressed storage and CPU modules. The address and slot map for each channel are connected to a common address bus. This bus bypasses the register files and directly drives the backplane transceivers during bus address cycles.

The far end of the I/O cable connects to a single-width TURBOchannel module in the DECstation 5000. This module contains ECC generation and checking logic, and FIFO queues for buffering data between the cable and the TURBOchannel. The FIFO queues also carry data across the clock-domain boundary between the I/O channel and the TURBOchannel modules.

The I/O module has a local CPU subsystem containing a 12-MHz Motorola 68302 processor, 128KB of erasable programmable read-only memory (EPROM), and 128KB of RAM. The CPU subsystem also includes an Ethernet interface, two serial ports, an SCSI interface, an Integrated Services Digital Network (ISDN) interface, and audio input and output ports. When in use, the local CPU subsystem uses one of the I/O channels otherwise available for the connection of a DECstation 5000. Although the local CPU on the I/O module is capable of running the full ADU I/O system, in practice we use it for supplying interval timer and real-time clock service for the ADU.

The I/O module was somewhat overdesigned for its original purpose of supplying disk, network, and console I/O service for ADU processors. This capability was put to use in mid-1991 when the demand for ADUs became so intense that we considered building additional systems. Instead, by using the excess I/O resources, the slot-mapping features of the hardware, and the capabilities of PAL-code, we were able to use a three-processor ADU as three independent virtual computers. Independent copies of the console program shared the I/O hardware through software locking and were allocated one CPU and one storage module each. Multiprocessor ADUs now routinely run both OpenVMS AXP and DEC OSF/1 AXP operating systems at the same time.

### *Packaging*

Simplicity was the primary goal in the design of the ADU package. Our short schedule demanded that we avoid innovation and use standard parts wherever possible.

The ADU's modules and card cage are standard 9U (280 millimeter by 367 millimeter) Eurocard components, which are available from a number of ven-

dors. The cabinet is a standard Digital unit, usually used to hold disks. Power supplies are off-the-shelf units. Three supplies are required to provide the 4,000 watts consumed by a system containing a full complement of modules. A standard power conditioner provides line filtering and distributes primary AC to the power supplies. This unit allows the system to operate on 110-volt AC in the United States, or 220-volt AC in Europe.

Cooling was the most difficult part of the packaging effort. The use of ECL throughout the system meant that we had to provide an airflow of at least 2.5 m/s over the modules. After studying several alternatives, we selected a reverse impeller blower used on Digital's VAX 6000 series machines. Two of these blowers provide the required airflow, while generating much less acoustic noise than conventional fans.

Since blower failure would result in a catastrophic meltdown, airflow and temperature sensors are provided. A small module containing a microcontroller monitors these parameters as well as all power supply voltages. In the event of failure, the autonomous controller can shut down the power supplies. This module also generates the system clock.

## Conclusions

Sometimes it is better to have twenty million instructions by Friday than twenty million instructions per second.  
— Wesley Clark

One hundred CPU and storage modules and 35 I/O modules have been built, packaged as 35 ADU systems, and delivered to software development groups throughout Digital. Not just laboratory curiosities, these systems have become part of the mainstream AXP development environment. They are in regular use by compiler development groups, operating system developers, and applications groups.

The ADU also provided a full-speed, in-system exerciser for the chips. By using the ADU, the chip developers were able to detect several subtle problems in early chip implementations.

The ADU project was quite successful. ADU systems were in the hands of developers approximately ten months before the first product prototypes. The systems exceeded our initial expectations for reliability, and provided a rugged, stable platform for software development and chip test. The project demonstrated that a small team, with focused objectives, can produce systems of substantial complexity in a short time.

## Acknowledgments

John Dillon designed the power control subsystem and the package. Steve Morris wrote the ADU console software. Andrew Payne contributed to ADU diagnostics. Tom Levergood assisted with the physical design of the I/O modules. Herb Yeary, Scott Kreider, and Steve Lloyd did module debugging and testing at Hudson and at SRC. Ted Equi handled project logistics at Hudson, and Dick Parle was responsible for material acquisition and supervision of outside vendors at SRC.

## References

1. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal*, vol. 4, no. 4 (1992, this issue).
2. C. Thacker, L. Stewart, and E. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers*, vol. 37, no. 8 (August 1988): 909-920.
3. R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon, "Implementing a Cache Consistency Protocol," in *Proceedings of the 12th International Symposium on Computer Architecture* (IEEE, 1985).
4. J. Archibald and L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4 (November 1986): 273-298.
5. *1991 GaAs IC Data Book and Designer's Guide* (GigaBit Logic, Newbury Park, CA, 1991): 2-39.

## Trademarks

The following are trademarks of Digital Equipment Corporation:

Alpha AXP, AXP, DEC OSF/1 AXP, DECstation 5000, Digital, OpenVMS AXP, TURBOchannel, and VAX 6000.

Motorola is a registered trademark of Motorola, Inc.

## Biographies

**Charles P. Thacker** Chuck Thacker has been with Digital's Systems Research Center since 1983. Before joining Digital, he was a senior research fellow at the Xerox Palo Alto Research Center, which he joined in 1970. His research interests include computer architecture, computer networking, and computer-aided design. He holds several patents in the area of computer organization and is coinventor of the Ethernet local area network. In 1984, Chuck was the recipient (with B. Lampson and R. Taylor) of the ACM Software System Award. He received an A.B. in physics from the University of California and is a member of ACM and IEEE.

**David G. Conroy** Dave Conroy received a B.A.Sc. degree in electrical engineering from the University of Waterloo, Canada, in 1977. After working briefly in industrial automation, Dave moved to the United States in 1980. He cofounded the Mark Williams Company and built a successful copy of the UNIX operating system. In 1983 he joined Digital to work on the DECtalk speech synthesis system and related products. In 1987 he became a member of Digital's Semiconductor Engineering Group, where and has been involved with system-level aspects of RISC microprocessors.

**Lawrence C. Stewart** Larry Stewart received an S.B. in electrical engineering from MIT in 1976, followed by M.S. (1977) and Ph.D. (1981) degrees from Stanford University, both in electrical engineering. His Ph.D. thesis work was on data compression of speech waveforms using trellis coding. Upon graduation, he joined the Computer Science Lab at the Xerox Palo Alto Research Center. In 1984 he joined Digital's Systems Research Center to work on the Firefly multiprocessor workstation. In 1989 he moved to Digital's Cambridge Research Lab, where he is currently involved with projects relating to multimedia and AXP products.