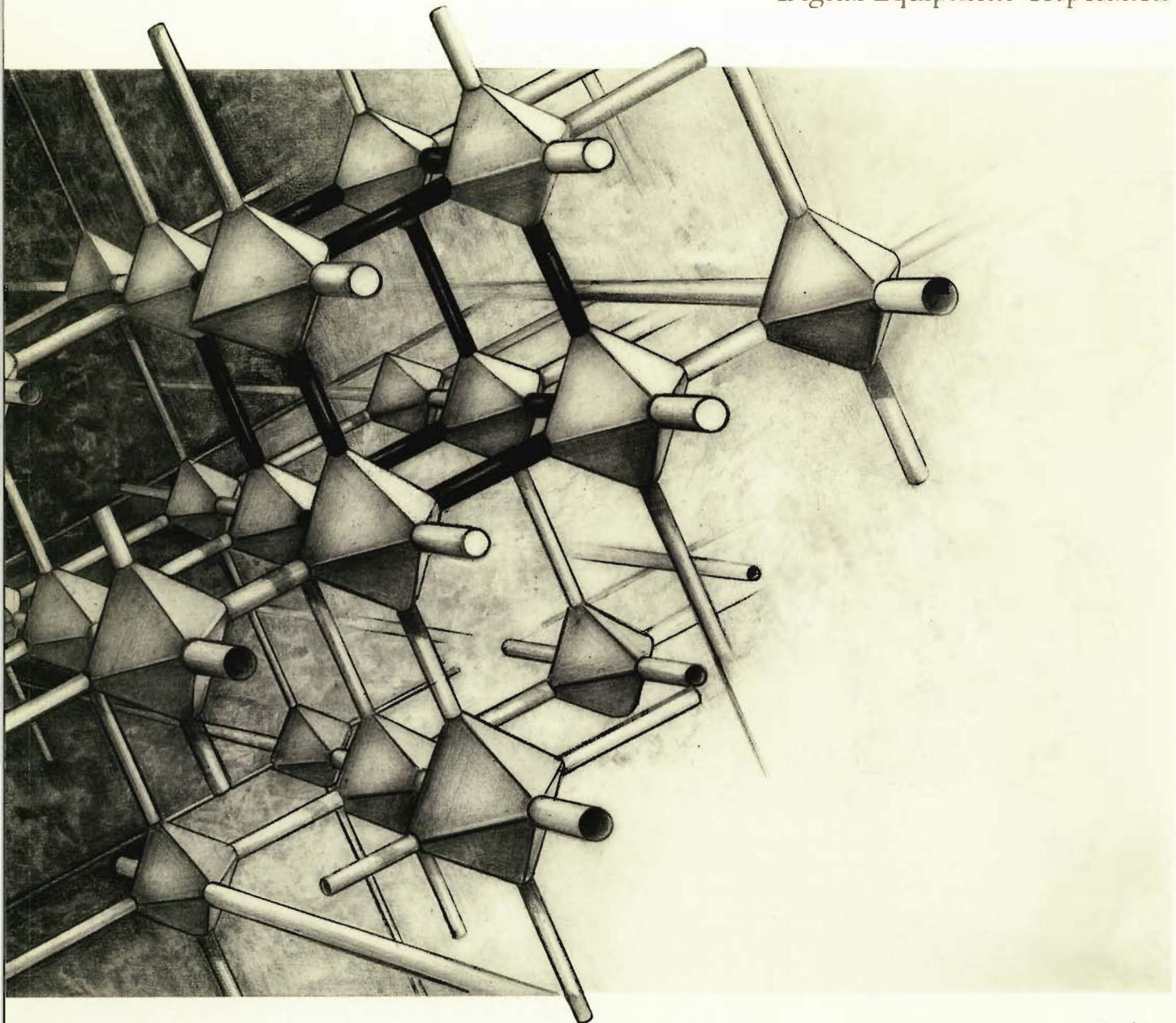


CVAX-based Systems

Digital Technical Journal

Digital Equipment Corporation



Number 7
August 1988

Managing Editor

Richard W. Beane

Editor

Jane C. Blake

Production Staff

Production Editor — Helen L. Patterson

Designer — Charlotte Bell

Typographers — Jonathan M. Bohy

Margaret Burdine

Illustrator — Deborah Keeley

Advisory Board

Samuel H. Fuller, Chairman

Robert M. Glorioso

John W. McCredie

Mahendra R. Patel

F. Grant Saviers

William D. Strecker

Victor A. Vyssotsky

The *Digital Technical Journal* is published by Digital Equipment Corporation, 77 Reed Road, Hudson, Massachusetts 01749.

Changes of address should be sent to Digital Equipment Corporation, attention: List Maintenance, 10 Forbes Road, Northboro, MA 01532. Please include the address label with changes marked.

Comments on the content of any paper are welcomed. Write to the editor at Mail Stop HLO2-3/K11 at the published-by address. Comments can also be sent on the ENET to RDVAX::BLAKE or on the ARPANET to BLAKE%RDVAX.DEC@DEC.WRI.

Copyright © 1988 Digital Equipment Corporation. Copying without fee is permitted provided that such copies are made for use in educational institutions by faculty members and are not distributed for commercial advantage. Abstracting with credit of Digital Equipment Corporation's authorship is permitted. Requests for other copies for a fee may be made to Digital Press of Digital Equipment Corporation. All rights reserved.

The information in this journal is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

ISSN 0898-901X

Documentation Number EY-6742E-DP

The following are trademarks of Digital Equipment Corporation: ALL-IN-1, DEQNA, HSC70, J-11, MicroVAX, MicroVAX II, MicroVAX 3000, NMI, NOTES, Q-bus, Q22-bus, RA81, RA82, RD54, RQDX3, RX50, TK50, ULTRIX-32, VAX, VAX-11/780, VAX-11/782, VAX 6200, VAX 6210, VAX 6220, VAX 6230, VAX 6240, VAX 8200, VAX 8300, VAX 8650, VAX 8800, VAX 8840, VAXB1, VAXELN, VAX MACRO, VAX SPM, VAX/VMS, VMS.

Compu-Share is a trademark of Compu-Share, Inc.

GDS II is a trademark of Calma Corporation.

SPICE is a trademark of the University of California at Berkeley.

Tektronix and DAS are trademarks of Tektronix, Inc.

Book production was done by Digital's Educational Services Media Communications Group in Bedford, MA.

Cover Design

Systems based on Digital's advanced CMOS technology are featured in this issue. The graphic on our cover includes the lattice structure of the silicon crystal, a basic element of this technology. The expansion of the image expresses the performance growth and the system extensibility of the new CVAX-based systems.

The cover was designed by Barbara Grzeslo and Jacquie Hockaday of the Graphic Design Department.

Contents

- 8 **Foreword**
Robert M. Supnik

CVAX-based Systems

- 10 ***An Overview of the VAX 6200 Family of Systems***
Brian R. Allison
- 19 ***The Architectural Definition Process of the VAX 6200 Family***
Brian R. Allison
- 28 ***Interfacing a VAX Microprocessor to a High-speed Multiprocessing Bus***
Richard B. Gillett, Jr.
- 47 ***The Role of Computer-aided Engineering in the Design of the VAX 6200 System***
Jean H. Basmaji, Glenn P. Garvey, Masood Heydari, and Arthur L. Singer
- 57 ***VMS Symmetric Multiprocessing***
Rodney N. Gamache and Kathleen D. Morse
- 64 ***Performance Evaluation of the VAX 6200 Systems***
Bhagyam Moses and Karen T. DeGregory
- 79 ***Overview of the MicroVAX 3500/3600 Processor Module***
Gary P. Lidington
- 87 ***Design of the MicroVAX 3500/3600 Second-level Cache***
Charles J. DeVane
- 95 ***The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor***
Thomas F. Fox, Paul E. Gronowski, Anil K. Jain, Burton M. Leary, and Daniel G. Miner
- 109 ***Development of the CVAX Floating Point Chip***
Edward J. McLellan, Gilbert M. Wolrich, and Robert AJ Yodlowski
- 121 ***The System Support Chip, a Multifunction Chip for CVAX Systems***
Jeff Winston
- 129 ***Development of the CVAX Q22-bus Interface Chip***
Barry A. Maskas
- 139 ***The CVAX CMCTL — A CMOS Memory Controller Chip***
David K. Morgan

Editor's Introduction



Jane C. Blake
Editor

The second issue of the *Digital Technical Journal* (March 1986) featured papers on the then recently announced MicroVAX II system, a system based on a single-chip VAX implementation. In this seventh issue, we present papers on the second generation of that chip set, CVAX, the two new systems that take advantage of its increased performance capabilities, and a new version of the VAX/VMS operating system for symmetric multiprocessing.

The new mid-range system based on the CVAX chip set is the VAX 6200 family of computers, which utilizes a multiprocessing architecture. The first of two papers by Brian Allison is an overview of this highly configurable, expandable system. Brian's second paper offers insights into the architectural definition process for the 6200.

One of the major decisions made by the 6200 engineers was to design a new interconnect to support the multiprocessor system. Rick Gillett presents an informative discussion of the complexities involved in interfacing a microprocessor to a high-speed, multiprocessing bus.

To ensure the availability of first-pass functional parts, a design verification team of engineers worked in parallel with the 6200 module designers. Jean Basmaji, Glenn Garvey, Masood Heydari, and Art Singer discuss the computer-aided engineering and verification principles the team instituted for the project.

Rod Gamache and Kathy Morse then describe the major features of symmetric multiprocessing in the VAX/VMS operating system. Of particular interest is their description of a new synchronization method implemented in VAX/VMS version 5.0.

In the last paper related to the VAX 6200 system, Bhagyam Moses and Karen DeGregory describe the development of workloads to measure VAX 6240 performance. As part of their discussion, they include performance measurements and analysis.

The second new system based on the CVAX chip set is the low-end MicroVAX 3500/3600 system, which offers three times the performance of its predecessor, the MicroVAX II. In his overview of the major sections of the processor module, Gary Lidington relates how schedule and performance requirements influenced product design decisions.

Charles DeVane then describes the MicroVAX 3500/3600 system's two-level cache architecture, with emphasis on the design of the second-level cache. He also presents some cache performance test results.

The high performance of both the VAX 6200 family and the MicroVAX 3500/3600 system is attributable in great measure to the CMOS VAX family of chips on which these systems are based. Our five final papers address the design and development of this chip set. Frank Fox, Paul Gronowski, Anil Jain, Mike Leary, and Dan Miner begin the discussion with an explanation of how designers achieved the performance goals for the single-chip VAX CPU by reducing ticks per instruction and machine cycle time.

A companion to the CVAX CPU, the floating point processor chip offers floating point performance equal to that of the microprocessor for integer operations. The approach taken to attain this goal and a description of the chip are presented by Ed McLellan, Gil Wolrich, and Bob Yodlowski.

Jeff Winston then discusses the development of the system support chip, which provides a common core of peripheral system functions.

Next, Barry Maskas relates the design efforts of three groups, one in Japan and two in the U.S., that resulted in a single-chip interface between the CVAX microprocessor and the Q22-bus I/O subsystem.

In our final paper, Dave Morgan describes the CVAX memory controller chip, CMCTL, which is optimized for Q-bus-based systems.

Jane Blake

Biographies



Brian R. Allison Brian Allison, a consultant engineer for mid-range VAX systems, was the system architect responsible for the coordination of the VAX 6200 system definition and design. Prior to this work, he served as system architect for a project that yielded several products, including DEBNA, DEBNK, and the KA800. As a member of the VAX-11/750 design team, he wrote various portions of the microcode for that product. Brian holds a B.S.E.E. and a B.S.C.S. from Worcester Polytechnic Institute (1977).



Jean H. Basmaji Jean Basmaji is the technical director of computer-aided engineering and design-verification testing for the VAX 6200 project. A software consultant engineer, he has also been involved with CAE/DVT planning and scheduling, and has served as CAE/DVT project leader for the VAX 6200 CPU module. Jean joined Digital after receiving his B.S.E.E. from Lowell Technological Institute in 1977.



Karen T. DeGregory A senior software engineer in the Systems Performance Analysis Group, Karen DeGregory is project leader of systems performance measurement for the VAX 8840 and VAX 6240 systems. In addition to planning and implementing the measurements, she helped develop appropriate workloads for these systems. Prior to this work, Karen was a senior software specialist in the Software Services Backup Support Group. She received her B.S. (1980) with honors and distinction and her M.S. (1981) from Cornell University.



Charles J. DeVane Charles DeVane is a senior hardware engineer in the MicroVAX Systems Development Group. For the MicroVAX 3500/3600 project, he designed the second-level cache on the KA650 CPU module and guided the process of module system debug and introduction to manufacturing. Before joining Digital in 1981, Charles received a B.S.E.E. from North Carolina State University in Raleigh, North Carolina. He is a member of Eta Kappa Nu and Tau Beta Pi engineering honor societies.



Thomas F. Fox Frank Fox, a principal engineer in the Semiconductor Engineering Group, worked on the implementation of the CVAX 78034 CPU chip. He is currently designing a high-performance microprocessor and consulting with the Advanced Semiconductor Development Group on the development of a submicron CMOS process. Frank was educated in Ireland and received a B.E. degree from University College Cork (1974) and a Ph.D. degree from Trinity College Dublin (1978), both in electrical engineering. He has published papers on ultrasonic instrumentation and magnetic resonant imaging (MRI) and has three patents pending.



Rodney N. Gamache A consulting software engineer in the VAX/VMS Software Development Group, Rod Gamache has been with Digital for 11 years. Shortly after receiving a B.S. in mathematics and computer science from the University of New Hampshire, he joined Digital to work on the development of DECnet Phases III and IV for both RSX-11M and VMS. For the last two years, Rod has been project leader for the VMS symmetrical multiprocessing project and has filed two patents on VMS SMP. Rod also serves as a consultant for the architectures of future low-end VAX processors.



Glenn P. Garvey Glenn Garvey is an engineering supervisor presently leading a team in the verification of a new VAX processor. He was the project leader for the system-level verification performed on the VAX 6200 system and has been involved in modeling and verification since coming to Digital in 1982. Glenn was a co-op student at Digital in 1980 and 1981. He holds a B.S.E.E. from Rensselaer Polytechnic Institute.



Richard B. Gillett, Jr. Rick Gillett, a consultant engineer, led the VAX 6200 CPU module project. Prior to his work on the 6200, he served as one of the architects of the XMI bus and was a member of the VAXBI bus project team. Relative to his work on the VAX 6200 design, he has recently filed 13 patent applications. Currently, he is system architect for a new VAX system. Rick joined Digital after receiving a B.S.E.E. (summa cum laude) from the University of New Hampshire in 1979. He is a member of Tau Beta Pi and Phi Kappa Phi.



Paul E. Gronowski Before receiving a B.S.E.E. from the University of Cincinnati in 1984, Paul Gronowski was a co-op student at Digital working on chips for the VAX 8200 and 8300 systems. Currently a senior engineer in the Semiconductor Engineering Group, he has been a codesigner of the E-box for the CVAX 78034 CPU chip, designer of the bus interface unit and exponent section for a CMOS floating point chip, and is now doing advanced development work for a new CMOS microprocessor. Paul is a member of Eta Kappa Nu.



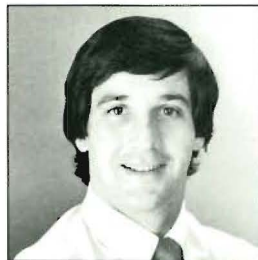
Masood Heydari Masood Heydari is an engineering manager responsible for the computer-aided design, verification, and testing of mid-range VAX systems. He is also managing the development of an I/O subsystem for VAX products. Since joining Digital in 1981, he has developed the diagnostics for a 36-bit system and has been responsible for functional test pattern generation for several products. Most recently, he was responsible for CAD/DVT on the VAX 6200 project. Masood holds a B.S. and an M.S. in computer engineering from Boston University (1980). He is a member of Tau Beta Pi.



Anil K. Jain Anil Jain received an M.S.E.E. from the University of Cincinnati (1980) and a B.S.E.E. from Punjab Engineering College (1978). Upon joining Digital in 1980, he worked on bipolar and CMOS-1 technology while a member of the Device Modeling Group. As a senior engineer working on the CVAX project, Anil designed the bus interface unit for the CPU chip and has a patent pending for the bus interface protocol. He is currently working as a project leader on the floating point chip project for vector applications.



Burton M. Leary Mike Leary is a principal engineer in the Semiconductor Engineering Group/Advanced Development Memory Group and is currently working on the design of advanced memory products. In previous work, he participated in the design of the floating point chips for the MicroVAX and 8200/8300 systems and the design of a CMOS serial interface controller chip. Mike joined Digital after receiving a B.S.E.E. from the University of Massachusetts. He is a member of Tau Beta Phi.



Gary P. Lidington Currently an engineering manager in the MicroVAX Engineering Group, Gary Lidington has served as a system engineer for the MicroVAX 3500/3600 system and as a maintainability engineering project manager for the MicroVAX II and single-board computers with Q-bus multiprocessing architectures. Before coming to Digital in 1981, Gary was a co-op student and test engineer on the L200 project at Teradyne, Inc. He holds a B.S.E.E. (honors) from Tufts University and an M.S. in computer engineering from the University of Massachusetts.



Barry A. Maskas Barry Maskas, a consulting engineer with the Semiconductor Engineering Group, is the project leader for the development of a custom VLSI memory controller. Prior to his current work, he was the U.S. project leader and architect for the CVAX Q22-bus interface chip and co-designer of the MicroVAX IICPU and memory boards. Barry came to Digital in 1979 after receiving a B.S.E.E. from Pennsylvania State University.



Edward J. McLellan Ed McLellan is a principal engineer in the Semiconductor Engineering Group. He has worked on the design of several chips, including the J-11 and the CVAX floating point accelerator chips. He holds one patent for previous work and has made application for two additional patents based on work done for the CVAX floating point chip. Ed joined Digital in 1980 after receiving a B.S. in computer and systems engineering from Rensselaer Polytechnic Institute. Currently he is project leader for a floating point chip for a new processor.



Daniel G. Miner Dan Miner came to Digital after receiving a B.S. in computer engineering from Rensselaer Polytechnic Institute in 1985. A software engineer in the Semiconductor Engineering Group, he wrote the debug and diagnostic tests for the CVAX CPU chip. Dan co-authored and presented a paper on the subject of testability strategy at the 1987 IEEE International Test Conference.



David K. Morgan Dave Morgan is the engineering manager of Advanced Peripheral Development in the Semiconductor Engineering Group. For his work on several engineering projects, Dave has three patents pending. Before joining Digital in 1975, he was a design engineer at the RCA Solid State Division and holds five patents for his work on integrated circuit designs. Dave received a B.S.E.E. (1969) from Western New England College, an M.S.E.E. (1972) from Rutgers University, and has pursued doctoral work in solid-state physics.



Kathleen D. Morse As a consulting software engineer, Kathy Morse is working on advanced development for the VAX/VMS Development Group. She is one of the VMS SMP architects and consults on enhancements to various parts of the VMS executive. Earlier, she implemented the VMS support for the first MicroVAX systems, the first asymmetric multiprocessing VAX system, and the MA780 multiport memory. Kathy joined Digital in 1976 after receiving her B.S.C.S. degree from Worcester Polytechnic Institute, where she also earned her M.S.C.S. degree in 1985. Kathy is a member of IEEE, the Professional Council, and ACM as well as Tau Beta Pi and Upsilon Phi Epsilon.



Bhagyam Moses Bhagyam Moses is the engineering manager of the Mid-range Systems Performance Analysis Group, a group which she established two and a half years ago. Prior to her current work with the VAX 8000 and VAX 6200 series of systems, she had been involved in the modeling and performance measurement of the VAX 8600 and 8650 systems, PDP-11 systems, DECSYSTEM-20, and earlier VAX systems. Bhagyam joined Digital in 1979. She received a B.S. degree (honors) in mathematics from Spicer Memorial College and an M.S. in applied mathematics from Howard University.



Arthur L. Singer Art Singer supervises the computer-aided design, simulation, and timing verification for a new I/O adapter product. His previous work includes supervision of the system simulation and timing verification of the VAX 6200 system and microdiagnostic and release support for the VAX 8600 project. Before joining Digital in 1984, Art was employed by IPL Systems as a design engineer and manager of a microdiagnostic development group. While at IPL, he received a patent for the design of an instruction unit. He is a member of Tau Beta Pi and Eta Kappa Nu.



Jeff Winston A member of the Semiconductor Engineering Group, Jeff Winston designed the microsequencer for the VAX 8200 CPU chip and led the design of two generations of the MicroVAX System Support Chip. He has also contributed to the development of many CAD tools used in chip design. Before joining Digital in 1980, Jeff received his B.S. (1979) and M.S. (1980) in Electrical Engineering from Cornell University. He is currently leading the development of the XMI interface chip set for a new mid-range VAXCPU.



Gilbert M. Wolrich A consulting engineer in the Semiconductor Engineering Group/Architecturally Focused Logic, Gil participated in the J-11 control chip design and was project leader for the CVAX CFPA and the J-11 FPA chip projects. He holds a patent for an ALU with Carry Length Detection used on the J-11 FPA. Gil received a B.S.E.E from Rensselaer Polytechnic Institute in 1971 and an M.S.E.E. from Northeastern University in 1977.



Robert AJ Yodlowski Bob Yodlowski, a principal engineer, is the chip implementation project leader and lead circuit designer for the CVAX floating point accelerator. For his work on this project, Bob has three patents pending. He was also senior circuit designer on the J-11 floating point chip project. Relative to this project work, he is a co-inventor and patent holder for ALU with Carry Length Detection (1987). Before joining Digital in 1977, Bob was a senior member of the technical staff at LFE Corporation in Waltham, MA. He received a B.S. in engineering physics from Cornell University (1968) and an M.S.E.E. from Syracuse University (1970).

Foreword



Robert M. Supnik
*Corporate Consultant,
VLSI Technology, and
Group Manager,
Semiconductor Engineering
Microprocessor Development*

In May 1985, Digital introduced the MicroVAX II computer system. Based on the MicroVAX processor chip set, the MicroVAX II system offered unsurpassed price, performance, and reliability characteristics. In the three years since then, Digital has sold more than 100,000 systems based on the MicroVAX chip set. There are more MicroVAX-based systems in the field than all other types of VAX systems combined.

In the same three years, the practice of computer engineering has advanced considerably. Faster processors, bigger memories, quieter packages, and more complex software have appeared in a steady stream. For Digital to remain competitive, we would need, over time, a second generation of VLSI-based VAX chips and systems. The chips and systems that constitute the second VLSI-based generation are described in this issue of the Digital Technical Journal.

The planning for the second generation began in 1983. That year, the LSI Group (now Semiconductor Operations) formulated a multiyear program for the development of both semiconductor process technology and leading-edge chip products. The key characteristics of this process/product plan were

- CMOS (complementary metal-oxide-semiconductor) process technology (Previous Digital chips were based on NMOS technology.)

- Multiple process generations related by optical scaling
- VAX microprocessors as the leading edge chip development projects
- Performance improvements targeted for greater than 50 percent per year

This program not only provided the LSI Group with an overall structure for its process and chip development projects; it also provided Digital's system groups with a stable, long-term basis for planning system products.

The program was also a significant leap of faith. When it was formulated, there was no MicroVAX business. The MicroVAX II system was two years away from shipment. Almost all design resources in the LSI Group and in the low end system groups were busy with the MicroVAX chip set and its related systems. Major development projects in technology, chip design, systems design, and manufacturing were required to bring the program vision to fruition.

Work began with development of the underlying semiconductor technology. Starting in 1983, a team from Semiconductor Manufacturing's Advanced Semiconductor Development (ASD) defined, simulated, and tested CMOS-1, Digital's first CMOS process. When first defined, CMOS-1's key features — N-well base on a p-type epitaxial layer, two levels of metal interconnect, 2.0 micron feature size, direct scalability to 1.5 micron feature sizes — were controversial within an industry that was still debating NMOS versus CMOS. Over time, these choices have been vindicated, and CMOS-1 has proven to be a mainstream, robust, highly manufacturable process.

Equally important was development of design methods for larger and more complex chips. The Semiconductor Engineering Computer Aided Design (CAD) Group continuously refined the structured design process first deployed for MicroVAX and V-11. The goals of this effort were improved simulation coverage, faster turnaround time, and more extensive automated verification. One consequence of the increased use of CAD tools was a dramatic increase in the amount of computing power required. This generation of chip development projects used four

times as much computing power as the first VLSI generation.

The Semiconductor Engineering Microprocessor Group began architectural prework on the second-generation chip set (called CVAX) in mid-1984. The overarching goal was simple: three times the performance of the MicroVAX chip set in less than three years — a compound performance growth rate of more than 50 percent per year. The central processor design started from the MicroVAX base but drew upon ideas from other VAX implementations, notably the 8700. The floating point unit design focused on minimal execution flows for the most common instructions. Both chips transitioned to implementation in 1985.

The original concept for the CVAX chip set had been to build chip-for-chip analogues of MicroVAX — a central processor and a floating point unit. However, as the flexibility of the new CMOS process, and the efficiency of the CAD tools, were appreciated by designers, the chip set concept expanded beyond the central processor to include key peripherals. The implementation of these peripheral functions in VLSI chips made systems faster, more reliable, and less expensive. In addition, it allowed peripheral functions to be standardized across multiple system implementations and additional functions to be added in modular fashion. The Semiconductor Engineering peripherals group (now Advanced Development) specified and implemented a memory controller, a memory driver, a console interface, and a Q-bus interface.

After the MicroVAX II system shipped in May 1985, the Low-end Systems Group and the Mid-range Systems Group became actively involved in the specification of the CVAX chips and in the definition of new systems utilizing the chip set. In the low end, the 3500/3600 systems were defined as evolutionary extensions of the MicroVAX II. Nonetheless, the performance targets for the new chips posed knotty design problems for a system family bounded by both cost and packaging considerations.

In the mid-range, the system designers wished to exploit the CVAX chip set's combination of high performance and low cost by constructing

an extensible multiprocessor system. They defined a new system interconnect (supported by unique chips) to provide unprecedented flexibility and extensibility in configuring systems, and new system packaging to support the concept. However, a general-purpose multiprocessor system was feasible only if the VMS operating system could take advantage of the incremental power offered by additional processors. This required a major restructuring of VMS to support symmetric (all processors equal) multiprocessing. Thus, the definition and implementation of the mid-range 6200 system family and of VMS symmetric multiprocessing support had to be closely linked.

As the engineering development projects progressed, manufacturing became heavily involved in planning and executing the transition from design to volume product. LSI Manufacturing in Hudson, Massachusetts, introduced CMOS-1 into multiple fabrication units in order to produce prototypes quickly and to ramp up to high volume production. System manufacturing groups in Westfield (Massachusetts), Albuquerque (New Mexico), Puerto Rico, and other sites worked closely with the system designers to introduce the new manufacturing processes required for system production.

The results of these development programs is a family of VAX systems with exemplary price, performance, and reliability characteristics. Moreover, the programs leave as residuals a set of VLSI components from which other products can be built, and base technology from which further advances in chip and system design will evolve. The initial program vision has been fulfilled, even exceeded. Many people, in teams and individually, worked together to bring this about. The excellence of the results reflects, in full measure, the excellence of the work that they have done.

An Overview of the VAX 6200 Family of Systems

Digital's VAX 6200 series is a high-performance, expandable family of computer systems that combines low-cost microprocessors with high-performance memory and I/O subsystems. Based on the CMOS VAX chip set, the VAX 6200 CPU module performs at 2.8 times the VAX-11/780 system; utilizing a multiprocessing architecture, system speeds are available up to 11 times the VAX-11/780 system. The memory subsystem utilizes a multi-controller architecture for up to 256MB of total system memory. The XMI bus, the electrical interconnect for the system, supports the multiple processors, memory subsystems, and VAXBI channel adapters. The VAXBI is used for all I/O devices.

The VAX 6200 family of computer systems is the most recent addition to Digital's line of VAX computer systems. The VAX 6200 systems, primarily based on CMOS technology, are mid-range systems which exploit multiprocessing techniques. The VAX 6200 family currently comprises four systems, all built from common subassemblies. Any VAX 6200 system may be upgraded to any other VAX 6200 system simply by adding CPU and memory modules to the existing cabinet. This paper provides an overview of the system and therefore a context for the five papers that follow in this issue. These papers describe several of the components in detail, the engineering design effort, the performance evaluation process, and some of the multiprocessing aspects of the operating system.

In the past, CMOS-based microprocessor technology has been used primarily to build low-cost systems. Today, by using multiples of these low-cost microprocessors, we are presented a unique opportunity to produce a high-performance computer system when the microprocessors are coupled with high-performance memory and I/O subsystems. Although this type of system architecture will not directly result in faster execution of a single task, it does result in greater system throughput in applications that have several simultaneously computable tasks. The architecture couples the effectiveness of the VMS operating system in multiprogrammed environments

with hardware optimized for efficient multiprocessor operation. The result is a system that offers similar performance for a large class of applications at a better price-performance ratio than that offered by traditional single-processor, high-performance computer systems.

A primary objective of the VAX 6200 system design is to provide a highly configurable and expandable computing environment. To achieve this objective, designers chose a modular subassembly design for the total system. This modular design provides for cost-effective basic systems and also allows for system expansion to achieve higher performance. All members of the VAX 6200 family are housed in the same cabinet and use the same basic subassemblies. The only difference is the number of processors, amount of memory, and number of I/O devices. Table 1 details the configurations of the VAX 6210, VAX 6220, VAX 6230, and VAX 6240 systems.

System Architecture

All VAX 6200 systems consist of CPU(s), memory, and I/O channel adapters connected to a common system interconnect known as the XMI. The VAXBI is used as the interconnect to all I/O devices in the system.¹ All memory and I/O devices are equally accessible by all CPUs in the system. Figure 1 shows a block-level diagram of the VAX 6200 system.

Table 1 VAX 6200 Family System Configurations

	VAX 6210	VAX 6220	VAX 6230	VAX 6240
Number of processors	1	2	3	4
Main memory	32MB	64MB	64MB	128MB
VAXBI channels	2	2	2	2
CPU cycle time	80 ns	80 ns	80 ns	80 ns
Cache size (per CPU)	1KB 256KB	1KB 256KB	1KB 256KB	1KB 256KB
Free XMI slots	10	8	7	4
Performance (times one VAX-11/780 system)	2.8	5.5	8.3	11.0
Maximum CPUs	4	4	4	4
Maximum memory	256MB	256MB	256MB	256MB
Maximum VAXBI channels	6	6	6	6

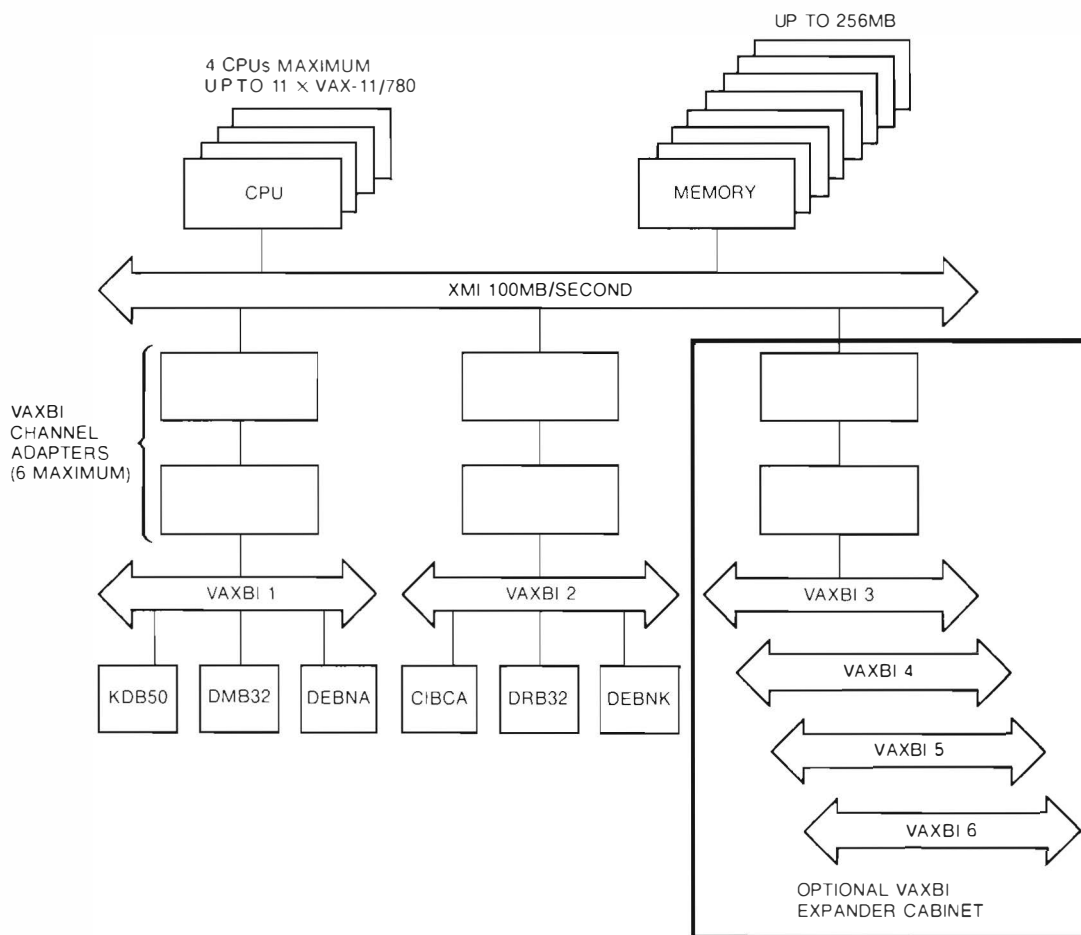


Figure 1 VAX 6200 System Block Diagram

The primary goal of the VAX 6200 system is to allow higher levels of system performance through multiprocessing. To simplify software design and to be consistent with previous multiprocessor architecture, it was essential to provide a shared memory resource. All system memory is a global resource accessible through the same address space from each processor and from all I/O devices. A sophisticated multilevel cache contained locally in each CPU minimizes memory accesses on the XMI. Cache coherency is maintained totally by hardware.

Technology

The VAX 6200 systems are based on a number of different CMOS technologies. The VAX CPU chip set and the system interconnect transceivers are implemented entirely in Digital's full custom CMOS process featuring a size of 1.5 microns.²

The interface between each module and the system interconnect is implemented in channelless 1.5-micron CMOS gate arrays. The number of gates used in these arrays varies from 18K to 50K gates. The interface to the VAXBI and the XMI arbitration system is implemented in 1.5-micron channeled arrays. The on-board CPU caches are implemented with 45-nanosecond (ns) 64K-by-4 CMOS static random-access memories (SRAMs) and industry-standard CMOS cache tag chips.

All VAX 6200 XMI and VAXBI modules are connected to their respective backplanes by a 300-pin zero insertion force (ZIF) connector. All modules use 10-layer controlled impedance printed circuit boards. All cables from the modules are connected through the backplane to improve reliability and to minimize the task of replacing modules.

The VAX 6200 XMI backplane is a 14-layer controlled impedance printed circuit board. Side 1 consists entirely of surface-mount contacts for the ZIF connector. Side 2 consists of plated through holes for power strips and I/O pins, and surface-mount pads for resistors. These surface-mount resistors form the termination network for the XMI signal lines.

VAX 6200 XMI modules use a printed circuit board very similar to the VAXBI printed circuit board. XMI modules have the same finger pin design as the VAXBI, but the module size is 28 cm (11.025 inches) deep instead of 20.38 cm (8.025 inches) deep.

The VAX 6200 modules make use of advanced module technology features to maximize both

the number of I/Os available to VLSI chips and the amount of logic that can be put on a module. Surface-mounted components are used extensively throughout the system. Further, many passive components and a limited number of active surface-mounted components reside on side 2 of the modules. All VAX 6200 modules limit the use of surface mount to 50-mil lead pitch components with vias on 100-mil centers. Across the modules in the system, there is a mixture of small outline integrated circuit (SOIC), plastic leaded chip carrier (PLCC), and cerquad surface-mount packages.

All VAX 6200 XMI modules interface to the XMI through a set of eight semicustom parts. These eight chips are physically mounted on a section of the module known as the "XMI corner." This section of the module is approximately 12.7 cm (5 inches) by 3 cm (1.2 inches) and is located by the A, B, and C connectors of the module. (See Figure 2.) The XMI interface area is identical on all modules so that a common electrical load is presented to all slots on the XMI. The XMI corner has four 44-pin cerquad packages on side 1 of the module and four 44-pin cerquad packages on side 2. In addition, approximately 100 surface-mounted-device (SMD) signal termination resistors and bulk power capacitors are divided evenly across both sides of the module in the XMI corner.

Figure 2 is a photograph of the three VAX 6200 XMI modules. Note that all three modules have the identical components in the lower right corner and a similar gate array directly above the XMI corner.

VAX 6200 CPU Module

As noted earlier, the VAX 6200 CPU (KA62A) is based on the CMOS VAX chip known as the CVAX. The KA62A is a single module that implements a full CPU subsystem. Included on the KA62A module are

- The CVAX chip, which includes a 1 kilobyte (KB) on-chip cache
- An external 256KB cache
- A floating point accelerator chip (CFPA)
- Console support hardware
- An interface to the XMI

Figure 3 shows a block diagram of the KA62A module.

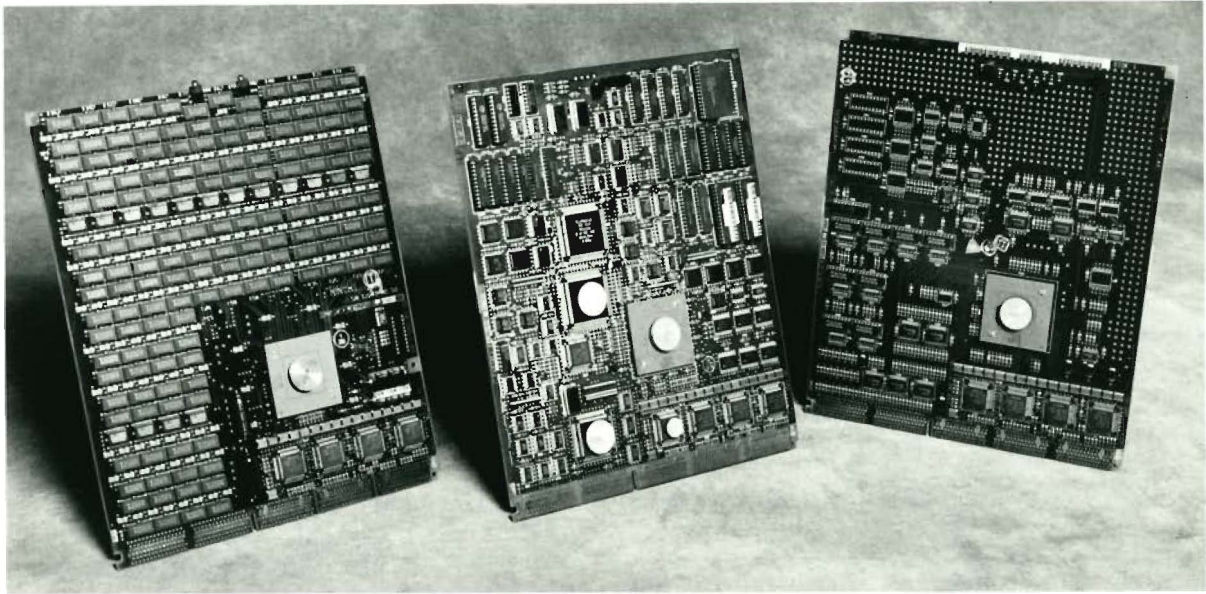


Figure 2 Three VAX 6200 XMI Modules

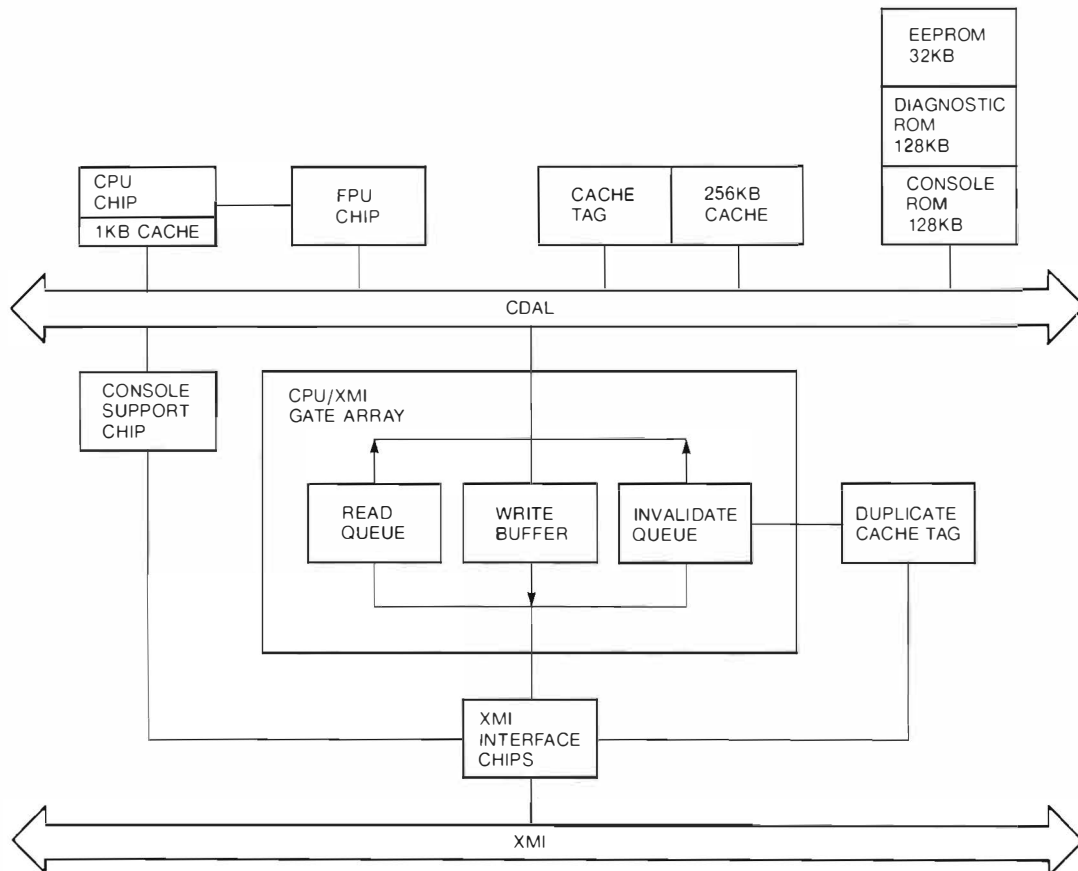


Figure 3 VAX 6200 CPU Module (KA62A) Block Diagram

Using the CVAX processor with an 80-ns cycle time, the KA62A CPU module performance is approximately 2.8 times that of the VAX-11/780 system. For a total system performance up to 11 times greater than the VAX-11/780, up to four KA62A CPU modules may be configured in a VAX 6200 system.

The KA62A CPU module contains a two-level cache to reduce memory access time. The primary cache is 1KB in size and resides inside the CVAX chip. This cache contains only instruction data to eliminate the need to invalidate this data as other processors write to cached data locations. (The VAX architecture provides strict rules for modification of instruction type data.) The secondary cache is 256KB in size and contains data as well as instructions. The KA62A monitors write transactions on the system interconnect and invalidates any cached locations written by another CPU or I/O device.

Memory

The VAX 6200 memory subsystem is made up of memory controller/array modules and is known as the MS62A. The MS62A module, shown in Figure 4, contains a memory controller chip and 32 megabytes (MB) of 1-megabit (Mb) dynamic RAMs (DRAMs). The MS62A maintains a 64-bit data path between the memory controller chip and the RAMs, and implements an 8-bit error-correcting code (ECC) for each 64-bit word. The MS62A contains hardware to implement up to 16 "lockable" memory locations per memory array. These memory locks are used extensively by processors and I/O devices to ensure singular access to data structures in a shared-memory multiprocessor system.

The greater memory bandwidth required by multiple processors and I/O channels is achieved by memory interleaving. The MS62A allows interleaving on 32-byte boundaries. As long as memory addresses are randomly distributed across the lower 6 address bits, the bandwidth of the total memory subsystem can be increased linearly with the addition of interleaved memory controllers.

The MS62A memory modules may be interleaved two, four, or eight ways. The interleave factor is automatically determined by the system upon power-up or system initialization. However, designers have given the user the ability to manually specify the interleave characteristics of the memory subsystem. Up to eight MS62A memory modules may be configured in a VAX 6200 system.

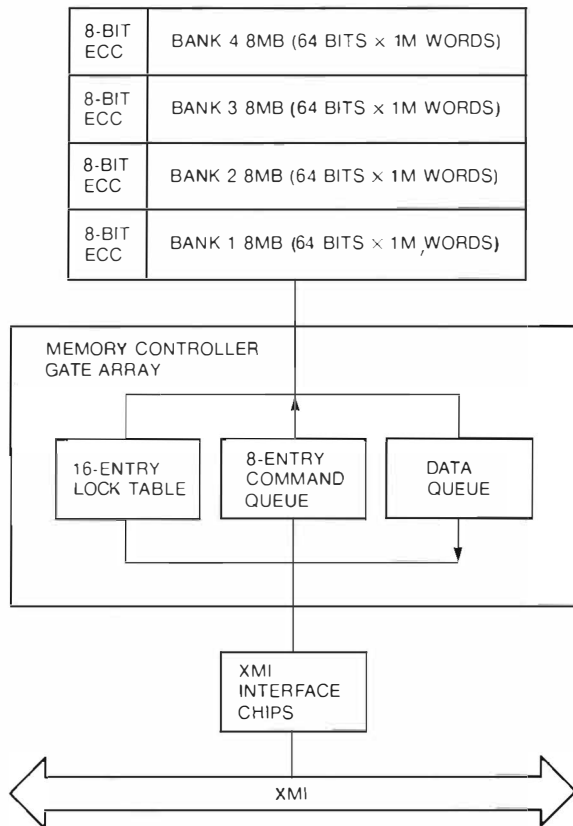


Figure 4 VAX 6200 Memory Module (MS62A) Block Diagram

I/O Channels

The VAX 6200 system uses the VAXBI bus as the interconnect for all I/O devices. The system interface to the VAXBI is a two-module set called the DWMBA. Figure 5 shows a block diagram of the DWMBA modules. The DWMBA/A module is connected to the XMI, and the DWMBA/B module is connected to the VAXBI. These two modules are interconnected with a 120-wire cable assembly which may be up to 4.6 meters (15 feet) long.

The DWMBA allows VAXBI devices to read system memory at up to 5.5MB per second and to write system memory at up to 13.3MB per second. Any VAXBI-compatible device may be connected to the VAX 6200 systems through the DWMBA. All VAX 6200 systems contain a minimum of two VAXBI channels and may optionally contain up to six VAXBI channels.

System Interconnect, the XMI

The XMI, the primary electrical interconnect in the VAX 6200 family of computer systems, encompasses

- The protocol observed by a node on the XMI
- The electrical environment of the XMI
- The backplane
- The logic used to implement the protocol

The XMI can support multiple processors, multiple memory subsystems, and multiple I/O channel adapters.

XMI nodes may be classified as commanders or responders, depending on their role in a given transaction. A commander is a node that is initiating an XMI transaction. A responder is the node that must act upon the transaction. A processor node usually acts as a commander. (However, a processor node may become a responder if another node reads a control/status register on the CPU.) Memory nodes, on the other hand, are always responders since they cannot initiate an XMI transaction. I/O nodes may act as either commanders or responders, depending on the type of I/O operation. The functions of these nodes are further explained in sections below.

Because the XMI is a pended interconnect, several transactions can be in progress simultaneously. When an XMI commander initiates a request for a read or to solicit an interrupt vector, an identifier code is also transmitted to the selected responder. At this point, control of the XMI is relinquished, and other transactions are allowed to take place while the responder fetches the requested read data or interrupt vector. The responder then arbitrates for control of the XMI and returns the requested data or vector along with the identifier code. By monitoring the identifier codes, the initial commander is able to receive the correct data and continue.

Arbitration and data transfers occur simultaneously over a multiplexed set of address and data lines, and a separate set of arbitration lines. The XMI supports quadword, octaword, and hexword reads to memory, as well as quadword and octaword memory writes. In addition, the XMI supports longword-length read and write operations to I/O space. These longword operations implement byte and word modes required by certain I/O devices.

The XMI has 30 address bits, and the smallest addressable entity is a single byte. XMI address space is divided into two halves by bit 29 of the address. When bit 29 equals zero, an address is said to fall into memory space. When bit 29 equals one, the address is said to fall within I/O space. This arrangement matches the maximum physical address as defined by the VAX architecture and allows up to 512MB of physical memory to be addressed. The XMI architecturally allows up to 16 nodes, but is physically and electrically constrained to 14 nodes.

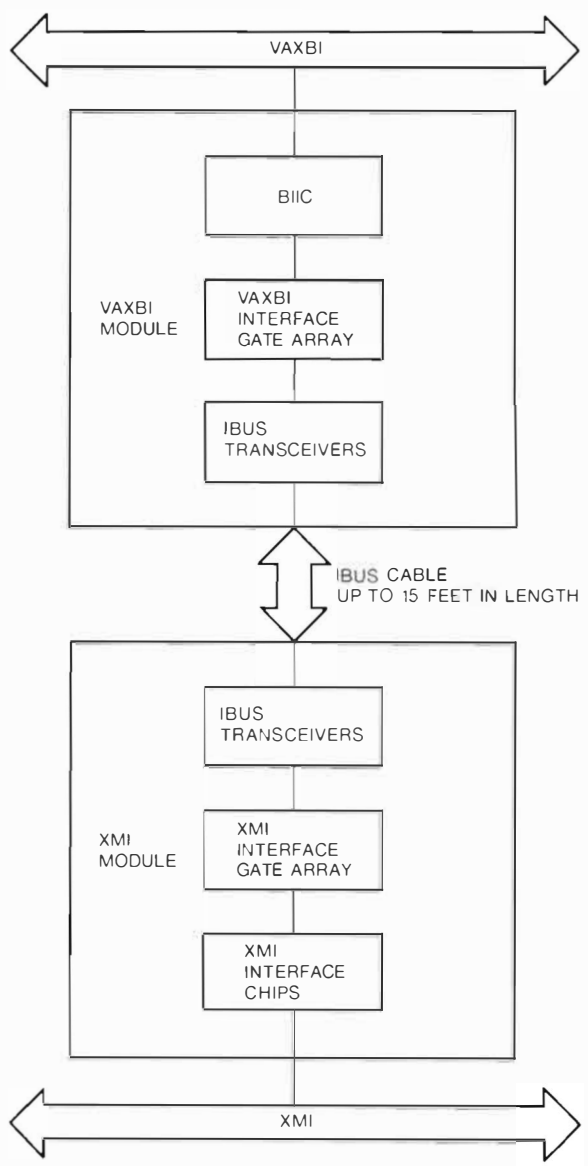


Figure 5 VAX 6200 VAXBI Channel Adapter Block Diagram

The XMI multiplexes data and address information onto the 64-bit data path. Data transactions are initiated with a "command and address" cycle, followed by multiple data cycles. The maximum length for an XMI transaction is 32 bytes of data. The XMI cycle time is 64 ns. The effective bandwidth of the XMI is a function of the data transfer size, as shown in Table 2.

The XMI architecture allows for three distinct classes of devices.

Processor Nodes

Each processor node contains a CPU that executes instructions and manipulates data contained in XMI memory. The processor node can execute any instruction set compatible with the VAX-style byte addressing and memory locking mechanisms. A processor node will have a cache that must force all written data back to main memory. Any cached processor module must also monitor write traffic on the XMI and invalidate any location in its own cache that is written into main memory. Processor nodes must also be capable of responding to interrupt requests generated either by other processors or by I/O nodes.

I/O Nodes

I/O nodes generally respond to I/O space references either by mapping the data onto another bus or by interpreting data as a command. An I/O node can also become a commander on the XMI and access global XMI memory. I/O nodes may generate interrupt sequences directed toward processor nodes. However, I/O nodes do not respond to commands directed toward memory space.

Memory Nodes

Memory nodes act only as responders on the XMI. They respond to read and write requests directed toward memory address space. These requests are generated either by processor or I/O nodes.

Data Integrity

The XMI contains a number of features to enhance the integrity and reliability of the interconnect. First, all XMI information transfer lines are parity protected, and XMI command confirmation signals are ECC protected. The XMI protocol is sufficiently robust to permit detection and recovery of all single-bit error conditions on these signals. Additionally, the XMI defines time

Table 2 XMI Bandwidth Based on Transaction Size

Transaction Size in Bytes	Interconnect Bandwidth in MB/second
4	31.25
8	62.50
16	83.33
32	100.00

out conditions that may be used to detect and diagnose failures.

VAX Console

The VAX 6200 system implements the standard VAX console functionality by means of software that conditionally executes on each of the KA62A CPU modules. Each KA62A CPU module contains a serial-line interface, 256MB of read-only memory (ROM), 32MB of electronically erasable ROM (EEROM), and 512 bytes of RAM. Control is passed to the console software upon any one of the following occurrences:

- System power-up
- Initialization
- Receipt of a control-P character from the console terminal
- Execution of the HALT instruction
- Some severe error conditions

Each KA62A CPU has access to console terminal transmit-and-receive lines carried on the system backplane. Upon power-up, control of the system console terminal is dynamically allocated to one of the CPUs present in the system. This CPU, known as the "boot" processor, provides the system interface to the console terminal as well as to the switches and lights located on the system control panel.

On receiving commands from the console terminal, the boot processor may run diagnostics or boot an operating system. This processor communicates with other processors by means of a structure maintained in memory known as the console communications area (CCA).

Also considered as part of the console subsystem, a TK50 tape drive is included in each VAX 6200 system. The tape drive is connected to the system by means of a TBK50 controller module located on a VAXBI I/O channel and is used for the following purposes:

- Saving all volatile parameters for the console subsystem
- Loading the VAX Diagnostic Supervisor (VDS) when no disk is available or functional in the system
- Distributing operating system and layered software

The TK50 tape drive is also available under operating system control as a general-purpose data interchange device.

Built-in Self-test

Extensive built-in self-test is used by all modules contained within the VAX 6200 systems. Upon power-up, all modules within the system, with the exception of the DWMBA, perform a self-test in parallel. After self-test is complete, the CPU modules examine each other's status; the one in the lowest slot number that passed self-test is selected as the boot processor. The boot processor then continues to execute an additional test to ensure memory accessibility and finally executes a test of the DWMBA.

Physical Packaging

All VAX 6200 systems are housed in the same cabinet, which is 78 cm (30.5 inches) wide by 154 cm (60.5 inches) tall by 76 cm (30 inches) deep. The cabinet contains one 14-slot XMI backplane, two 6-slot VAXBI backplanes, and all necessary power and cooling to sustain a wide range of configurations. Figure 6 shows a VAX 6240 with the front door removed.

The XMI is physically implemented in a 14-slot backplane assembly containing ZIF module connectors, signal terminating networks, and a centralized clock and arbitration system. Modules are located on 2 cm (0.8 inch) centers. The XMI backplane is supplied with +5 volts (V) for general logic, a separate +5 V supply for memory, ± 12 V for the console terminal line drivers, and -5.2 V/ -2 V for emitter-coupled logic (ECL). Presently none of the VAX 6200 XMI modules utilizes the ECL voltages, but ECL is included for potential future use.

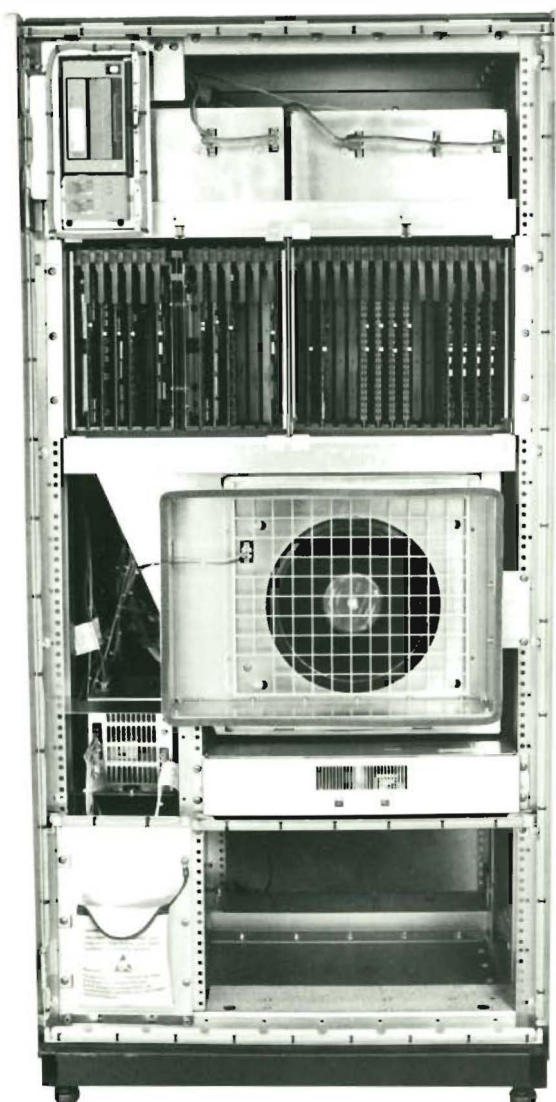


Figure 6 VAX 6240 System, Front Door Removed

The VAX 6200 systems all contain two 6-slot VAXBI backplanes, which are configured as independent channels. The first slot of each VAXBI backplane is occupied by the DWMBA/B module, leaving 5 slots for standard VAXBI interfaces. All systems contain a DEBNK TK50 tape controller and a DEBNA Ethernet controller as standard equipment. The two VAXBI backplanes are supplied with +5 V, ± 12 V, -5.2 V, and -2 V.

Summary

The VAX 6200 family of systems merges the CMOS VLSI VAX chip, which is used in a number of Digital's products, with a very high perfor-

mance memory and I/O subsystem. This hardware, combined with the new fully symmetric multiprocessing capabilities of VMS version 5.0, allows very high system throughput previously achievable only with ECL technology. Moreover, the extensive use of CMOS technology results in physically smaller systems. These smaller systems consume less power and are more reliable due to the lower component count and lower power consumption.

References

1. P. Wade, "The VAXBI Bus — A Randomly Configurable Design," *Digital Technical Journal* (February 1987): 81–87.
2. T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital Technical Journal* (August 1988, this issue): 95–108.

The Architectural Definition Process of the VAX 6200 Family

The architectural definition of Digital's VAX 6200 family was governed by a twofold goal: to build a system with higher throughput than previous CMOS, Q-bus-based systems at a cost lower than ECL-based systems. Decisions made during the definition process were influenced by firm schedule guidelines. Further, the very nature of the multiple processor system imposed its own requirements, particularly in the definition of the XMI bus. This new 64-bit-wide interconnect is specifically designed to meet the memory and I/O needs of the symmetric multiprocessor system. Throughout the architectural definition process, engineers continually evaluated the interdependency of one design decision upon another and against the project and schedule goals. By this process, the total definition of the system — the XMI bus, the processor module, memory module, console subsystem, and packaging — was achieved.

Definition of the VAX 6200 family of systems began in March 1985. The engineers' intent was to design a follow-on product to the VAX 8200/8300 family of systems, still in development at that time. This paper discusses the system architectural definition process that took place during 1985.

Like the VAX 8200/8300 family before it, the VAX 6200 family provides a system environment for a VLSI VAX chip set. This new family of systems is a mid-range VAX implementation. In this context, a mid-range system is defined as a product with more capability than the Q-bus-based systems and less capability than the emitter-coupled logic (ECL) based systems.

Project Goals

The primary goal of the VAX 6200 program was twofold: to build a system with greater system throughput than the CMOS, Q-bus-based VAX systems, and to ensure system cost was lower than that of high-performance ECL-based systems. Designers would achieve this goal by designing a system architecture that allows a moderate number of low cost CMOS VAX microprocessors to share a common system environment. Such an efficient multiprocessor system environment would offer higher throughput for a large number of applications and at a cost lower than a high-performance single processor.

Once the decision to build a multiprocessor was made, the next question was how many processors to include. Several small computer manufacturers were building 8- to 32-processor systems at the time. Our belief was that the market for systems with numerous processors was fairly small because few applications would run efficiently on these systems. Therefore, we decided to design the VAX 6200 as a 4-processor system, with the possibility of expansion to 8 processors. This arrangement would allow us to still configure cost-effective 1- to 2-processor systems. If we found a significant number of applications could benefit from the larger number of processors, we could expand to 8 processors.

Building an efficient multiprocessor system would necessitate optimization of both hardware and software functionality. The VMS asymmetric multiprocessing code (VMS versions 2 through 4) that supported the VAX-11/782, VAX 8300, and VAX 8800 systems worked well for compute-bound, dual-processor systems. However, asymmetric operating system software would not be acceptable for larger scale multiprocessors. In the existing VMS asymmetric multiprocessing design, most operating system code was executed on the processor designated as the "primary" processor. Whenever a process needed to perform I/O or invoke most of the VMS system services, the process would have to be scheduled

on the primary processor. The task of making VMS more symmetric in its handling of I/O and VMS system services was undertaken to support the VAX 8840 and the VAX 6200 families.¹

Discussion of how we chose to optimize the VAX 6200 hardware begins in the section *The System Interconnect*.

Schedule

In March 1985 the design of the CVAX chips was already well under way. These chips would be delivered in time to allow systems to ship in late 1987. Based on the CVAX chip set schedule, we established the following schedule for the development of the VAX 6200 system:

Six months of architectural definition

Twelve months of design/simulation

Three months to build and test approximately five first-pass prototypes

Six months to build approximately 70 second-pass prototypes

Three months for final testing and manufacturing introduction

This two and a half year schedule significantly influenced the definition of the system architecture as well as the selection of implementation technologies. (Actual implementation took three years. The design/simulation phase took three months longer than expected, and the first-pass prototype phase took three months longer than expected.)

The System Interconnect

The first order of business was to define a new system interconnect. This interconnect would have the bandwidth required to support the memory and I/O needs of the multiple processors. We outlined three requirements that would affect the design of the new system interconnect.

- We estimated that each CVAX processor would require between 3 megabytes (MB) and 6MB per second of data to/from memory. This rate would depend on the clock rate of the processor, the selected cache architecture, and the cache "hit" rate of the program being executed.
- We also estimated that each processor could require peaks of 1MB to 1.5MB per second of I/O bandwidth.

- To maintain predictable memory access time, we decided that the system bus should not be run over 75 percent utilized.

Using the worst-case anticipated bandwidth needs, 80MB per second of peak bus bandwidth would be required to support 8 processors.

Because of the tight schedule and our awareness of the significant amount of time needed to design a new system bus, we first looked into the feasibility of using an existing bus. We considered but rejected the existing VAXBI bus, the primary interconnect for the VAX 8200/8300 system, because of its limited 13.3MB per second bandwidth. We also rejected the NMI bus, the VAX 8500/8700/8800 family interconnect, because this bus uses ECL technology. At one point we even considered using the SBI from the VAX-11/780 system with a 64-bit data path instead of its existing 32-bit data path. After extensive analysis, however, we decided a new system bus would have to be engineered for the product to meet its goals.

Although we would have to define a new bus for processor-to-memory communications, the schedule did not allow us to design a full complement of I/O interfaces for the new bus. Since a large number of I/O interfaces would be available on the VAXBI, the design team decided to use the VAXBI as the interconnect to all I/O devices. The new system interconnect, the XMI, would be used only to connect processors, memories, and VAXBI channel adapters. Therefore, in addition to the requirements listed above, the XMI architecture would also allow multiple VAXBI channel adapters to optimize I/O throughput where necessary for large systems. Use of the VAXBI for I/O adapters also had the positive effect of minimizing the number of electrical interconnects to the XMI; the physical length of the XMI would consequently be shorter and the total capacitance lower. Further discussion of the channel adapters is presented in the section *VAXBI Channel Adapters*.

In June 1985 a team of 11 senior-level engineers was assembled to produce the architectural and electrical specification for the XMI bus and the VAX 6200 system. In addition to architectural and electrical experts, this team included one representative from each of the anticipated module design teams. Almost all members had previously worked on projects involving the VAXBI bus. It was understood that the XMI would be

used solely for the VAX 6200 family of systems, unlike the VAXBI, which would be used across many different applications. A strict adherence to this premise greatly helped the specification team to put technical trade-offs in perspective.

XMI Electrical Interface Definition

Since most of the VAX 6200 system is CMOS and transistor-transistor logic (TTL) based, we immediately decided the XMI could not be implemented in ECL. To maintain a TTL-level bus and to achieve the desired bandwidth, the data path clearly would have to be 64 bits wide. Further, to meet our goal of 80MB per second bandwidth, the XMI would have to transfer 64 bits of information every 80 nanoseconds (ns). (This transfer rate assumes a protocol in which address and data are multiplexed, and up to 32 bytes of data can be transferred per address cycle.)

Several electrical alternatives were considered for the XMI. A scheme using the commercially available FutureBus components was seriously considered. However, we rejected this scheme because a large number of components would be necessary to implement the 64-bit data path.

The lack of commercially available components to drive a 64-bit bus at the required speed finally led us to a decision. We would design a bit-sliced custom CMOS bus interface chip set. Each chip would transceive 11 lines, and seven chips would be used for the entire data path. Although the "sliced" bus interface would use more module real estate than a larger chip, the sliced bus design greatly simplified the chip packaging problems. Each chip would fit into a standard 44-pin cerquad package. A sliced XMI interface also allows each chip to dissipate under 0.5 watt (W), which enhances reliability and relieves the need for heat sinks on the part. Without heat sinks, the XMI interface parts can be mounted on both sides of each module. This arrangement saves 50 percent of the real estate necessary to interface to the XMI.

To simplify the design of the full custom XMI interface parts, we would keep the functional requirements for the parts as simple as possible. The XMI interface chips have little knowledge of the XMI protocol and serve only as the electrical interface. Due to the divergent needs of processor, memory, and I/O interfaces, designers already knew that each module would need a different VLSI chip for XMI interface functions. We decided, therefore, that each module VLSI

chip would be required to supply the logic to implement the bus-level protocol.

As the electrical design of the XMI progressed, a bus cycle as fast as 64 ns appeared feasible. Although not entirely necessary to support the stated system performance goals, the faster XMI cycle time was strongly pursued to gain extra margin in the system design. Furthermore, this fast cycle time would allow the possibility of system upgrades to faster processors in the future. Consequently, 64 ns became the stated goal for the XMI cycle time; 80 ns was the fall back strategy if the design complexity of a 64-ns cycle time began to place the overall project schedule at risk.

Logic design across the entire system was done assuming a 64-ns cycle time. Eventually 64 ns became the actual speed of the bus as the CMOS process was characterized and the first parts were sampled and found to contain sufficient margin to support the faster cycle time.

XMI Protocol Definition

XMI protocol definition took place in parallel with the electrical definition of the bus. It was clear from the start that the bus would cycle several times faster than the memory subsystem. This difference in cycle times immediately led us to the decision that the XMI would run a "pended" bus protocol. A pended bus protocol allows control of the XMI to be relinquished between a "read" command and the return of the data from the memory subsystem. With multiple processors and multiple memory controllers, several read commands could be outstanding at a time.

To optimize data traffic on the XMI bus, we needed to define data transfer commands of several lengths. Since VAX instructions may write as little as 1 byte of data, a 64-bit write command was defined. (There is a mask field associated with the write command that allows single bytes to be written.) Since the VAXBI bus already had commands to transfer 16 bytes of data per address, it was essential to allow similar commands on the XMI bus to minimize the interface complexities to the VAXBI. Eventually we added a 32-byte read command to allow processors to prefetch larger amounts of data upon cache misses. A 32-byte write command was not implemented, because it would be too great a burden for the memory controller to buffer multiple 32-byte write commands.

In many cases the protocol of the XMI is similar to that of the VAXBI. In part this similarity resulted because the designers of the XMI were very familiar with the VAXBI. The similarity between protocols was also deliberately chosen because it greatly reduced the complexity of interfacing the XMI to the VAXBI for I/O purposes.

The bus arbitration scheme is one area where designers had to deviate from the method used by the VAXBI bus. The VAXBI uses the main bus data path for arbitration, which requires extra bus cycles. This approach was not feasible for the XMI pending protocol, since two arbitrations are necessary for each read transaction. Further, the VAXBI arbitration scheme also requires a great deal of duplicated logic in every module. Due to the large number of allowable XMI nodes, it was not feasible to implement an arbitration mechanism located on an XMI module. To implement arbitration on any XMI module would have required a great number of signal pins. The solution was to implement a centralized arbiter. The XMI uses a module physically attached to the rear of the backplane as a centralized arbiter as well as the source of the master clock.

The subject of data integrity on the XMI was of great concern to the designers. Initially carrying error-checking and correction (ECC) bits on the bus was considered. However, this scheme was rejected because additional encode/decode timing would have been required, and because additional bits would have to be carried on the bus. Eventually a robust protocol was implemented based on parity detection and hardware/software retries when errors are detected. All transient single-bit errors on the XMI are recoverable.

XMI Physical Definition

The physical definition of the XMI was a difficult task. There were a great number of interdependent trade-offs for module size, module spacing, number of backplane slots, and cabinet size.

To minimize design complexity, we had decided at a very early stage that each module within the VAX 6200 system would implement a single function. Thus the task of designing each module was simplified and the diagnosability of the system enhanced. Initially, the size of the XMI module was largely governed by the space needs of the processor. Analysis showed that a processor based on the CVAX chip set could fit on

a module the same size as the existing VAXBI module 20.32 cm (8.0 inches) by 23.33 cm (9.187 inches). In addition, 32MB of memory could fit on the same size module.

System packaging was another factor to consider in selecting the module size. From the very start of the VAX 6200 program, it was not clear what type of system-level packaging was optimal. Designers knew, however, that the larger systems would primarily be placed in computer-room environments. For these applications, a standard 153.67-cm (60.5-inch) tall cabinet would be necessary. What was not clear was if office-type packaging or rack-mount-type packaging would be required. Since VAXBI formfactor pedestal and rack-mount box packages were both available, designers found it very attractive to use the same formfactor module for the XMI to ease the development of these packages if necessary. Based on the functionality fit and the desire to potentially reuse existing packaging, we decided to adopt the VAXBI module size for the XMI.

Another advantage to using the VAXBI module size was the opportunity to use the VAXBI zero insertion force (ZIF) backplane connector. Historically, developing new backplane connector technologies has proven difficult and time-consuming. The VAXBI uses a five-segment, 60-pin-per-segment connector. Of the 300 pins, 120 pins are assigned to the VAXBI signals and 180 pins to each module for I/O use. Since the XMI has 32 more data-path bits than the VAXBI, designers chose to allot an extra 60 pins for the XMI signals. This leaves 120 pins for general module use. Designers believed the arrangement to be acceptable, since there are no I/O modules for the XMI. The only use for the I/O pins is to connect to the VAXBI card cages. The 120 available pins are more than adequate for this function.

To meet the cycle time goals for the XMI bus, the length of the XMI would have to be limited to about 0.3 meters (12 inches) and the number of loads limited to approximately 16. The XMI protocol assumed a maximum of 16 devices would interface to the XMI bus. Eventually the number of slots in an XMI backplane became 14 for two different reasons. First, 14 slots would allow a system to have 8 processors, four memory arrays, and two VAXBI channels. Second, a 14-slot XMI backplane would be very similar in size to the pair of 6-slot VAXBIs that already existed in the VAXBI pedestal and rack-mount box packages.

XMI module spacing of 2.03 cm (0.8 inches) is the same as that on the VAXBI bus. We chose this spacing to allow for heat-sink components on side 1 of the module. Enough height would remain to allow non-heat-sink, surface-mounted components on side 2.

About 18 months into the program, the module designs were complete, and both the processor module and memory module were experiencing great difficulty during printed circuit board layout. Although all components could be placed within the area available, the very high pin-count gate arrays in use (223 pins) were causing considerable routing problems. To lower the schedule risk to the program, designers decided to lengthen the module by 7.62 cm (3 inches). The impact to the computer-room packaging was minimal because a 76-cm (30-inch) cabinet depth could accommodate the change. However, the change in module length made impossible the adaptation of the existing VAXBI pedestal and rack-mount packaging to the XMI. At this time the pedestal-based strategy for the MicroVAX 3500/3600 systems was clear, thus reducing the need to package the VAX 6200 family of systems for office use. Further, extremely low sales of rack-mounted VAX 8200/8300 systems led us to the decision that a rack-mount package was not immediately necessary.

XMI Interface Technology

The decision to implement the XMI electrical interface in simple full custom CMOS parts dictated that each module have additional logic to complete the XMI interface and to supply module-specific logic. To simplify both the design of the XMI interface parts and the CAD tools, we decided that all module-to-XMI interfaces would be implemented in the same technology. Given the aggressive design schedule, we would need a technology that was mature as well as easy to design for.

We initially focused on a family of 2-micron CMOS gate arrays available from LSI Logic and Toshiba. However, it quickly became clear that array limitation of approximately 10,000 gates would force us to place multiple chips on each module. The use of multiple chips was highly undesirable from the perspective of design resources, module real estate, and cost. A search was started to locate a suitable alternative. To get the desired logic density, several semicustom

alternatives were explored but ultimately rejected because of the immaturity of their CAD tools.

Discussions with LSI Logic Corporation led us to consider their newly developed 1.5-micron "Sea of Gates" array, which offers up to 50,000 routable gates. Although this array did not give us the mature technology we were seeking, it did appear to offer the flexibility needed by all XMI designs. We ultimately chose the LSI Logic LL10000 family of gate arrays because all designs could use the same technology. Moreover, we could focus our CAD tool development on a single technology.

The 64-bit-wide XMI data path forced the pin count of a single interface chip to be 200-plus pins. The LSI Logic LL10000 array was offered in a 223-pin pin-grid-array (PGA) package which appeared suitable. Although most of the logic on each module was implemented in surface-mounted components, we did not pursue a 223-pin, surface-mount package. We wanted to avoid the manufacturing problems presented by components with 25-mil pitch leads.

The Processor Module

The VAX 6200 processor module uses the CVAX chip set to implement the VAX instruction set. Due to an uncertainty about the final CVAX chip speed, the CPU module was designed to operate over a range of 70 ns to 100 ns. The intent was to use "binned" parts in the VAX 6200 system, and to use the "nominal" parts in the MicroVAX 3500/3600 systems. (Chip manufacturing processes yield parts of different speeds; "binning" refers to the process of testing the chips over a range of speeds.) For the CVAX chip set, the nominal parts run at 90 ns, and the binned parts run at 80 ns.

A major system-wide architectural issue, which primarily affected the processor module, was whether the cache should be write-back or write-through. Although a write-back cache could potentially reduce the number of processor writes on the XMI by 50 percent, such a cache was complicated and had never before been designed for a multiprocessor VAX system. Our final decision was based on the need to reduce overall risk to the program. Therefore, we would implement the more straightforward write-through cache design and build the extra bandwidth into the XMI to handle the additional write traffic.

Once the decision to implement a write-through cache was made, the major architectural issue for the processor module became the cache organization. The CVAX chip contains an internal 1-kilobyte (KB), two-way set associative cache accessible to the internal micro engine in one cycle. Due to the long latency to main memory, a second-level cache on the processor module was imperative. The size of the second-level cache was determined by the available static random-access memory (SRAM) technology. The newly available high-speed 64K-by-4 SRAMs would provide a 256KB cache with only eight parts. Although no accurate simulation was available to indicate the effect of this large cache, the effects were assumed to be positive. Therefore we decided that the higher cost of the SRAMs was a worthwhile trade-off given the potential gains in system performance.

A third major issue relative to the caches on the processor module was the invalidation scheme. In the past, VAX processors have managed cache invalidation, since processors and I/O devices have always shared a common memory subsystem. The issue of cache invalidation became much more important to our program because of the multiprocessor nature of the VAX 6200 system. This type of system could cause large amounts of stale data as a process migrates from processor to processor.

The 1KB cache contained within the CVAX chip caused the largest problem. If it were allowed to cache data that could become stale, every write to memory would potentially have to be invalidated within the CVAX cache. This meant choosing one of two approaches: (1) broadcasting every write in the system onto the CDAL bus of every processor, or (2) finding a way to maintain a duplicate tag store of tags within the CVAX chip and only passing writes known to reference cached data within the CVAX onto the CDAL. Another alternative was to cache only instruction-stream (I-stream) data within the internal cache. This alleviates the need to invalidate, because I-stream data is defined to be read-only by the VAX architecture. We projected this alternative could cause a 3 to 5 percent degradation in CPU performance.

Analysis of the cache-invalidate problem proved very difficult, because we did not know what percentage of data would be shared in this class of multiprocessor system. With the potential for 8 processors, it was clear that all writes

could not be broadcast into each of the CVAX chips. The possibility of maintaining a duplicate external tag store proved to be very difficult to implement. Consequently, we chose the alternative to store only I-stream data within the internal CVAX cache.

A similar problem was knowing when to invalidate data in the external cache. In this case it was feasible to implement a duplicate tag store. The second-level cache has two tag stores. One is located on the CDAL and is used for cache look-up by the CVAX chip. The second tag store is located within the XMI interface and is used to determine if XMI writes hit the second-level cache. When hits are detected, a request is queued to invalidate the entry within the second-level cache.

Another problem to be solved on the processor module was the issue of combining writes into larger blocks before issuing them to the XMI. Since the CDAL data path is only 32 bits wide, the CVAX chip is incapable of generating a write command any larger than 32 bits. The 64-bit data path of the XMI would need larger writes to operate efficiently. The solution to this problem was to implement a "write buffer" in the XMI interface of the processor module. The write buffer takes advantage of the fact that writes generated by VAX processors are often sequential. The write buffer will buffer up to four sequential 32-bit writes and combine them into a single XMI write transaction.²

The Memory Module

The system design goal was to provide the capacity for 15MB to 30MB of memory per processor. As mentioned earlier, the module size was partially governed by the need for 32MB of memory per memory module. The number of slots in the XMI backplane was also partially determined by the desired amount of system memory.

The wide range of possible VAX 6200 configurations dictated the need for an expandable memory subsystem. Since full memory bandwidth would only be necessary for very large configurations, it was decided to adopt a distributed memory architecture. An individual memory controller could be made simpler if it did not have to supply full XMI bandwidth. Full XMI bandwidth could be achieved by interleaving multiple memory controllers.

With the module size and number of slots determined, the first architectural decision to be

made for the memory was internal organization of the memory. The 64-bit width of the XMI made it desirable to have a 64-bit data path internal to the memory module. The very tight module real estate made it very attractive to consider implementing a 64-bit data path to reduce the number of required ECC check bits. A 64-bit-wide data path was also attractive given that the processor module would issue a read for 32 bytes whenever there was a cache miss.

The negative side of a 64-bit internal memory organization was that any write of less than 64 bits in width would result in a read-modify-write operation to calculate the proper ECC code. An analysis of the expected write traffic through the processor's write buffer showed that approximately 50 percent of all writes would be a full 64 bits in width. Further analysis showed that as long as there was at least one memory controller for every 2 processors, there would be sufficient memory bandwidth for the system. Given the performance characteristics of the CVAX processor, it seemed reasonable to require a 32MB memory array for every 2 processors. We therefore decided to implement the 64-bit memory internal organization.

Since it was very difficult to design the memory module to accommodate the full bandwidth of the XMI, designers used memory interleaving to provide an aggregate memory bandwidth compatible with the speed of the XMI bus. The interleave size of 32 bytes was determined by the protocol of the XMI, which allows reads of 32 bytes per address cycle.

The multiprocessing design of the system made it possible for a single memory controller to be the object of several simultaneous requests. To avoid rejection of processor traffic, we designed the memory controller with an input queue. This queue accepts memory access requests and services them in a first-in, first-out (FIFO) order.

Initially the memory controller was designed with a four-command queue that would reject new requests once the queue was full. As the design progressed, we realized that with our XMI arbitration scheme, a processor or VAXBI channel adapter might possibly be denied memory access. A processor or channel adapter might be denied access for indeterminate periods of time if the memory array was allowed to reject commands when its queue became full. To avoid this problem, the memory array was allowed to assert a

signal on the XMI that would inhibit all new commands from being issued on the XMI. Unfortunately, due to the pipelined nature of the processor and the memory array, three additional commands could possibly be received by the memory controller after it had determined the need to stop additional requests. Since the depth of the command queue was four, the memory array would need to "stall" the bus after receiving only a single command. Since this effectively eliminated the command queue, we decided to lengthen the depth of the command queue to eight entries.

The VAX architecture forces the use of a hardware-based memory lock to control access to shared data structures. The memory lock is used by some intelligent I/O adapters as well as processors.

System performance suffers when there is conflict over different lock variables that acquire a common hardware lock. Given that Digital had never built a fully symmetric multiprocessor system and that major changes were being made to VMS, we did not know what the lock traffic pattern would look like in a large system. We did know, however, that the existing VAXBI I/O adapters and the CVAX processor could not hold more than a single hardware lock at one time. Based on this, we designed the memory controller to have up to 16 locked locations. This number seemed more than adequate given a maximum of 8 processors and only three existing VAXBI I/O adapters that use memory locks (Ethernet, CI, and TK50). The granularity of each lock is 32 bytes to simplify the memory controllers' handling of 32-byte read requests. Lock congestion is still possible if multiple lock variables are allocated within the same 32-byte region of memory. An examination of VMS code shows lock congestion to be very rare.

VAXBI Channel Adapter

We had decided right from the start to design an XMI-to-VAXBI channel adapter to handle all I/O. To meet the desired maximum I/O rates of 1MB to 1.5MB per second for each processor, we would include multiple XMI-to-VAXBI adapters. Although two VAXBI channels would allow 1.5MB per second per processor, it was decided to allow up to eight VAXBI channels to be connected to the VAX 6200 system. The design was not made more complex by the change from two to eight VAXBI channel adapters.

Designers wanted to optimize data transfer from the VAXBI into XMI memory, since statistically more data is read from I/O devices than is written. A double-buffered direct memory access (DMA) data path from the VAXBI to the XMI allows transfers at the full 13.3MB per second VAXBI data rate.

For reads of XMI memory, it was known that full bandwidth could not be maintained due to the memory read latency through the VAXBI channel adapter and the XMI memory subsystem. Since most I/O transfers are sequential, we considered building prefetch buffers into the VAXBI channel adapter. Because transfers could be in progress to several VAXBI nodes at once, multiple prefetch buffers would be needed. Since prefetch buffers can architecturally be considered to be small caches, the VAXBI channel adapter would also have to monitor all XMI traffic for potential invalidate conditions. Eventually the need for large amounts of buffer storage and the complication of XMI monitoring decided us against building prefetch buffers. This decision was influenced by other factors as well. No single existing VAXBI I/O adapter could read at full bandwidth, and multiple I/O devices could be spread across several VAXBI channels to achieve a higher aggregate XMI read bandwidth.

To ease physical implementation, the VAXBI channel adapter was implemented on two modules that were interconnected by four 60-pin cables between their I/O pins. Unlike the VAX 8500/8700/8800 VAXBI channel adapter, the VAX 6200 could not use a single XMI module to connect to multiple VAXBI buses. The 6200 is restricted by the 120 I/O pins available on an XMI module.

Console Subsystem

The console function in a VAX 6200 system is performed by code run on the CVAX CPUs. This use of the main CPU-based console contrasts with the more traditional use of a dedicated front-end processor, which has access to all system resources. We chose to use a main CPU-based console primarily because we had no way to externally access the internal state of the CVAX processor. Furthermore, we did not want to add the cost of a dedicated console processor.

A side benefit to a system design that employs multiple processors, memories and I/O adapters, is the opportunity to design in extra availability by reconfiguring the system in the event of a single component failure. To accommodate for

reconfiguration, all processors would have to be allowed access to the physical console terminal as well as the physical front control panel of the system. This access is accomplished by busing the signals that interface to the console terminal and front control panel across the XMI. However, we needed a mechanism to ensure that only one of the processors would actually respond to the console terminal and front control panel. This mechanism is a protocol whereby the processor in the lowest XMI slot that passes self-test assumes control of these external resources. The processor that takes control of the console terminal is known as the "primary processor." The primary processor communicates with all other processors by means of a message passing protocol through system memory.

It is necessary for the console subsystem to have access to a mass storage device. Such access is needed for distribution of software and for loading of diagnostics. The TK50 was selected because of its high density and the availability of a preexisting VAXBI interface (the DEBNK). The TK70 was not used because there was no VAXBI interface to it. The only other alternative was the RX50, which has superior access time but a data capacity of only 400KB. The longer time to run diagnostics from the TK50 was unimportant since the system can be diagnosed largely by means of diagnostics contained in CPU read-only memory (ROM) and by the built-in self-test contained in all VAXBI I/O adapters. Further, the TK50 makes an excellent software distribution device and allows VAX 6200 systems to be configured without nine-track magtape drives.

In previous systems dependent on ROM-based console programs and ROM-based diagnostics, code updates have been a problem. To alleviate the need to physically change the ROM, each VAX 6200 processor contains a 32KB electronically erasable ROM (EEROM). Most console and diagnostic code is accessed by means of an address table contained in the EEROM. In the event that a code bug needs to be corrected, the address table is rewritten to point to a replacement routine that is also written into the EEROM. The console program implements a routine that can patch the EEROM image from a database distributed on TK50 tape.

Power and Packaging

As noted earlier, we projected that the VAX 6200 system would be used as a large system generally located in computer-room environments. An

important goal was to design for a high degree of flexibility and configurability in the system. The decision to use a 14-slot XMI backplane had been based on desired maximum configurations and the size of existing pedestal and rack-mount packages.

In addition to housing the XMI backplane, the computer-room package would need to house VAXBI backplanes to accommodate I/O adapters. The VAXBI backplane is manufactured in cascadeable 6-slot segments. It seemed that two 6-slot VAXBI backplanes would provide adequate I/O adapters for most systems. To ensure that all customers' I/O requirements could be met, a design was also initiated for a VAXBI expander cabinet that could house four additional 6-slot VAXBI backplanes.

To avoid developing a new power subsystem, we looked into modifying an existing power system. Although we could find no preexisting perfect match, we did locate a previously designed 5 volt (V) regulator specified at 100-ampere (A) output. We respecified this design to 120 A by using slightly higher power components. The VAXBI requires ± 12 V, -5.2 V, and -2 V in addition to the main $+5$ V channel. To accommodate the VAXBI requirements, a new regulator was designed. The XMI backplane is supplied with two of the 5 V regulators (one for main logic and one for memory). Although not required by any current designs, one of the ± 12 V, -5.2 V, and -2 V regulators also supplies the XMI for potential future designs. The two VAXBI backplanes are supplied by one $+5$ V regulator and one of the ± 12 V, -5.2 V, -2 V regulators.

Conclusion

The design of a complex system like the VAX 6200 is much more than making well-informed engineering decisions based on hard data. Engineers based the initial system definition on their perceptions of the needs for future computing systems. The definition was further shaped by what was technically feasible with a defined degree of risk. Throughout the architectural specification phase, many trade-offs were made with only partial data and the intuitive insight of very experienced engineers.

The design process for the VAX 6200 system was extremely smooth, and the product was designed within six months of the initial engineering goals. Due to the large degree of built-in configuration flexibility, the product definition never changed enough to force a change in direc-

tion during the design phase. Careful balancing of technical complexity with the necessary minimum functionality yielded an architecture that could be implemented with a manageable amount of risk in a bounded amount of time.

Acknowledgments

The initial VAX 6200 system definition and architectural group was made up of the following people: Brian Allison, Charlie Barker, Frank Bomba, Darrel Donaldson, Rick Gillett, Dave Hartwell, Dave Ives, Jim Stegeman, Pat Sullivan, Mike Uhler, and Doug Williams.

References

1. R. Gamache and K. Morse, "VMS Symmetric Multiprocessing," *Digital Technical Journal* (August 1988, this issue): 57-63.
2. R. Gillett, "Interfacing a VAX Microprocessor to a High-speed Multiprocessing Bus," *Digital Technical Journal* (August 1988, this issue): 28-46.

Interfacing a VAX Microprocessor to a High-speed Multiprocessing Bus

The design decisions involved in interfacing a microprocessor (CVAX) to a high-speed, shared-memory multiprocessing bus (XMI) are more complex than those encountered in designing a single-processor system. Although the same basic interface architectures are used, the significantly different multiprocessing environment requires a much more complex implementation. In particular, the performance of a multiprocessor system is very dependent on the efficiency of its main memory interface. To achieve the desired system performance, appropriate compromises between design complexity and performance must be made. In the case of the VAX 6200 system, performance simulations made early in the project guided the complexity/performance trade-offs. Actual system performance results have largely confirmed the validity of the design trade-offs.

The primary goal of the VAX 6200 design was to provide a general-purpose, high-performance, mid-range VAX computing system. Further, this system design would take advantage of Digital's proprietary CMOS technology and VMS version 5.0 symmetric multiprocessing capabilities. VMS version 5.0 has dramatically changed the way we approach mid-range system design; no longer do we design a system to support just one or two processors. With the ability to effectively utilize the power of four or more processors within the same system came the need to design significantly higher performance interconnects to tie these processors together.

The VAX 6200 was to be Digital's first CMOS multiprocessor system. The designers were therefore strongly motivated to provide the best performing product they could within reasonable time and complexity constraints. Complexity was of particular concern since the product schedule did not allow for the production of second-pass parts prior to the first shipment to customers. Complex multiprocessor interfaces give ample opportunities for the kinds of elusive design bugs that can be very difficult and time-consuming to exercise and diagnose. In addition, unlike other recent VAX systems, the VAX 6200 system required a major new release of VMS. (In many ways the new release represented a new operat-

ing system.) We expected its availability could be the critical path to product shipment.

The operating system software would probably not stabilize in time for us to discover and fix any major hardware problems and still stay on our original schedule. Unfortunately, until the operating system stabilizes, testing for complex bugs is difficult. This concern about complexity relative to the schedule affected several design decisions.

On the VAX 6200 CPU module, the design challenge was to interface a custom CMOS VAX microprocessor (called CVAX) to a high-speed multiprocessor bus (called XMI). The trade-offs made during the design of a multiprocessor system are more complex than those made in designing a single-processor system. For a single-processor system, the performance trade-offs are relatively straightforward. The goal is to design the highest performance single-processor system that is practically possible. For a multiprocessor system, the goal of maximum single-processor performance must be tempered to obtain maximum system throughput (i.e., multiprocessor performance).

The foundation of the CPU interface is the cache subsystem, which reduces the effective read access time to main memory. By reducing the processor's need to access main memory, a

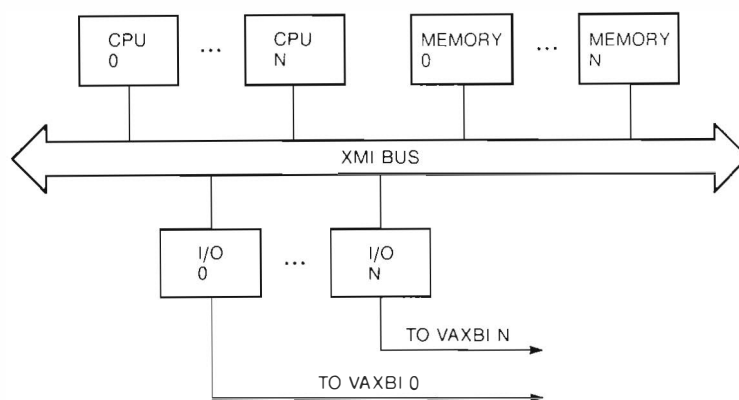


Figure 1 VAX 6200 System Architecture

cache improves both single-processor and multiprocessor system performance. This paper discusses the complexities involved in choosing the optimal cache design and the simulation techniques used to ensure informed design decisions.

One of the biggest problems in cache design is choosing the correct set of workloads to characterize the cache performance. Cache performance can vary tremendously with different workloads. Therefore, we chose a set of workloads that spanned a wide range of system activities. Toward the end of this paper, we present actual cache performance results that largely confirm the legitimacy of our approach.

We also examine one of the more complex aspects of multiprocessor designs, which is ensuring cache coherency across the entire system. Cache coherency refers to the maintenance of a sufficiently consistent memory state from the perspective of all processors and I/O devices within the system.

The designers also went to great lengths to ensure maximum system reliability. As part of this effort, we generated a set of error-detection and response rules. These rules ensure that the operating system software can easily recover from almost all transient cache or bus failures. These rules are discussed.

The following section is an overview of the VAX 6200 system architecture. It provides a basis for the subsequent discussions on the challenges of multiprocessor design, the VAX 6200 CPU responses to those challenges, the performance simulation environment, cache coherency and error handling, and finally, real performance results.

Summary of VAX 6200 System Architecture

The basic architecture of the VAX 6200 system shown in Figure 1 is no different from architectures used on recent VAX systems.¹ The architecture most closely resembles that of the VAX 8800 series. Processors and memories reside on a single, high-speed interconnect called the XMI bus. All memory is shared and equally accessible by all processors. Adapters to the VAXBI bus also attach to the XMI. I/O devices, in turn, are attached to the VAXBI buses. The XMI supports a total of 14 slots, which can be populated with modules to provide a wide range of system configurations. These configurations can range from small single-processor systems with 32 megabytes (MB) of memory and a single I/O channel to a large multiprocessor system with 256MB of memory and multiple I/O channels. One of the primary system design goals was to support up to eight processors with very good multiprocessor performance. This goal guided the performance decisions concerning the bus, memory, and processor designs.

The heart of the system, the XMI bus, is largely a hybrid of the VAX 8800 NMI and VAXBI buses. The XMI is a synchronous bus that runs with a 64-nanosecond (ns) cycle time. The data path is 64 bits wide, and the maximum transfer rate is 100MB per second. The protocol supports "pended reads" (as does the SBI on the VAX-11/780 system and the NMI on the 8800). In a pended read transaction, the CPU that wishes to read a location requests use of the bus. When the request is granted, the CPU transmits the address of the desired location. The appropriate

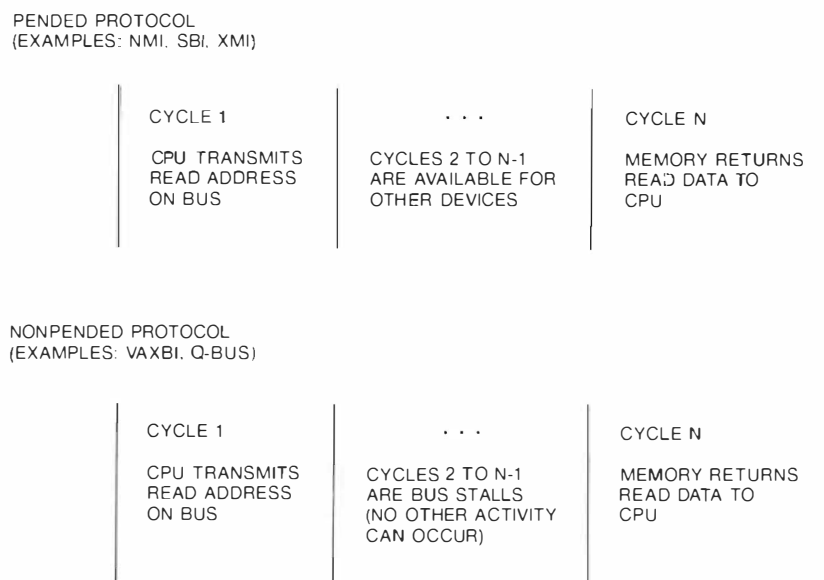


Figure 2 Pended versus Nonpended Protocols

memory controller latches the address into an input queue and begins a read access to the specified location. In the meantime, bus ownership is relinquished by the CPU, and the bus may be used by other devices. When the memory has completed the look-up and has the data, it makes a request for the bus. When granted the bus, the memory drives the requested data on the bus, which is latched by the CPU that originally requested the data. Pended protocols are contrasted with nonpended protocols in Figure 2.

Pended protocols are a big advantage when the bus cycle time is significantly less than the memory access time. As a case in point, a memory read on the XMI bus requires about 500 ns (roughly 8 XMI cycles). Without a pended protocol, these 8 cycles on reads would result in wasteful bus stalls. Another advantage of pended protocols is that they allow multiple memory controllers to be used to advantage. In the case of the VAX 6200, it was not practical to build a single memory controller that could keep up with a saturated XMI bus. But it was relatively easy to construct a memory controller that could comfortably run at about one third the bus maximum. With four interleaved memory controllers on the XMI, memory controller bandwidth is greater than XMI bandwidth.

Challenges of Multiprocessor Design

The major challenges faced by the multiprocessor system designer result primarily from one simple system characteristic. The intimate interface between processor and memory that most single-processor systems enjoy must be broken, and main memory must be shared among a large number of devices. This sharing has several effects:

- Main memory access time is significantly increased.
- Bandwidth to main memory becomes a precious commodity that determines overall system performance.
- Complexity results from increased bus traffic and parallel activities.

In the following sections, we expand on each of these effects in relation to the VAX 6200 system.

Increased Main Memory Access Time

In a single-processor system, main memory is generally closely coupled to the CPU. An example of this closely coupled architecture is shown in Figure 3. Clearly, this architecture provides the potential for low-latency and high-bandwidth CPU-to-memory transactions.

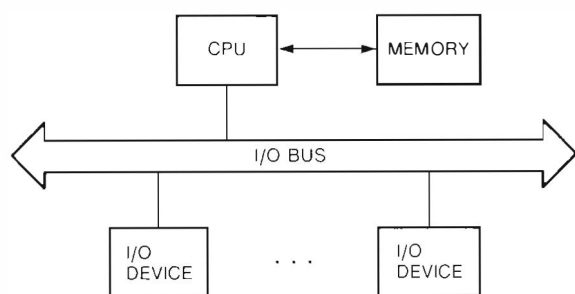


Figure 3 Typical Single-Processor Architecture

In the VAX 6200 multiprocessor system, memory must be shared by several devices and therefore cannot be closely coupled to a single processor. The result is a significant increase in main memory access time. Since the MicroVAX 3600 and VAX 6200 systems are both CVAX-based, a comparison of the main memory access times for the two systems illustrates this point. Table 1 shows the access time in processor cycles for the two-level cache subsystem and the main memory.

Table 1 shows that the VAX 6200 takes three times as many processor cycles to access the first longword in memory as does the MicroVAX 3600 system. The main reason for this difference is that the MicroVAX 3600 memory controller actually resides on the CPU module. Therefore, the system architecture is optimized to provide minimum access time for processor accesses to main memory. On the VAX 6200, system memory is a shared resource equally accessible by all CPUs and I/O devices. The price of this equality is increased latency on all memory references. Note however that although latency has increased, the VAX 6200 can support almost ten times more memory bandwidth (the time required per unit of data transferred).

As will be later presented, the VAX 6200 system uses memory bandwidth to compensate for increased memory latency. Trading bandwidth for latency is one of the fundamental tools of the multiprocessor designer. Cache memory systems essentially convert increased memory bandwidth (manifested as a larger fill size) into lower average read latency (due to the decreased miss rate in the cache resulting from the larger fill size). This explanation is an oversimplification; details of the trade-offs in cache design are pre-

sented below in the section on the multiprocessor environment.

Table 2 reinterprets the data in Table 1 in terms of bandwidth instead of latency. For example, the MicroVAX 3600 system fetches 8 bytes of data from memory on a cache miss, which requires 8 (90 ns) processor cycles, or 720 ns. This corresponds to 8 bytes of data every 720 ns, or 11.1MB per second. In comparison, the VAX 6200 system fetches 32 bytes of data on a cache miss, which corresponds to 16.7MB per second (32 bytes of data every 1920 ns).

Table 1 Comparison of MicroVAX 3600 and VAX 6200 Memory Latency

	VAX 3600	VAX 6200
Cache 1 (CVAX internal cache)	1 (90 ns)	1 (80 ns)
Cache 2 (Second-level cache)	2 (180 ns)	2 (160 ns)
Main Memory		
First longword	5 (450 ns)	14 (1120 ns)
Second longword	8 (720 ns)	15 (1200 ns)
Third longword	na	19 (1520 ns)
Fourth longword	na	20 (1600 ns)
Fifth longword	na	21 (1680 ns)
Sixth longword	na	22 (1760 ns)
Seventh longword	na	23 (1840 ns)
Eighth longword	na	24 (1920 ns)

Table 2 Comparison of Processor Read Bandwidths on MicroVAX 3600 and VAX 6200 Systems (in MB per second)

	MicroVAX 3600	VAX 6200
Cache 1 (CVAX internal cache)	40.0	50.0
Cache 2 (Second-level cache)	20.0	25.0
Main Memory		
First longword	8.8	3.6
Second longword	11.1	6.7
Third longword	na	7.9
Fourth longword	na	10.0
Fifth longword	na	11.9
Sixth longword	na	13.6
Seventh longword	na	15.2
Eighth longword	na	16.7

Limited Bandwidth to Main Memory

In a single-processor system such as the MicroVAX 3600, the performance is generally limited by the CPU itself and not by the main memory subsystem. The opposite is generally the case on large multiprocessor systems where a large number of processors can create a bottleneck to the main memory subsystem. A major goal of the multiprocessor designer is to minimize the bandwidth required to support a given level of CPU performance. In that way, the main memory bus can support more processors; therefore, the system can attain higher total throughput. For example, assume a processor requires an average of 20 percent of the total bandwidth available to main memory to run a given workload. Based just on bus bandwidth considerations, the total system performance would not exceed five times the single-processor performance if the system is simultaneously running that workload on all processors. For a number of reasons, systems are rarely designed such that the bus must be saturated to meet its performance goals. This same method of calculating performance can be used to estimate performance at some lower level of bus utilization. A bus utilization level of 75 percent is often used; in that case, the system performance would be limited to 3.75 times the single-processor system.

This example reveals one of the main compromises multiprocessor system designers must make: increased bandwidth, which would reduce the main memory access time seen by a single processor, is traded off to reduce the total bandwidth consumed by a single processor and thereby increase total system throughput. Bandwidth is really not the characteristic we are trying to minimize; the real goal is to reduce the number of bus and memory cycles used to sustain a given level of performance. As we will demonstrate, the efficiency of the transfer generally increases as the transfer size increases. Therefore the system can fetch twice as much data from memory without using twice as many bus and memory cycles. This characteristic is important when evaluating various cache alternatives.

Again looking at the MicroVAX 3600 design, the CPU actually starts accessing main memory once the first-level cache has determined a miss occurred but before the look-up in the second-level cache has completed. This overlap means the memory controller will start a large number

of accesses that will never result in data being returned to the processor. (The second-level cache will probably "hit" on more than 80 percent of these references.) This behavior is desirable for many single-processor systems but would be inappropriate for a multiprocessor design in which main memory bandwidth is precious.

In the multiprocessor system, main memory bandwidth is shared by all processors and I/O devices. Table 3 compares the system bandwidth in the MicroVAX 3600 and VAX 6200 systems. Since the VAX 6200 uses a pended bus that supports 1 to 8 memory controllers, we present two sets of bandwidth numbers for the VAX 6200 memory subsystem: one for a single memory controller and another for a four-way interleaved, four-memory controller subsystem.

The data makes a strong argument for large transfer sizes to achieve high bandwidths on the VAX 6200. A large cache fill size can be used to assure high read bandwidth, and a write buffer can be used to provide longer length write transactions. Note that longword writes are particularly inefficient in the memory controller; nine cycles are required for a longword write compared with only five cycles for a quadword write. This inefficiency results from the implementation of the error-correcting code (ECC) across a quadword on the VAX 6200 memory. (VAX systems have traditionally implemented ECC across a longword.) This implementation improved the memory module capacity at the cost of forcing all longword writes to be a read-modify-write sequence in the memory.

Increased System Bus Traffic

Another challenge to the multiprocessor designer is the increased memory traffic in the system due to the increased total system performance. For a given workload, it is fairly accurate to assume that the traffic to main memory increases linearly with the total performance system. Therefore, a VAX 6240 (a four-processor 6200 system) would have roughly four times the main memory traffic of the VAX 6210 (a single-processor 6200 system). Since processors must monitor main memory traffic to maintain cache coherency, this increase in main memory traffic has to be considered when looking at cache invalidate implementations. Again the single-processor system has a much less severe problem. The single processor has to monitor only the traffic from I/O devices,

Table 3 MicroVAX 3600 and VAX 6200 Main Memory Bandwidth (in MB per second with corresponding number of cycles in parentheses)

	MicroVAX 3600* (90-ns cycles)	VAX 6200 XMI Bus (64-ns cycles)	VAX 6200 Memory (64-ns cycles)	
Reads			1 Memory	4 Memories
Longword (4B)	8.8 (5)	31.2 (2)	10.4 (6)	41.6
Quadword (8B)	11.1 (8)	62.2 (2)	20.8 (6)	83.2
Octaword (16B)	na	83.3 (3)	31.2 (8)	124.8
Hexword (32B)	na	100.0 (5)	38.5 (13)	154.0
Writes				
Longword (4B)				
Full	11.1 (4)	31.2 (2)	6.9 (9)	27.6
Masked	6.3 (7)	31.2 (2)	6.9 (9)	27.6
Quadword (8B)				
Full	na	62.2 (2)	25.0 (5)	100.0
Masked	na	62.2 (2)	13.9 (9)	55.6
Octaword (16B)				
Full	na	83.3 (3)	31.2 (8)	124.8
Masked	na	83.3 (3)	16.7 (15)	66.8

* These numbers represent a CPU perspective. I/O devices on the Q-bus can use longer transfer lengths.

which typically generate about one-tenth the traffic generated by a single CPU. Extending this argument, it appears to indicate that a VAX 6240 system must handle invalidate look-ups at a rate more than 30 times that of the MicroVAX 3600 system. (The VAX 6200 CPU has to handle invalidates from three other CPUs and for about four times as much I/O traffic.)

The increased system bus traffic is a symptom of the large number of parallel activities that characterize a multiprocessor system. The abundance of queues in a multiprocessor system results in a more complex system. The section on cache coherency in this paper discusses several manifestations of this increased complexity.

Table 4 summarizes the major differences between the single-processor and multiprocessor systems.

This discussion has demonstrated that the performance of a multiprocessor system is very dependent on the designers making the right decisions about the CPU interface. In the next section, we discuss the basic architecture of the VAX 6200 CPU and specific aspects of the multiprocessor environment.

VAX 6200 CPU Design Alternatives

This section presents an overview of the VAX 6200 CPU architecture, followed by a discussion of the various implementation alternatives that we considered during the design process. We conclude with a list of specific design alternatives and a discussion of our performance simulation environment, which we used to examine these alternatives.

Table 4 Summary of Differences between Single-processor and Multiprocessor Systems

Characteristic	Single-processor System	Multi-processor System
Memory latency	Low	Medium
Performance bottleneck	CPU	Memory bandwidth
Invalidate rate	Low	High
Level of parallel activity	Low	High

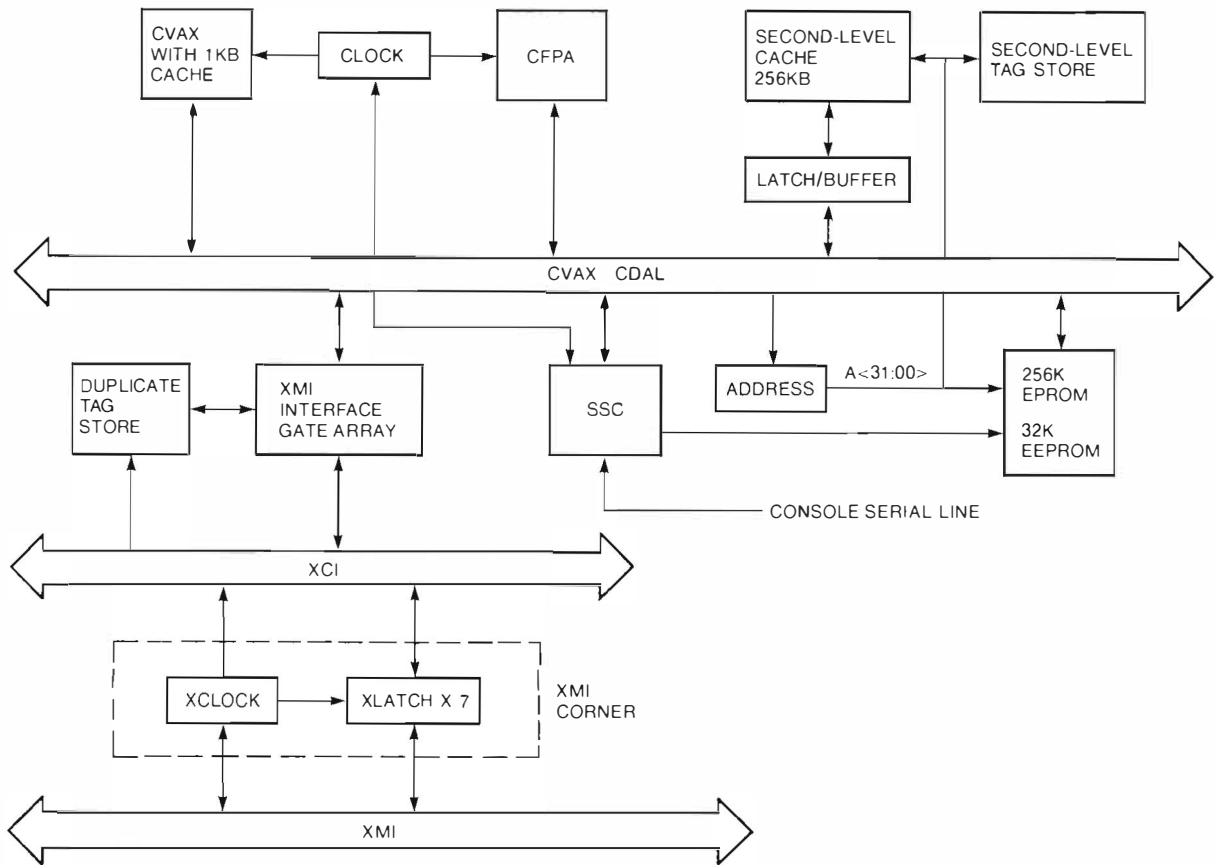


Figure 4 VAX 6200 CPU Block Diagram

The VAX 6200 CPU is a single-board VAX processor based on the CVAX chip set designed and built by Digital. The 6200 CPU has a CVAX cycle time of 80 ns (as compared to the MicroVAX 3600 90-ns CVAX cycle time); its nominal performance is 2.8 times the VAX-11/780 system (slightly more than three times the MicroVAX II).

A block diagram of the module is shown in Figure 4. Three major buses are associated with the module. The CVAX processor chip set communicates over the CVAX data and address bus (CDAL).^{2,3} The SSC chip connects to the CDAL bus and provides such functions as read-only memory (ROM) address decoding, time-of-year clock support, and console terminal interface.⁴ The CVAX chip contains the first-level cache. Also connected to this bus is the second-level cache data store and tag store logic. The path to the XMI bus is provided entirely by the XMI interface gate array and the XMI corner. This gate array provides all necessary synchronization between the CVAX and XMI. Each CPU module has its own CVAX

clock source, and the XMI bus has a single clock source that provides synchronous clock signals to all XMI nodes.

The XMI corner represents a standard set of interface components and a physical interconnect that ensure all XMI devices meet the timing and electrical characteristics required by the XMI specification. The XMI corner components interface to the rest of the logic on the module over the XMI chip interconnect (XCI). A duplicate tag store also attaches to the XCI bus.

As outlined in the previous section, several specific challenges must be addressed by the multiprocessor designer. At the CPU level the design responses are as follows:

- Implement an effective cache to reduce the effective access time and the total traffic to main memory
- Implement a write buffer to decouple and reduce write traffic.

- Implement a duplicate tag store to reduce the overhead and complexity of maintaining cache coherency.

Cache Subsystem

We will first look at the issues associated with designing an effective cache. The main characteristics of a cache are size, associativity, fill size, and block size. Size is simply the size in bytes of the data store section of the cache. As the size of the cache increases, the effectiveness of the cache also increases. Associativity refers to the number of sets in the cache. A cache with a single set can store data with a particular tag address in only a single location. (A single-set cache is often referred to as a direct-mapped cache.) A two-set cache has two locations capable of storing data with a particular tag address. As associativity is increased, the likelihood of cache "thrashing" decreases. (Thrashing occurs when two pieces of data cannot simultaneously be in the cache due to an insufficient number of sets.) The likelihood of thrashing also decreases as the cache size increases. Therefore it follows in most cases that as the cache size increases, the benefits of increased associativity decrease. Fill size defines the amount of data that is fetched from main memory on a cache miss and loaded into the cache. Over the range of cache sizes of interest, the miss rate decreases as the cache fill size increases. Block size refers to the size of the data block covered by a single tag address. In a direct-mapped cache, the block size is equal to the cache size divided by the number of tags. The fill size is equal to or less than the block size.

A major issue facing the designer of any computing system is the amount of variation in performance that can be accepted over a wide range of workloads. Since we were concerned about our ability to accurately model the effect of large caches, we wanted to err on the side of conservatism. This meant we would choose the largest cache size practical. The state-of-the-art technology static random-access memories (SRAMs) available to the VAX 6200 team were expected to be 256-kilobit (Kb) parts with speeds down to 35 ns. We determined that a pipelined cache design with 35-ns SRAMs could support CVAX cycle times down to 60 ns. This cycle time was comfortably beyond our product goal, which was to support a range of 70 ns to 100 ns, depending on the success we had speed-binning CVAX parts. We tentatively decided to use 64K-by-4 SRAMs for the data store, largely because the 64K-by-4

configuration was expected to be the most readily available. Since the CVAX has a 32-bit data path, eight 64K-by-4 parts would naturally provide a 256KB direct-mapped cache (four times the size of any previous VAX). This configuration also provided the optimal one-output-load per data line. We also examined configurations with increased associativity to confirm our belief that the benefit of set sizes greater than one is small for caches in the range of 256KB.

Having selected a very large cache, we next considered block size and fill size. The XMI bus supports only 8 (quadword), 16 (octaword), and 32 (hexword) byte transfers to memory. Therefore, the fill size would have to be one of these three sizes. The block size can be larger than the fill size if the design supports what are called subblock valid bits. Ideally the fill size and block size would be the same. With a very large cache, however, providing sufficient tag storage can be a real problem. Again in an attempt to be conservative, we looked into state-of-the-art, tag-integrated circuits. The best we found in the required 25- to 30-ns speed range was a 2K-by-9 part. With two of these parts, we could implement a 2K tag store subsystem. A 256KB data store with 2K tags would have a 128-byte fill size. Subblock valid bits would be needed to identify which subblocks are actually valid. We decided it would be practical to choose a larger tag store size in which four tag chips would be used to implement a 4K tag store subsystem.

Choosing the ideal fill size was expected to involve an interesting compromise between several characteristics. As the fill size is increased, several things happen.

- The cache miss rate drops. Over reasonably large ranges, the miss rate can be reduced by requiring that more data be fetched on a cache miss. This is not true when the likelihood of using the new data is less than the likelihood that bringing in the additional new fill data will force the flushing of other cache data more likely to be used.⁵ This will not occur with cache and fill sizes in the range considered for the VAX 6200.
- CPU stalls per miss increase. In VAX 6200 CPU architecture, as the second-level cache is being filled, the CVAX cannot access it. On a second-level cache miss, the XMI interface does return the actual requested data item to the CVAX first and then completes the remainder of the cache fill. Therefore, the number of

cycles in which the CVAX is stalled waiting for the second-level cache to become available again after a cache miss increases as the fill size increases. The CVAX internal cache remains accessible while the second-level cache is being filled.

- The MB per second to main memory required to support a given level of performance increases. If twice as much data is fetched on a cache miss, the miss rate does not drop by a factor of two.⁵ Therefore, as fill size increases, the MB per second required to support a given level of performance increases.
- The “available MB per second” of the bus increases. The efficiency of buses that do not have separate address lines (such as the XMI) increases as the transfer size increases. Basically, the required address cycle can be amortized over more data cycles.
- The “available MB per second” of the memory controller increases. The memory controllers in the VAX 6200 can deliver more MB per second if more data is fetched for a given fetch address.

Based on our significant experience with VAX systems, we knew that either the 16-byte or 32-byte fetch would be the right choice. The results from simulation would be used to select the final value.

Another major cache design issue was the configuration of the CVAX 1KB internal cache. This cache can be configured to run in a conventional instruction and data stream write-through mode. In this mode, the cache must be invalidated when writes occur to a stored block. Alternatively, the cache can be run in I-stream-only mode in which the cache does not have to be invalidated on writes. Instead, the cache is automatically flushed on VAX Return from Exception or Interrupt (REI) instructions. The methods we used to ensure the success of this cache coherency mechanism are discussed in the section Maintaining Cache Coherency and Handling Cache Error Conditions.

Assuming all other things remain equal, there is a performance penalty for choosing the I-stream-only mode. If we select I-stream-only mode, the following occurs:

- All D-stream references will require a minimum of two cycles instead of one. Generally, for VAX CPUs an average of 0.8 D-stream

references are made per instruction⁶ and an average instruction on the CVAX requires between 9 and 10 cycles. This would seem to indicate that the performance penalty would be about 8 percent (0.8 references divided by 9.5 cycles), assuming the D-stream miss rate in the internal cache is 0 percent. With an expected more-typical 40 percent miss rate, the penalty would be about 5 percent.

- CVAX stalls will increase for references that occur while the second-level cache fill for a previous reference is still not complete. This increase results because the CVAX will need to access the second-level cache on all D-stream references.
- Assuming a low frequency of REI instructions, the I-stream miss rate should improve since there will be no contention for cache blocks between the I and D streams. (REIs will cause the I-stream-only cache to flush.)
- The module space needs will be less because there will be no need for an extra duplicate tag to track the CVAX internal cache. Since the CVAX internal cache has two sets, it cannot be practically “followed” by a simple second-level, direct-mapped cache.

Looked at another way, we could afford to devote more logic to making the second-level cache more effective if we did not support CVAX D-stream caching.

- The complexity lessens with one less cache to keep coherent with hardware. We also had more flexibility in implementing error-recovery mechanisms and would not have to implement a complex mechanism to suppress the generation of XMI write transactions when the invalidate queue was at risk of overflowing.

We planned to use the simulation environment to quantify the performance penalty that results from running the CVAX cache in I-stream-only mode.

Write-Buffer Subsystem

Conventional write-through caches greatly reduce read traffic to main memory but do not reduce the write traffic. Therefore, although the mix of read and write references from the CPU itself is weighted heavily toward reads, the traffic downstream of a write-through cache is primarily writes. Other cache architectures offer the potential to reduce write traffic. A write-back cache

might be considered the obvious approach. By caching writes as well as reads, a write-back cache offers the potential for the highest performance multiprocessor system. Nevertheless, the complexity is significantly higher than a write-through design. Industry experience is that very few write-back caches work on first-pass, and their bugs are very difficult to fix. Another risk with write-back caches is in the area of error recovery. It is much more difficult to recover from transient cache errors with a write-back protocol. To avoid the increased complexity and resulting schedule risk, we decided to pursue a hybrid approach. We would implement a write-through cache with a write buffer design very similar to that of the VAX 8800 cache.⁷

A write buffer resides between a write-through cache and the system bus. A write buffer is actually a simple, very effective form of a write-back cache. A write buffer takes advantage of the locality of write transactions to reduce the number of write references to main memory by combining several small write references into a single larger transaction to main memory. This behavior has three main advantages. First, almost all buses (including the XMI) increase in efficiency as the transfer size is increased. This efficiency results because every transfer generally requires the transmission of an address cycle before the data. This address cycle is basically fixed overhead that can be more effectively amortized as the transfer size is increased. The transfer sizes and relative efficiencies of the XMI bus are shown in Table 3.

Second, as previously mentioned, the VAX 6200 memory does not efficiently process longword write transactions. The write buffer converts significant numbers of longword write transactions into full quadword and octaword transactions that are processed with many times greater efficiency.

Finally, the buffer helps to reduce the frequency of processor "write stalls," that is, processor cycle slips due to writes to main memory that back up. The buffer largely decouples the processor from the main memory write timing; the processor perceives that most writes are completed in minimum time.

The VAX 6200 write buffer accumulates write data until a memory write address falls outside the address range of the current block. When this occurs, an alternate octaword buffer begins filling. The first buffer is emptied either with an octaword XMI transaction (if the buffer contains more than an aligned quadword) or with a quad-

word XMI transaction (if the buffer contains no more than an aligned quadword). CVAX CPU reads (unless interlocked or made to I/O space) are allowed to bypass the write buffers after first being checked for an address match with the write buffer.

Either a read address comparison match or an interlocked or I/O space transaction forces the write buffer to be purged. There are several other conditions under which the write-buffer must be flushed. These conditions are discussed in the section Maintaining Cache Coherency and Handling Error Conditions.

We believed the write buffer could provide about half the bandwidth benefit of the write-back cache but with little more complexity than a simple write-through design. As an added benefit, the buffer architecture was already implemented and running with very good performance results in a VAX multiprocessor (VAX 8800 family). We planned to use performance simulations to confirm that the write buffer was adequate to meet our performance goals.

Duplicate Tag Store

As noted earlier, a multiprocessor environment puts significant strain on the cache coherency logic. The rates at which write addresses on the system bus must be checked against the addresses stored in the cache require that a different architecture be used for servicing invalidates.

The 2K-by-9 tag chips used to implement the main tag store are also used to implement a duplicate tag store. The duplicate tag store runs synchronously with the XMI bus and permits filtering of invalidates, so the CPU would stall only on an XMI write hit. It is not uncommon to have ratios of 100 to 10,000 to 1 between duplicate tag misses and duplicate tag hits.

The operation of the duplicate tag store is discussed in the section Maintaining Cache Coherency and Handling Error Conditions.

We have now defined the basic architectural issues that needed to be resolved and have indicated the alternatives we would like to pursue. In the next section we present the results of our performance simulations.

The following list summarizes what we examined in our simulation environment:

- Determine the loss in performance that would result from running the CVAX internal cache in I-stream-only mode instead of combined I- and D-stream mode.

- Investigate octaword (16-byte) versus hexword (32-byte) fill sizes for both I-stream and D-stream. Further, examine the relative miss rates, MBs per unit of performance, bus cycles per unit of performance, memory cycles per unit of performance, and absolute performance. Look at a large multiprocessor system's sensitivity to main memory access time.
- Determine the effectiveness of a write-through with write-buffer cache architecture. In other words, can the writes be reduced sufficiently to avoid write-back in the chosen architecture.
- Examine the benefits of a two-way, set-associative cache over a simpler direct-mapped design.

Performance Simulation

The basis of the simulation environment was a high-level performance model of the CVAX chip. Written in PASCAL, this model was interfaced to a configurable second-level cache, write buffer, and memory subsystem. The model accepted instruction traces for input. At the time the performance modeling was done, seven standard benchmarks were available: DIRECTORY, EDT, FORTRAN, LINKER, MAIL, RUNOFF, and SORT. All instruction traces were captured from a VAX-11/780 system. Since each trace was for a single process, one of the major issues was determining how to correctly model the effect of timesharing on cache performance.

The very nature of timesharing has a negative effect on cache performance as compared with single process runs. Ideally, the cache would be dedicated entirely to holding instructions and data associated only with a single process. In timeshare systems, processes are not initiated and then run nonstop to completion; instead the CPU is constantly switching from process to process. This switching requires the cache resources to be distributed across a number of processes and therefore reduces the effectiveness of the cache. A VAX-11/780 study⁶ indicates that the average number of instructions between context switches on a VAX system is about 5,000 instructions. A traditional and very conservative approach to simulating the effect of context switches is to flush the entire cache every 5,000 instructions. Flushing the cache every 5,000 instructions was not a big penalty for small caches that could quickly refill themselves after a flush; however, the advantage of larger caches (that

we know actually exists) could not be demonstrated when the model ran with a flush every 5,000 instructions.

To more accurately model the benefits of large caches, internal studies of complex timeshare loads were undertaken. Multiuser program traces were run against a cache model, subjecting the cache model to the context-switch behavior of a real system. The cache performance results of that run were compared with single jobs run against a cache model that was flushed after various numbers of instructions had been executed.

The results indicated that similar cache performance results could be obtained in simulation by using a single job trace and complete cache flushing every 35,000 instructions. The number 35,000 applies only to a 256KB cache; smaller caches would have a smaller context-switch interval. We decided to simulate the VAX 6200 with the 256KB cache flushed every 35,000 instructions; the 1KB CVAX internal cache would be flushed at the more traditional 5,000 instruction rate. All simulations would represent a single-processor system; main memory access times would be minimum. The performance results would generally be presented as a set of relative numbers comparing the alternatives.

Table 5 summarizes all the cache characteristics we would simulate.

CVAX Internal Cache Configuration

The first aspect examined was the CVAX cache configuration. As shown in Table 6, the I- and D-stream design offered an average increase in performance of 5 percent over the I-stream-only cache. We concluded 5 percent average performance could be sacrificed in return for the reduced complexity of the I-stream-only design.

Table 5 Cache Characteristics Simulated

	CVAX Cache	Second-level Cache
Associativity	2-way	Direct-mapped/2-way
Configuration	I & D/I only	I & D
Size	1KB	256KB
Block size	8B	64B
Fill size	8B	16B/32B
Tags	1K	4K
Simulated context switch rate	5,000 instructions	35,000 instructions

**Table 6 CVAX I-stream and I- and D-stream
Relative Performance**

	I-stream	I- & D-stream
Average	1.00	1.05
Minimum	1.00	1.03
Maximum	1.00	1.07

Octaword versus Hexword Fill Size

Choosing an octaword or a hexword fill size was the next and probably the most complex major issue. The results are shown in Table 7. In all cases, relative numbers are used with the characteristics of the octaword machine as the reference.

Table 7 Octaword versus Hexword Fill Size Results

Relative Fill Size	Performance																																																												
Octaword																																																													
All	1.00																																																												
Hexword																																																													
Average	1.01																																																												
Minimum	1.01																																																												
Maximum	1.02																																																												
<table border="1" style="width:100%; border-collapse: collapse;"> <thead> <tr> <th rowspan="2">Fill Size</th> <th colspan="3">Relative Miss Rates</th> <th colspan="3">Relative MB/sec</th> </tr> <tr> <th>I-stream</th> <th>D-stream</th> <th>All Reads</th> <th>I-stream</th> <th>D-stream</th> <th>All Reads</th> </tr> </thead> <tbody> <tr> <td>Octaword</td> <td colspan="6"></td> </tr> <tr> <td> All</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> </tr> <tr> <td>Hexword</td> <td colspan="6"></td> </tr> <tr> <td> Average</td> <td>.56</td> <td>.84</td> <td>.71</td> <td>1.12</td> <td>1.68</td> <td>1.42</td> </tr> <tr> <td> Minimum</td> <td>.54</td> <td>.81</td> <td>.68</td> <td>1.08</td> <td>1.61</td> <td>1.36</td> </tr> <tr> <td> Maximum</td> <td>.57</td> <td>.87</td> <td>.76</td> <td>1.14</td> <td>1.74</td> <td>1.52</td> </tr> </tbody> </table>							Fill Size	Relative Miss Rates			Relative MB/sec			I-stream	D-stream	All Reads	I-stream	D-stream	All Reads	Octaword							All	1.00	1.00	1.00	1.00	1.00	1.00	Hexword							Average	.56	.84	.71	1.12	1.68	1.42	Minimum	.54	.81	.68	1.08	1.61	1.36	Maximum	.57	.87	.76	1.14	1.74	1.52
Fill Size	Relative Miss Rates			Relative MB/sec																																																									
	I-stream	D-stream	All Reads	I-stream	D-stream	All Reads																																																							
Octaword																																																													
All	1.00	1.00	1.00	1.00	1.00	1.00																																																							
Hexword																																																													
Average	.56	.84	.71	1.12	1.68	1.42																																																							
Minimum	.54	.81	.68	1.08	1.61	1.36																																																							
Maximum	.57	.87	.76	1.14	1.74	1.52																																																							
<table border="1" style="width:100%; border-collapse: collapse;"> <thead> <tr> <th rowspan="2">Fill Size</th> <th colspan="3">Percent XMI</th> <th colspan="3">Percent Memory</th> </tr> <tr> <th>I-stream</th> <th>D-stream</th> <th>All Reads</th> <th>I-stream</th> <th>D-stream</th> <th>All Reads</th> </tr> </thead> <tbody> <tr> <td>Octaword</td> <td colspan="6"></td> </tr> <tr> <td> All</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> <td>1.00</td> </tr> <tr> <td>Hexword</td> <td colspan="6"></td> </tr> <tr> <td> Average</td> <td>.93</td> <td>1.40</td> <td>1.18</td> <td>.91</td> <td>1.36</td> <td>1.15</td> </tr> <tr> <td> Minimum</td> <td>.90</td> <td>1.34</td> <td>1.16</td> <td>.88</td> <td>1.31</td> <td>1.13</td> </tr> <tr> <td> Maximum</td> <td>.96</td> <td>1.45</td> <td>1.27</td> <td>.95</td> <td>1.41</td> <td>1.27</td> </tr> </tbody> </table>							Fill Size	Percent XMI			Percent Memory			I-stream	D-stream	All Reads	I-stream	D-stream	All Reads	Octaword							All	1.00	1.00	1.00	1.00	1.00	1.00	Hexword							Average	.93	1.40	1.18	.91	1.36	1.15	Minimum	.90	1.34	1.16	.88	1.31	1.13	Maximum	.96	1.45	1.27	.95	1.41	1.27
Fill Size	Percent XMI			Percent Memory																																																									
	I-stream	D-stream	All Reads	I-stream	D-stream	All Reads																																																							
Octaword																																																													
All	1.00	1.00	1.00	1.00	1.00	1.00																																																							
Hexword																																																													
Average	.93	1.40	1.18	.91	1.36	1.15																																																							
Minimum	.90	1.34	1.16	.88	1.31	1.13																																																							
Maximum	.96	1.45	1.27	.95	1.41	1.27																																																							
<table border="1" style="width:100%; border-collapse: collapse;"> <thead> <tr> <th>Fill Size</th> <th>Relative Percent XMI Utilized</th> <th>Relative Percent Memory Utilized</th> </tr> </thead> <tbody> <tr> <td>Octaword</td> <td colspan="2"></td> </tr> <tr> <td> All</td> <td>1.00</td> <td>1.00</td> </tr> <tr> <td>Hexword</td> <td colspan="2"></td> </tr> <tr> <td> Average</td> <td>1.08</td> <td>1.04</td> </tr> <tr> <td> Minimum</td> <td>1.06</td> <td>1.03</td> </tr> <tr> <td> Maximum</td> <td>1.09</td> <td>1.07</td> </tr> </tbody> </table>							Fill Size	Relative Percent XMI Utilized	Relative Percent Memory Utilized	Octaword			All	1.00	1.00	Hexword			Average	1.08	1.04	Minimum	1.06	1.03	Maximum	1.09	1.07																																		
Fill Size	Relative Percent XMI Utilized	Relative Percent Memory Utilized																																																											
Octaword																																																													
All	1.00	1.00																																																											
Hexword																																																													
Average	1.08	1.04																																																											
Minimum	1.06	1.03																																																											
Maximum	1.09	1.07																																																											

A summary of the results in Table 7 follows:

- The fill size has a negligible effect on performance (less than 1 percent difference). The hexword alternative delivered an average of 1 percent better performance.

It is important to keep in mind that the simulation was performed assuming the minimum delay from main memory. In a multiprocessor system, the alternative with the lower miss rate increases in performance relative to the other alternatives as the main memory access time increases.

- Hexword fetches dropped the overall miss rate by almost 30 percent. (As expected, the 1-stream miss rate improvement was much higher — almost 50 percent.)
- The megabytes per second required to maintain a given performance level increased by about 40 percent overall for the hexword fetch.
- As mentioned earlier, we were not as concerned about megabytes per second as much as the percentage of the bus and memory controller cycles per second. In this light the hexword alternative required about 18 percent

more bus cycles and 16 percent more memory cycles to support read traffic to main memory. Eighteen percent and 16 percent may seem like a big increase, but it is important to look at overall bus bandwidth. On a write-through interconnect, the writes generally dominate the traffic.

- The overall bus traffic (taking into account writes) increased by only about 9 percent. Overall memory controller cycles increased by even less — only about 4 percent. The low increase resulted because the ratio of write cycles to read cycles is higher in the memory controller than on the XMI bus.

Based on this data, we chose the hexword fill alternative. We felt the potential for significantly more consistent performance in large multiprocessor configurations (due to decreased cache miss rate) was worth the estimated 9 percent increase in bus utilization.

Write Buffer Effectiveness and Overall Bus Utilization

We were pleased to find that the write buffer was about as effective as we had predicted. The data in Table 8 compares the XMI write traffic generated with and without a write buffer. The data is quite consistent. On average, the write buffer reduced the number of write cycles on the bus by slightly less than half (45 percent) and reduced the memory controller cycles by slightly more than half (51 percent).

Table 9 shows the bus utilization by the VAX 6200 CPU running the test benchmarks. Using the average bus utilization number of 6.27 percent still yields only 50 percent for a full eight-processor system; the 7.25 percent maximum value yields 58 percent utilization. These figures are well within our 75 percent utilization design goal, and we decided to implement the write-buffer instead of the write-back design.

Another more conservative way to look at the data is to assume that we may not have the worst-case environment covered in any single benchmark. Therefore we should look at the “sum of maximums” to determine whether the design goal is met. Using the sum of maximums approach, we require 9.72 percent of the XMI per processor, or about 78 percent for eight processors. This figure is sufficiently close to our design goal of 75 percent maximum utilization to be acceptable.

Table 8 Write Buffer Effectiveness

	Write Buffer Miss Rate	Ratio With Write Buffer/ Without Write Buffer*	
		XMI Utilization	Memory Utilization
Average	47.1%	.55	.49
Minimum	40.4%	.50	.42
Maximum	54.9%	.64	.58

* The utilization numbers are expressed as ratios between the utilization with a write buffer and the utilization without the write buffer.

Table 9 XMI Bus Utilization per CPU

	I-stream Reads	D-stream Reads	Writes	Total*
Average	.89%	1.39%	4.41%	6.27%
Minimum	.24%	1.26%	3.57%	5.27%
Maximum	1.65%	2.10%	5.97%	7.25%

* The numbers in this column are averages of the total XMI bus utilization across the seven workloads. These numbers are not sums of the individual utilization percentages in each column.

Effect of Associativity

We next explored the benefits of associativities greater than one. Implementation of a cache other than a direct-mapped cache was probably not practical. However, we wanted to examine the performance results.

The results given in Table 10 indicate that a two-way, set-associative cache could reduce the overall miss rate by 13 percent, whereas the performance gain was negligible (1 percent). This improvement in miss rate is fairly significant; but we determined it was not practical from a module real estate and electrical timing perspective to implement other than a direct-mapped scheme. To implement a fast two-way cache, two separate RAM arrays must be supported. This implementation requires roughly twice the module area of a directed-mapped approach. A two-way cache can be implemented with a single RAM array (cannot start the RAM look-up until the proper set has been identified), but this would force the access time to increase by a cycle. Increasing the access time to the second-level cache would be particularly undesirable to the VAX 6200 designers since we had already decided to configure the CVAX cache in I-stream-only mode. (With an additional cycle, all D-stream references would then require a minimum of three cycles.) Board area constraints and increased cache access time are the two most common reasons for rejecting the miss reductions of the multiway cache in favor of the simplicity and the practical, fast access time of the direct-mapped cache.

Maintaining Cache Coherency and Handling Cache Error Conditions

As mentioned in the introduction, a major challenge to a multiprocessor designer is to implement a reliable scheme for cache coherency. Coherency is a term somewhat difficult to define. In this section, we give some insight into the

Table 10 Direct-mapped versus Two-way Cache Performance

	All Reads Relative Miss Rates		Relative Performance	
	Direct- mapped	Two- way	Direct- mapped	Two- way
Average	1.00	.87	1.00	1.01
Minimum	1.00	.74	1.00	1.00
Maximum	1.00	.95	1.00	1.02

meaning of coherency and the methods employed by the VAX 6200 project engineers to ensure coherency. We also describe our techniques for supporting recovery from all single-bit transient cache errors.

For this discussion, we divide the cache subsystem of the VAX 6200 into three sections. Figure 5 shows the three major subsystems in the VAX 6200 cache:

- The CVAX internal I-stream-only cache
- The 256KB I-and D-stream cache
- The 16-byte write buffer (a form of write-back cache)

CVAX I-stream-only Cache

The first cache, contained within the CVAX chip itself, is configured for I-stream-only operation. In that mode, the CVAX flushes the entire contents of the cache whenever a VAX REI instruction is executed. Motivated originally by the potential problems with instruction prefetch buffers, the VAX architecture defines rules for software to assure that writes to I-stream data produce predictable results.⁸ In all cases, if the rules are not followed, stale data may be read from the cache and cause unpredictable results.

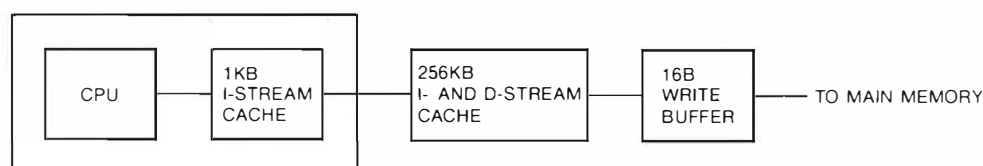


Figure 5 VAX 6200 Cache Subsystems

Second-level I- and D-stream 256KB Cache

The second-level cache is architecturally similar to caches used on most VAX systems. With a write-through design, the cache stores both I- and D-stream data. Coherency is maintained by monitoring all writes from other devices to main memory and invalidating cached locations that correspond to any of the monitored writes. The processor does not generate invalidates for its own writes to main memory since the cache is write-through; a write by the processor itself that hits in the cache immediately updates the appropriate location.

The VAX 6200 second-level cache coherency logic is shown in Figure 6. A duplicate tag store is located on the multiplexed XCI bus. This store contains a duplicate copy of the 4,096 cache tag entries, which are in the second-level cache located on the CDAL. The duplicate tag store tracks the primary tag store on allocates by monitoring XMI read transactions. Whenever an XMI memory space read is initiated, the CPU allocates the cache block that corresponds to the read address.

The duplicate tag store also monitors all XMI write transactions and performs a duplicate tag store look-up. If a hit occurs and the write was not from this CPU, then the duplicate tag location is invalidated. The address is then loaded into an eight-entry invalidate queue implemented in the XMI interface gate array. Cache invalidates are not performed in response to an individual CPU's own writes since the write-through second-level cache always contains the most recent data.

When an entry has been loaded into the invalidate queue, the CDAL interface logic arbitrates for the CDAL and invalidates the full 64-byte block in which the write address was located. The use of a duplicate tag store reduces CDAL traffic to only necessary invalidate transactions. After performing an invalidate, the XMI interface gate array checks for any additional invalidates that may have accumulated while the previous invalidate was being serviced. If another invalidate request exists, then it is serviced prior to release of the CDAL. This procedure ensures that invalidates are serviced as quickly as possible. The CVAX bus interface ensures that the invalidate logic is given an opportunity to use the CDAL between every CVAX bus operation.

Though occurring very infrequently, the XMI bus could issue writes quickly enough to over-

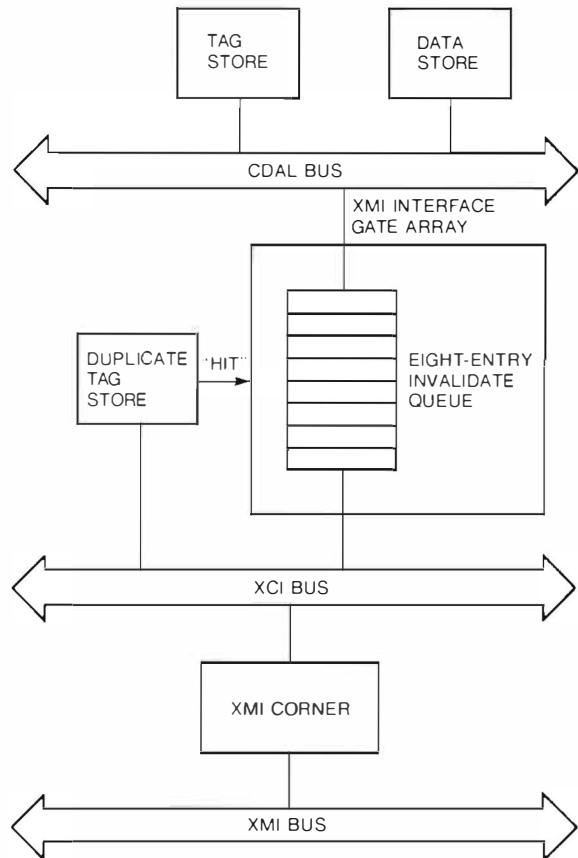


Figure 6 Second-level Cache Coherency Logic

flow the CPU's invalidate queue. Instead of adding significant complexity to the invalidate controller to suppress the generation of XMI write commands when the invalidate queue is at risk of overflowing, the overflow condition is handled as an exception condition. (This subject is discussed in the section Handling Second-level Cache Error Conditions.) For this alternative to be practical, we had to ensure that invalidate queue overflows would be very rare; we felt this was ensured by the depth of the invalidate queue (eight entries) and the optimized design of the invalidate controller.

The Write Buffer

A write buffer design offers the designer opportunities to break cache coherency rules. The VAX 6200 CPU follows several rules to maintain coherency. The VAX 6200 hardware automatically flushes the write buffer under the following conditions:

- In response to a write that misses the currently active write buffer. The current write buffer is

flushed while the new write is accepted by the alternate buffer, thus write ordering is maintained.

- Before an XMI I/O space read or write reference is performed. I/O references could result in the initiation of an I/O operation that may require the data from the write buffer.
- Before an interlock read or unlock write reference is performed. Interlock sequences are the primary means for synchronization between processors and must always force all outstanding writes to main memory.
- Before an interprocessor interrupt is performed. As with interlocks, interprocessor interrupts are used for synchronization between processors and must always force all outstanding writes to main memory.
- Before issuing an XMI read to a location that includes the data contained in the write buffer. The write buffer contents are flushed to main memory and then the XMI read is issued. Reads that miss the write buffer do not force a write buffer flush ("write buffer bypass").
- Following the assertion of the CVAX clear-write-buffer pin, the CPU flushes the write buffer to main memory. This form of write buffer flushing is primarily used to associate failed writes with a given process. If no association could be made, then the operating system would always have to crash the entire system on every failed write transaction.

Handling Second-level Cache Error Conditions

One of the major goals of the VAX 6200 design was to provide improved system reliability. One method we used was hardware-enforced soft failover in response to many error conditions, combined with efficient software recovery procedures. This method was used extensively when dealing with all types of second-level cache errors.

In general, the individual processors have the responsibility to recover from potential cache coherency failures. When errors occur that may leave the second-level cache incoherent, the VAX 6200 processor hardware automatically disables the cache. Disabling the cache ensures that the system can continue to run "safely," albeit at reduced performance. The processor then posts a "soft" error interrupt. The interrupt service rou-

tine responds by logging the error and then flushing and reenabling the cache.

The following error conditions cause the XCP hardware to disable the second-level cache. The errors are of two forms. The first two are error conditions that potentially result in a missed cache update on a write-through; the last three deal with conditions under which an invalidate is potentially missed:

- Subblock valid bit parity errors — The VAX 6200 CPU supports a doubly-redundant set of subblock valid bits. On a cache look-up, if the two corresponding valid bits do not match, then the hardware reports a parity error and forces a cache miss. If this error occurs on a write-through that should have hit in the cache, then the cache state is no longer consistent.
- Cache tag parity errors — The tag chips used on the VAX 6200 CPU support parity on the full tag address. As with valid bit errors, a tag parity error can result in a missed write-through.
- XMI inconsistent parity error — If the CPU detects an XMI cycle that has bad parity and that cycle is acknowledged by another processor, then the worst-case assumption is that the duplicate tag logic just missed a write transaction that should have resulted in an invalidate.
- Duplicate tag store parity error — As with the previous error, the processor has to assume the parity error resulted in a missed invalidate.
- Invalidate queue overflow — Again, this condition is similar to the one above except that this condition does not require a transient error in the system. Instead, an invalidate queue overflow is the result of a very rare combination of XMI writes that result in a queue backup and the potential loss of invalidates. The system responds to this condition just as it would for all other cache errors.

Actual System Performance Results

We were very interested in determining how well our simulation results matched real-world operation. We decided to focus on several key aspects of the system to bound the task of correlating simulation with the real world. Specifically, we planned to

- Confirm that the VAX 6200 CPU performs as expected relative to the MicroVAX 3600 systems. If the cache subsystem behaves as

expected, then the VAX 6200 performance should exceed that of the MicroVAX 3600 systems by the clock rate improvement minus the penalty for running the CVAX cache in I-stream-only mode.

- Confirm that the simulation traces adequately "stress" the memory interface such that extrapolation to real workload performance is valid. The percentage of the XMI bus consumed would be the basis for this comparison. This characteristic includes all the effects of references per instruction and miss rates and ultimately determines the performance of a multiprocessor machine.
- Confirm that the cache subsystem supports very effective utilization of multiple processors. VAX 6200 multistream throughput measurements form the basis of this verification.
- Compare the results from the simulation tests with similar workloads run on real machines.

Comparing VAX 6200 and MicroVAX 3600 Systems

Due to the similarities between the two systems, our first approach was to compare the performance of the VAX 6200 to the MicroVAX 3600 systems by running a set of 100 compute-intensive benchmarks. The VAX 6200 has a 12 percent cycle time advantage (90 ns to 80 ns), but it is somewhat handicapped by the I-stream-only limitation placed on the internal cache. Recall that our performance simulation indicated this penalty would average about 5 percent. (See Table 6.) On average then, we expected the VAX 6200 CPU to be about 7 percent faster than the MicroVAX 3600 CPU. The compute-intensive benchmarks basically confirmed this number; VAX 6200 performance averaged 6 percent faster than the MicroVAX 3600 CPU.

Multiprocessor Bus Bandwidth Utilization — Real and Simulated Workloads

We have run several forms of multiuser time-sharing workloads on the VAX 6200 system. These workloads include Digital's standard ALL-IN-1 workload, an order processing benchmark (Compu-Share), an electrical CAD workload, and a software development workload.⁹ In all cases, the average percentage of the XMI used per processor ranged from 3.75 to 5.0. Recall

that our simulation indicated that the percentage XMI consumed would be 6.27 percent. (See Table 9.)

Multistream Performance on Compute-intensive Benchmarks

It is beyond the scope of this paper to present the multiprocessor simulation data that was generated prior to design. That data indicated that the VAX 6200 system performance on compute-intensive benchmarks would be nearly linear when running from one to eight processors.

Tests to date have confirmed our high expectations. On compute-intensive workloads, a four-processor system consistently provides better than 3.95 times the throughput of the single-processor system (less than 2 percent degradation). Limited configuration testing on systems with up to eight processors indicates that compute-intensive workloads continue to perform very well. An eight-processor system performed at 7.75 times the single-processor (less than 5 percent degradation).

Fully Characterized Workloads

We also instrumented a VAX 6200 system to measure a number of processor characteristics, including bus utilization. We wanted to determine how much the real workload runs varied from the simulated runs. The test methodology was quite simple.

- Command files were created that executed a single benchmark. These individual benchmarks were designed to correspond with the simulation traces listed at the beginning of the Performance Simulation section.
- The Digital Command Language (DCL) command files were of the following form:

```
$
$ @flushcache ! initially flush the
    cache
$ @starthardwaresample ! start the
    measurement hardware
$ @getcputime ! get the initial CPU time
$
$ run benchmark
$
$ @getcputime ! get the final CPU time
$ @stophardwaresample ! stop the
    measurement hardware
$
```

- The measurement hardware consisted of two Tektronix DAS 9200 Logic Analyzers; one monitored the processor bus, and the other was attached to the XMI. The start-measurement command file simply referenced a specific XMI I/O space address on which the DAS 9200 analyzers would trigger and start taking measurements. Similarly, the stop-measurement command file would reference another XMI I/O space address that would cause the logic analyzers to stop acquiring data.

This technique made the measurement process simple and repeatable. The overhead of the command file was measured by running the command file with the "run benchmark" line removed. This overhead was then subtracted from the results obtained from benchmark runs. Run-to-run consistency was better than ± 10 percent.

The logic analyzers captured the data necessary to determine the total number of XMI read and write references that occurred during the execution of the command file. This data was used to calculate the total number of XMI cycles used by the processor. To derive the percentage of the XMI utilized, the total XMI cycles were reduced by the command file overhead, and the result was divided by the benchmark CPU time. This method ensures that the XMI percentage is not artificially low due to the inclusion of null time elapsed while the processor is waiting for I/O activities associated with the benchmark to complete. The results are shown in Table 11.

The data indicates that the simulation traces required significantly more XMI read bandwidth (on average more than double) than the similar actual benchmarks. This result is not unexpected, since the simulation runs were designed to simulate a worst-case timeshare workload. (This goal influenced the choice of 35,000 instructions for the cache flush interval.) The real workloads were run on standalone systems, and therefore the cache performance was expected to be higher. We are currently studying the effect of heavy timesharing in multiprocessor systems on cache performance. Initial results indicate that our simulation runs are still conservative.

The results for writes, which are unaffected by context switch rates, matched the actual benchmarks quite closely. The actual benchmarks required about 4 percent to 8 percent more bandwidth than the equivalent simulation trace. Combined read and write bandwidth require-

Table 11 Simulated versus Actual XMI Bus Utilization

	Simulated I-stream	Actual I-stream	Simulated/Actual Ratio
Average	0.84%	0.32%	2.6
Minimum	0.24%	0.17%	1.4
Maximum	1.65%	0.52%	3.2
	Simulated D-stream	Actual D-stream	Simulated/Actual Ratio
Average	1.63%	0.74%	2.2
Minimum	1.26%	0.26%	4.8
Maximum	2.10%	1.10%	1.9
	Simulated Writes	Actual Writes	Simulated/Actual Ratio
Average	4.46%	4.86%	0.92
Minimum	3.57%	3.84%	0.93
Maximum	5.97%	5.75%	1.04
	Simulated Overall	Actual Overall	Simulated/Actual Ratio
Average	6.09%	4.86%	1.25
Minimum	5.27%	3.84%	1.37
Maximum	7.25%	5.75%	1.26

ments indicated that the simulated traces used 25 percent more bandwidth than the actual workloads.

Conclusions and Future Work

The VAX 6200 design experience has demonstrated that trace-driven simulation is a powerful tool in the design of a multiprocessor bus interface. Because the designers were able to make informed trade-off decisions, the design met or exceeded all performance goals; and the reduced design complexity helped bring the system to market on schedule. It is a tribute to the team's appropriate control of complexity and to the rigorous verification process¹⁰ that the first-pass VAX 6200 CPU printed circuit design and XMI interface gate array are currently shipping in VAX 6200 systems. At Digital, this level of success is unprecedented for a machine of this complexity.

The continuing trend toward multiprocessing and faster processors will force increasing dependence on complex cache subsystems to deliver the desired system performance. It follows that minimizing the complexity of the cache subsystem will help support ever decreasing time-to-market schedules. Accurate cache simulation techniques will be required to select the implementation that meets the performance goals and is minimally complex.

Acknowledgments

The author would like to acknowledge the work of the following individuals who contributed significantly to the VAX 6200 CPU architecture and design: Brian Allison, Giau Dau, Ron Desharnais, Glenn Herdeg, Dave Ives, Bimal Sarecn, Simon Steely, and Doug Williams.

References

1. B. Allison, "The Architectural Definition Process of the VAX 6200 Family," *Digital Technical Journal* (August 1988, this issue): 19-27.
2. T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital Technical Journal* (August 1988, this issue): 95-108.
3. E. McLellan, G. Wolrich, and R. Yodlowski, "Development of the CVAX Floating Point Chip," *Digital Technical Journal* (August 1988): 109-120.
4. J. Winston, "The System Support Chip, a Multifunction Chip for CVAX Systems," *Digital Technical Journal* (August 1988, this issue): 121-128.
5. A. Smith, "Line (block) Size Choices for CPU Cache Memories," *IEEE Transactions on Computers*, vol. C-36, no. 9 (September 1987): 1063-1075.
6. D. Clark and J. Emer, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the 11th Annual Symposium on Computer Architecture* (June 1984).
7. J. Fu, J. Keller, and K. Haduch, "Aspects of VAX 8800 C-Box Design," *Digital Technical Journal* (February 1987): 41-51.
8. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1987).
9. B. Moses and K. DeGregory, "Performance Evaluation of the VAX 6200 Systems," *Digital Technical Journal* (August 1988, this issue): 64-78.
10. J. Basmaji, G. Garvey, M. Heydari, and A. Singer, "The Role of Computer-aided Engineering in the Design of the VAX 6200 System," *Digital Technical Journal* (August 1988, this issue): 47-56.

The Role of Computer-aided Engineering in the Design of the VAX 6200 System

The success of the VAX 6200 design is partly attributable to the development and implementation of a total verification plan. The goal of this plan was to shorten the total system design cycle; the approach was to perform sufficient verification to ensure that first-pass parts would boot and run VMS at speed. The team responsible for achieving the goal began implementing the verification process on availability of the first design specification. The team's efforts continued concurrent with those of the module design team. Milestones for the process reflect the verification team's top-down functional approach, proceeding from architectural-level verification through logic, timing, and system verification, and concluding with vector generation. Review and reporting methods established for the project ensured all functions were tested and verified.

This paper presents an overview of the computer-aided engineering (CAE) and CAE-based design verification test (DVT) approach to the development of the VAX 6200 system. Our intent is not to give a step-by-step description; therefore, few details of the implementation are given. The CAE/DVT Group developers believe that project-specific problems are generally best solved by project-specific solutions. Instead, we offer a broad overview of CAE which includes the engineering principles established for the VAX 6200 project and which we believe will be of use to those planning a task of similar scope.

A Brief VAX 6200 System Overview

No discussion of CAE or DVT methodologies can take place without a description of the task to which these methods are applied. For our purposes, the overall task was to engineer, prototype, debug, and release for manufacture the VAX 6200 mid-range computer system.

The VAX 6200 multiprocessor architecture is implemented with CMOS technology.^{1,2} The system is housed in a 156 by 79 by 76 cm cabinet, which contains a system bus backplane, two 6-slot VAXBI backplanes, a TK50 tape drive, space for future rack-mount devices, power supplies, and blowers.

The heart of the system is a new interconnect called the XMI. This interconnect was specifically designed to serve as the processor-to-memory interconnect in the VAX 6200 system and its derivatives. Optimizations of and trade-offs in the design of the XMI were made with that function foremost in mind. The key features of the interconnect are as follows.

- The pended bus design allows multiple transactions to be in progress at the same time; thus waste of bandwidth is minimized, for instance, during memory read accesses.
- The XMI implements the concept of commander nodes and responder nodes. A commander node initiates a bus transaction to which a responder node must respond.
- The XMI is a centralized arbitration interconnect. Arbitration logic, resident on the backplane, grants bus mastership according to a modified round-robin scheme. There is a higher priority responder round-robin queue and a lower priority commander round-robin queue.
- Bus width is 64 bits.

- Cycle time is 64 ns.
- The XMI supports reads of quadword, octaword, and hexword length, and writes of quadword and octaword length.
- Raw bandwidth is 125 megabytes (MB) per second.

The XMI supports three module types in the VAX 6200 system: the CPU module (KA62A), a 32MB memory array (MS62A), and an XMI-to-VAXBI adapter module set (DWMBA).

The CPU module is based on the CMOS VAX (CVAX) chip set, which includes a MicroVAX architecture microprocessor (CVAX), a floating point accelerator (CFPA) chip, and a system support chip (SSC). The module supports full VAX capabilities, excepting only PDP-11 compatibility mode. In addition to a two-way associative I-stream cache in the CVAX chip, the module contains a 256 kilobyte (KB) direct-mapped cache. Performance is approximately 2.8 times that of a VAX-11/780 processor.

The MS62A is a 32MB memory array module with an on-board controller. Modules may be interleaved up to eight ways to decrease latencies. Each module has an eight-deep command queue. The arrays are fully error-correction code (ECC) protected.

The DWMBA is an adapter module set which allows the 6200 system to access I/O devices on the VAXBI bus. The DWMBA/A module, which resides in a single XMI slot, is connected by cable to the DWMBA/B module, which resides in a single VAXBI slot. The DWMBA can support up to full VAXBI bandwidth of 13.3MB per second on write transactions and approximately 5.5MB per second on read transactions.

Figure 1 illustrates how these system elements interconnect in a two-processor system with two VAXBI channels.

Because the VAX 6200 system backplane has 14 slots, many system configurations are possible with differing numbers of processors, memory modules, and I/O channels.

In the sections following, we describe the engineering process employed in the design of these logic elements.

CAE Verification Challenges and Organizational Structure

The overriding goal of any CAE effort is always the same: to shorten the development time

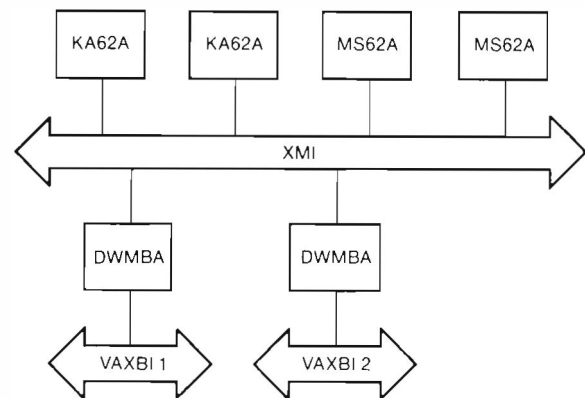


Figure 1 XMI Module Connections on a VAX 6220 System

needed to bring a product to market. The definition of CAE and the way engineers use CAE to accomplish this goal differs from project to project and even within a single project. Nevertheless, two principles are preeminent.

1. CAE should provide the tools, the methods, and perhaps most importantly, the discipline that together enhance an engineer's productivity without unduly restricting his or her creativity.
2. CAE should provide a continual check to ensure that the engineer's product meets the needs of the project in terms of both function and quality.

The role of the CAE/DVT Group on the VAX 6200 project was different from the traditional CAE role in one significant respect. The group's primary responsibility would not be the development of CAE tools and processes. Instead, its responsibility was the delivery of first-pass hardware that was functional at speed. Explicitly, our goal was to ensure that the system would boot the operating system (VMS) and run software the first time the system was powered up. The only tools and processes developed were those specifically necessary to fulfill that goal.

The project team felt that objective simulation and verification of the hardware and its performance by the CAE/DVT Group would (1) enhance the chances of first-pass functionality, and (2) reduce the overall design cycle by paralleling the CAE and the design efforts. Consequently, the CAE engineers were active contributors to the architecture and participated in

choosing alternatives, effecting compromises, and implementing details of the design. The CAE Group was responsible for the correctness and quality of the designs and not just for the delivery of tools to accomplish that correctness. To achieve this goal, tasks traditionally performed using DVT methods would be accomplished using CAE methodology.

CAE Tasks

Given the charter described above, the CAE/DVT Group outlined the following tasks:

- Select a tool suite
- Create a process for the CAE effort
- Maintain the databases
- Construct a CAE environment (models and computes)
- Generate test cases to run against the environment
- Isolate and report bugs
- Verify the hardware
- Generate test vectors for outside vendors
- Generate test vectors for manufacturing
- Fault grade the test vectors
- Define exit criteria for committal of design to hardware
- Enforce compliance with exit criteria

Though the list is long and has some interesting tasks, two items constituted the largest portion of the work: *generation of test cases to run against the environment*, and *verification of the hardware*.

The generation of test cases is the most time-consuming, least glamorous, and most often overlooked task; yet the test cases are the single most important piece of a superior CAE effort. A successful specification of the test cases (the DVT specification) to be run against a CAE environment requires a lengthy period of development. The development time for the KA62A, MS62A, and DWMBA DVT specifications was approximately 6 man-months each. Moreover, the specification is not static and must be kept current with the evolving design.

The DVT specification must begin as early as specification of the hardware functionality begins. Working the two specifications in parallel ensures functional verification of the system. Further, the DVT specification should be treated with the same formality as the hardware specification; that is, it should be reviewed, and all reviewers must agree upon its completeness. By formalizing the specification review, project members are in effect establishing its value to the project. The DVT specification defines what is to be simulated; therefore, superior design tools and modeling cannot substitute for the assurance of design accuracy that the specification affords.

As to the verification of the hardware, the responsibility of the CAE team was to ensure bug-free and operable component, module, and system designs. Team members ran the simulations, isolated the bugs, and ensured designs were corrected by the design team. Simulations were not done exclusively by the CAE team, however. The environment was available equally to all design team members. To the extent that each team felt was appropriate, designers initially debugged their designs before passing them to the CAE team for more formal debug. In this way, obvious bugs were found more quickly. Design developers did excellent work in this regard and greatly eased the burden on the CAE team.

Further discussion of the VAX 6200 hardware verification is presented in the section Verification Milestones.

Modeling Approach

Hardware verification done in software is by nature a slow process. The major factor contributing to the slowness of the verification is the size of the design. The size is not simply the number of logic elements in the design, but the collective size of the models of each of the elements in the logic network.

We used two types of models for the VAX 6200 project, behavioral and structural (or gate level). Behavioral models, in general, were more abstract and efficient in terms of increasing overall simulation performance as compared to detailed structural models.

Behavioral models of many of the components used in the system were generated early in the design cycle. As the design progressed and detailed logic schematics became available, however, the behavioral models, in most cases, gave way to detailed structural models. The exception

was the behavioral models of the CVAX chip set. These detailed models were used throughout the verification process. Given the size and complexity of these components, simulation with structural models, for all practical purposes, was impossible.

In general, our objective throughout the verification process was to ensure accuracy and not speed. The slow speed of the more accurate models was addressed by applying more compute power to the task at hand.

CAE Staffing and Resources

The CAE group was divided into small teams, each responsible for the verification of a VAX 6200 subsystem. The size of the teams varied. The KA62A gate array and module team had four CAE engineers. Four CAE engineers worked on the DWMBAs gate arrays and modules. The MS62A gate array and module was assigned one CAE engineer. As it turned out, these numbers represented nearly a one to one ratio with the hardware designers. As senior, experienced engineers, team project leaders were responsible for the overall coherence of the DVT plan and its quality, and were responsible as well for tracking and resolving problems.

Each team included a diagnostic engineer who was also working on design verification test. This arrangement provided the diagnostic engineers early training and also facilitated testing. Moreover, diagnostic engineers were in a position to easily evolve some of the DVT tests into self-tests and ROM-based diagnostics for the VAX 6200 product.

The educational background of the CAE team was a mix of electrical engineers, computer engineers, and software engineers. Their levels of experience varied from new college hires to those with 10 or 15 years of work experience. The level of relevant hardware experience in this group is indicative of the group's tasks, as compared with other CAE groups that are more involved in tools generation.

Our computer resources consisted of a cluster of eight CPUs, including one VAX 8800 system, one VAX 8650 system, and six VAX-11/780 systems. All four modules (KA62A, DWMBAs/A, DWMBAs/B, and MS62A) and their associated gate arrays were verified throughout most of the project on this cluster. During final regression testing of each module, in which the full set of DVT tests was run against the design,

an additional cluster of eight VAX 8800 systems was used.

Verification Milestones

Key milestones were established for the verification team throughout the VAX 6200 design verification process. In December 1985, we began with the first XMI interconnect verification; we proceeded to performance evaluation, logic verification, timing verification, system verification, and vector generation.

These milestones were derived as part of our functional top-down verification approach. We selected this approach based on our determination that if a function works correctly, then all of its component logic must be working correctly.

We therefore chose to model our different design objects in the largest reasonable forms and then functionally test these models. Every step naturally lead into the next task of the system design. This approach was later extended to the system as a whole; the system simulation combined the logic and ran code to exercise the entire system.

Architectural Verification

At the architectural level, the simulations focused on the verification of the new system interconnect, the XMI. As noted earlier, this interconnect, specifically developed for the VAX 6200 system, is a memory interconnect bus with a new arbitration scheme and a defined bus interface protocol. Both the arbitration and the protocol are implemented in CMOS semicustom technology.

Once the design for the bus protocol and arbitration was established in a specification form, we immediately transformed the specification into high-level behavioral models: the arbiter chip model, and an XMI commander transactor model. The behavioral arbiter model represented a generic, round-robin arbitration scheme; the commander model represented a generic XMI commander design. The commander model contained a flexible user interface to allow the specification of any desired well- or ill-formed transaction to be generated on the bus. Further, the commander transactor model was designed to selectively self-check for any protocol violations.

The two models were the basis for all XMI design verification. This first level of verification provided feedback to the architecture team quickly and answered questions about the inter-

face protocol and arbitration scheme. As a result, the arbitration was enhanced and the protocol was refined to satisfy the design goals. Specifically, a few new signals were added, and the arbitration was changed from true round-robin to a modified round-robin.

At the next level of architectural verification, we modeled an XMI responder node and incorporated this model into the simulation environment. The team developed a behavioral XMI memory model and completed a high-level system model. This model was still totally behavioral and represented a system with generic XMI commanders and responders.

Two pieces of test code were generated and verified on that model. The environment modeled a fully loaded XMI interconnect. The first piece of test code was structured in such a way that every node on the XMI generated its own traffic. Commanders generated all possible commander sequences, and responders generated all possible responder sequences. The goal of this first test code was to ensure that the protocol was sound. By protocol soundness, we mean that commanders and responders can coexist on the XMI and can generate traffic sequences without loss of data. The results of this verification gave the team sufficient confidence in the protocol to allow the design of the XMI interface components to proceed.

The second piece of test code was verified on the same environment. Every commander generated the same sequence of traffic on the XMI. The goal of this test was to verify arbitration fairness and to guarantee that all XMI nodes got their fair share of the XMI. The absence of phenomena such as lockouts was also verified.

This architectural verification proved to be a tremendously valuable exercise. First, feedback to the architecture team was accomplished quickly. Second, this architectural verification for the VAX 6200 project established design verification tools that can be used for all future XMI designs.

In time, the behavioral model of the arbiter was replaced with a structural model derived from the chip design database. We enhanced the accuracy of the behavioral models of the XMI commander and memory by incorporating structural models of the XMI interface components once their gate-level designs were complete. These tools are now in use throughout the corporation by numerous XMI design teams.

Clearly, an architectural verification that concentrates on a new bus leaves out many other areas of architectural interest. A severe restriction of the scope of the VAX 6200 system's architectural verification was deemed necessary because of the lack of schedule time and because of the immaturity of the art. Nevertheless, architectural verification is a key area where much work should be done for the development of the next system.

Performance Evaluation

The next verification task was performance evaluation. Again, work was concentrated into two well-defined areas, that is, the bandwidth performance of the XMI, and the processor performance in the multiprocessing environment.

A model of the CVAX processor was obtained from the Semiconductor Engineering Group design team. We enhanced this model to include an XMI interface with a memory port. The stimulus for this model proved difficult to generate because multiprocessing benchmark traces were not available. The traffic patterns had to be deduced from single-stream benchmark traces and extrapolated for VAX 6200 symmetric multiprocessing.

We ran several benchmarks. We then used the results to make decisions about the appropriate trade-offs in the area of the processor cache and write buffer algorithms. These trade-offs dealt specifically with cache and write buffer depth versus performance gained.

Other tools were created to decompose XMI traffic into histograms and to generate reports on bus bandwidth for the different types of traffic. Eventually, XMI memory design latency targets were incorporated into the XMI behavioral memory model. These system performance simulations were used to establish such design criteria as the memory controller input command queue depth and the command queue processing algorithm.

Logic Verification

The next major task was logic verification. The main objective of module verification was to ensure that the implementation conformed to all design goals documented in the system specification. In other words, the goal was not to verify what the design was, but what the design was supposed to be.

Members of the CAE team were assigned to each design object; each member would work in

a team with the designers. The verification teams required complete and coherent specifications for each design object. These specifications had to be sufficiently complete to support both design implementation and logic verification. Moreover, all functions had to be documented in a specification. This documentation served two purposes: (1) to ensure that the function received the proper attention during the verification phase, and (2) to give the responsible CAE engineer the information needed to understand the functions without referring to logic schematics or meeting with the designer.

With the functional specification as the foundation, team members generated a verification working document for every design object. This DVT specification, as mentioned earlier, guided the verification work and constituted the primary hardware-submittal exit criteria.

The logic verification was grouped into three categories:

- Basic functional verification. Basic functional tests exercised each function as a standalone piece of the design. This testing isolated obvious bugs.
- Interaction sensitivities. Interaction sensitivity exercised the design as a whole, making sure that functions could interact with each other and could occur in series without cumulative fault mechanisms. Testing of function interaction included any boundary conditions, margin testing, and back-pressure on different key points in the design.
- Error handling. Error handling verification tested that portion of the design created specifically for error detection and recovery mechanisms.

Timing Verification

Timing verification was performed separately upon all key components in the VAX 6200 system. All of this work was performed by applying functional patterns to timing models for each of the module gate arrays and the XMI arbiter logic. This work was done using AUTODLY, an internal Digital tool.

The XMI components were tested first. Testing consisted of applying all possible XMI bus cycles against this logic while allowing the timing verifier to analyze the logic for any timing paths

with problems. A number of problems were found and resolved as a result of this testing.

The timing verification of the module gate arrays was performed in a similar fashion. Patterns of functions were extracted from logic verification and then applied to the standalone chip timing models. As each pattern was applied, the timing verifier would run a complete check of the gate array and generate a list of violations. These violations would then be checked by the designer. If they were valid, logic changes would be made. The reason that just the gate arrays were verified, and not their complete modules, was that each module contained some logic for which no structural model existed (for example, the CVAX chip set on the KA62A module). The lack of a complete module-level timing verification model was rectified by requiring the module design team to thoroughly analyze its module. This approach was possible only because of the highly bus structured nature of our technology.

System Verification

Once every design object met its exit criteria and satisfied the specified testing, the next milestone was the start of system simulation. Our task was to verify the actual design in a system environment. We constructed a model consisting of multiple processors, memories, and I/O modules. This model contained structural representations of the actual designs wherever possible. Where there were multiples of a design object in the system simulation environment, one instantiated copy of the model would be the detailed (and slow to simulate) structural model; the other instantiations were the faster yet less accurate behavioral models.

In addition to actual design objects in this system model, we included different types of transactor and traffic generators on both the XMI and the VAXBI buses.

The stimulus for this environment had to be specific enough to ensure that every type of traffic pattern was generated during simulation. The stimulus attempted to stimulate every node and function concurrently. In a system simulation in which the simulation rate is so slow, as much as possible must be achieved in every single simulated clock tick.

Key to making a system simulation successful is to start the simulation only after the constituent pieces of the system have been very thoroughly

verified in isolation. Given the complexity of the system model and its slow running rate, finding simple design bugs at this stage is a waste of schedule time. Instead, the model should identify the system interaction problems and assure developers that the base logic verification was thorough.

Several logic problems were found during our system simulation dealing with complex interactions, some after a few microseconds of simulation. If undetected, these problems would have seriously impeded progress toward our goal of providing functional first-pass hardware.

Vector Generation

The start of system simulation takes place, by definition, near the end of the logic verification process. At about that time, we began to prepare for submittal of the designs for fabrication. Therefore, in parallel with system simulation, test pattern generation was started.

Test vectors were needed at this time, primarily to test chips coming off the fabrication line. Therefore we generated test vectors for the very large channel-less arrays contained on each of our modules. The basic criterion for approval was attainment of 99 percent internal node toggle coverage of the gate array logic. In addition to the 99 percent internal node toggle criteria, we also included the much more stringent criteria of 95 percent stuck-at coverage as measured by a fault grading mechanism. The methods used to determine coverage are discussed in the section Problem Reporting and Resolution.

The vectors were extracted from a strategic subset of our functional DVT simulation and graded on a hardware accelerator/fault evaluator.

We set a goal that the vector count should not exceed the chip's gate count; that is, a chip with 25K gates should have no more than 25K vectors to exercise its logic. The vectoring process, including extraction, grading, and complementing, took an average of one month per gate array.

As is true of architectural verification, vector generation is an area where work remains to be done. If we had been able to include some testability features in these very dense chips, we could have saved this month of schedule time.

Follow-through

Even beyond the prototyping phase, the simulation database was maintained and updated to

reflect any changes in the design as a result of hardware debug. The purpose of this on-line soft representation of the design was twofold. First, the representation would aid in the isolation of any problems discovered in the lab. Second, the database could be used to investigate any suspicious problem areas that could not easily be triggered in the hardware.

Review and Reporting Methods

Throughout the design verification process, a means to ensure coverage was established for each phase. At the project outset, DVT specification coverage of functions was assured by several levels of team review. As the simulations progressed, the project leaders were given the responsibility of ensuring bugs were consistently reported and corrected. Vector extraction and grading of our gate arrays provided a strong measure of the completeness of the verification of these chips. Additionally, the internal controllers to the gate arrays were measured for complete state and product term coverage. Lastly, before being released for manufacture, the design was checked against our own exit criteria to ensure that the verification was complete.

This section presents details of these methods and tools for ensuring all functions were tested and verified.

Functional Coverage

The VAX 6200 project team chose the functional verification approach to verify all VAX 6200 designs. One problem with this approach is that there is no method of measuring functional coverage. Since all verification is based upon the DVT specification, functional coverage will be a reflection of the completeness of this document. Therefore, the DVT specification becomes the vehicle by which the functional coverage of the verification is to be measured. This specification must be made as comprehensive as possible. Therefore, the specification underwent many levels of review by a wide audience, including the entire design team.

Problem Reporting and Resolution

Another means used to ensure coverage was the problem-resolution and bug-reporting mechanism. Every design verification team project leader was responsible for tracking bugs in

the designs and ensuring these bugs were corrected and the correction was verified. Communication for this tracking was through VAX NOTES conferences.

For each design verification team, two conferences were created. The first was for bug reporting and bug-fix resolution. Only verification team members could write notes in the bug conference. Every entry indicated the date, model revision levels, test case number, failure symptom, and any assessment of the problem. Replies to each entry were entered, either by the project leader or the CAE team member responsible for the failing test, to indicate when the bug was verified as being fixed and the model revision levels at the time of verification. If the problem remained unresolved, the reply would indicate any action taken or patches made.

The NOTES conference review ensured that all bugs were given the proper attention and visibility.

The second conference was informational. Using this conference, engineers could learn about key aspects of the design as the verification progressed. For example, they could obtain information on undocumented features on which certain verification tests were based.

Fault Grading

Another process, which was implemented to measure functional coverage of the component patterns, was the fault-grading mechanism. In this approach, all component patterns for the large compacted arrays were generated at the functional level. The simulation environment for pattern capture was the same one used for functional verification. The stimulus generated was driven by high-level functions. The test patterns were captured at the chip's boundaries while the chip was being exercised on the module.

Traditionally, component patterns are generated by simulating the chip standalone and driving hand-crafted stimulus through the chip simulation. Due to test overlap, the approach taken by the VAX 6200 team did not ensure the optimum number of patterns for the maximum stuck-at coverage. However, the approach proved to be very beneficial. Ranging from 20K to 50K patterns for each gate array, the patterns were generated in the very short time of approximately one month. In reaching our goal of 95 percent fault coverage with these test patterns, additional

areas of logic were found that had not previously been tested. This additional logic yielded additional bugs.

The fault grading process also provided an additional degree of confidence in the coverage of the functional verification test cases. The 95 percent fault coverage goal was achieved with patterns derived from a subset of those test cases. It should be mentioned that the hardware fault evaluator was used extensively during this phase of the project and proved to be an irreplaceable tool.

State Machine Coverage

Tools were developed that would analyze traces generated from the internal gate-array controllers and sequencers. Traces were collected while the functional tests were being simulated and verified. All traces were later analyzed, and coverage was ensured for every state and product term. This mechanism was put in place and automated, so that after each regression, coverage could be rechecked.

After every regression run of all test cases, the results were analyzed to ensure that no product terms or states were missed as a result of test modification or bug fix. Additional test cases were generated to find specific and hard-to-activate conditions.

Exit Criteria

Before a design is sent to manufacturing, the design must meet the exit criteria. These criteria are as follows:

- All the specified test cases have been generated and have run bug-free against the latest design.
- The system simulation has run bug-free for two continuous weeks.

In other words, if bugs still exist in the design, the design is not yet ready for manufacture.

As judged by the nearly bug-free condition of the implemented hardware, these design-completion criteria and coverage metrics were appropriate for the VAX 6200 development effort.

The VAX 6200 project's tremendous success has established the process for future systems verification and for engineering quality measurement.

Results Attained

The development cycle for the VAX 6200 system was quite short, and therefore the need to produce functional first-pass hardware was very strong. The first XMI specification was released in December 1985. Eight months later, all of the XMI parts had been designed and simulated and were being manufactured. Two months later, the parts were up and running.

During this time, specifications were released for the KA62A, MS62A, and DWMBAs, and logic design was begun. Concurrently, test specification and test generation also began. In the late summer of 1986, all logic design was completed, and verification began. Two to three months after design completion, verification was completed for each module. With a complete and verified design, one month was used to generate all gate-array test vectors and then submit the gate arrays for manufacture.

In February 1987 — 14 months after the first complete XMI specification — the DWMBAs were manufactured, powered on, and run with first-pass hardware. One month later, the KA62As were powered on and running. Two weeks later, with functional MS62As, the first VAX 6200 system was powered on. Two weeks after that, on April 1, the first VAX 6210 system booted VMS with all first-pass functional parts.

Although a few bugs were later to be found and fixed, the goal of using simulation to generate hardware that works at speed the first time was attained. In fact, many of those original parts are being shipped with the VAX 6200 systems today.

Opportunities for Improvement

Although our verification process proved to be quite successful, we plan to make a few changes in this process for future projects.

Architectural verification, in so far as that means an effort to discover system-level inadequacies or bottlenecks, is in its infancy. We consider this a wide open area where much can be accomplished.

As module designs call for increases in speed, timing verification and signal integrity verification will make a much larger contribution to the total verification effort. Although the XMI interconnect was verified to all circuit, signal, and timing specifications, signal integrity was not emphasized to the same degree in the modules themselves. Although no significant problems

arose, we became strongly aware that future generations of hardware will be much more dependent on the type of verification used for the XMI. Although timing verification was performed on all gate arrays on the VAX 6200 system, this verification. In the future, we feel it is important to perform timing verification on the design during early development. Thus we can identify and solve the timing problems before they become too entrenched in the design to be fixed easily.

Since the wire delays for gate arrays can only be estimated until gate layout has taken place, all verification must be repeated once the actual timing numbers are returned. Additionally, floor planning of the gate array can have a significant effect on the performance and specific wire delays. On the VAX 6200 project, the layout and final wire delay calculations were performed by our gate array vendor and then sent back to us for reverification. These steps can take quite a long time in the design cycle of a gate array. To reduce the wait for real wire delays, we plan to perform all floor planning and preliminary layout operations at the design site. Additionally, this will allow us much more input to the floor plan and layout.

Summary

The success of the VAX 6200 verification effort can be attributed mainly to the decision to begin verification at the same time as the design and to continue verification and design as parallel efforts. This decision was implemented by assembling verification teams at the same time design teams were being built.

Verification was performed during each stage of development — from initial concept to system integration. The architectural verification confirmed the XMI architecture and arbitration algorithms. Performance verification helped define the processor and memory architectures and ensured that these architectures could take full advantage of the new XMI. The logic of all XMI modules, their gate arrays, and the XMI arbitration logic was verified against their specifications, not against the designs themselves. Lastly, the entire VAX 6200 system was simulated in a multiprocessing environment, proving that the different component modules could function together as a system. Verification from system architecture to gate arrays, modules, and then

back to a complete system again, throughout the life of the project was the only way to assure the main verification goal — first-pass, functional hardware.

During logic verification, attempts were made to perform verification using the smallest detail, while still keeping the scope of the logic under test large enough to allow system-level testing. By performing all testing at these much higher levels, a greater number of functions and more global functions can be tested at one time. The only drawback to testing at this level is simulation speed. The trade-off of speed for accuracy is a good one, for without accuracy the costly alternative is to design and manufacture multiple passes of hardware.

In conclusion, the most important outcome of our verification effort was a management philosophy that, in the end, verification is as important as logic design. With this understanding, verification criteria now determine when and whether

designs are to be released for manufacture. To make this work successfully, the necessary resources must be allocated for the verification effort. Furthermore, project teams must develop and follow through with complete verification strategies. These strategies focus on verification as a part of the total design process rather than as a process that takes place after designs are complete. The VAX 6200 project was proof that this philosophy can be made to work.

References

1. B. Allison, "An Overview of the VAX 6200 Family of Systems," *Digital Technical Journal* (August 1988, this issue):10-18 .
2. T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital Technical Journal* (August 1988, this issue): 95-108.

VMS Symmetric Multiprocessing

The symmetric multiprocessing features of VMS version 5.0 effectively utilize the greater computing power of Digital's multiple CPU systems. Key to the SMP design is an innovative mechanism, called a spinlock, that provides a high degree of parallelism for kernel-mode code. Where formerly VMS software used interrupt priority levels (IPLs) to synchronize processes, VMS now uses spinlocks. Because each VMS resource can be protected by a spinlock, this design provides more synchronization levels than could IPLs alone. Spinlock granularity directly affects system performance.

This paper describes the major features of symmetrical multiprocessing (SMP) in the VAX/VMS operating system. These enhancements are included in VAX/VMS version 5.0. Although it is impossible to present details of every aspect of the SMP design in these few pages, this paper provides an overview of the key mechanisms developed for VMS SMP.

Technology Developments

Over the last several years advances in computer technology, especially in VLSI, have yielded greater computing power in increasingly smaller packages. VLSI CPU chips have made possible multi-CPU, single-board computers. These multiple CPU systems are having an increasing impact on the general-purpose computing environment. The net result is that recent technology trends have redirected the challenge of building multiprocessing systems from the hardware engineers to the systems software engineers. Systems software engineers must now design effective ways to utilize systems with six, eight, or even more CPUs.

VAX Hardware Features Required by the VMS Operating System

The VMS SMP design requires that certain fundamental features be implemented in VAX multiprocessing hardware. These features are as follows:

- The ability to share common memory among all CPUs in the system

This shared memory allows all CPUs to execute a single copy of the operating system and to

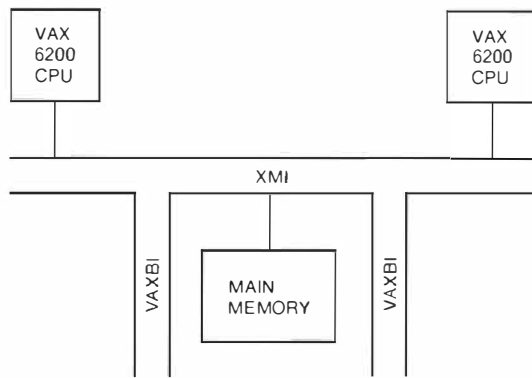
share state information that provides load balancing capabilities.

- An interprocessor interrupt capability that enables one CPU to interrupt all other CPUs or a single CPU
- The set of interlocked instructions (BBSSI, BBCCI, ADAWI, INSQxI, and REMQxI), which are part of the VAX architecture and thus present in every VAX system
- Cache coherency maintained by the hardware, without software assistance
- One CPU, known as the primary CPU, that must have access to all I/O, console subsystem, and timekeeping hardware

With these hardware features, VMS can provide symmetric multiprocessing support for any VAX system. All code executing in user, supervisor, or executive mode can execute on any CPU without restriction. Most (if not all) kernel-mode code can execute on any CPU without restriction. The only restricted code is that small amount of kernel-mode code that requires access to the time-of-day internal processor register or to the console terminal and the console block storage device.

The SMP design has no requirement regarding the system topology or interconnect joining the multiple processors. It supports systems implemented by means of a single bus architecture, such as the VAXBI bus, as easily as systems that use a cross-bar connection.

Therefore, the VMS SMP design is flexible enough to support current VAX systems and



KEY:

XMI — SYSTEM-TO-MEMORY INTERCONNECT

VAXB1 — BACKPLANE INTERCONNECT (FOR I/O)

Figure 1 VAX 6200 System Block Diagram

future VAX systems that take advantage of advancing technologies and architectures.

New Multiprocessing Hardware

The design of recent VAX systems, such as the VAX 8800 and the VAX 6200 series of computers, offers an elegantly simple, symmetric hardware configuration. Central to the design of these systems are two new bus architectures — the XMI bus and the VAXB1 bus (Figure 1). The VAXB1 architecture provides a protocol that allows (1) multiple processors to issue device requests, and (2) operating system software to specify which processors a device controller will interrupt.

The symmetry of these I/O subsystems presented a new challenge to the VMS SMP designers: to provide an I/O database design that would make possible simultaneous execution of interrupt handlers, thus taking advantage of these new hardware features.

The Development of VMS SMP

Critical to SMP was a new method, used throughout the VMS kernel, to synchronize multiple processors. One possible SMP design would have been to create a single lock for kernel-mode operations and allow any processor to acquire that lock. However, the VMS engineers believed that such a design would not have provided sufficient parallelism to achieve good system throughput for systems with more than a few processors. This single-lock method would have

been a non-scalable solution; if more CPUs were added to the system, system performance would not increase due to blocking for the single lock.

A more ambitious yet costly design was to provide a high degree of parallelism for kernel-mode code. With this kind of parallelism, many processors are allowed to execute different portions of the executive at the same time. For example, a process adding a system-wide logical name should be able to execute on one CPU while another CPU handles a device interrupt for completion of a disk I/O request, etc. This design would require creation of numerous locks and careful design of the interactions between the critical regions that use those locks. This design approach was the one finally chosen by the VMS engineers, and is discussed in the following sections.

Synchronization in VMS:

Raising IPL, Mutexes, and Spinlocks

The original VMS version 1.0 design used two types of synchronization: (1) raising interrupt priority level (IPL) and (2) mutual exclusion semaphores (mutexes). The VAX architecture provides 31 IPLs; 1 through 15 are dedicated for use by software, and 16 through 31 are reserved for hardware. (IPL 0 is not really an IPL but rather the level at which user, supervisor, and executive mode programs execute.) VMS blocked different types of system events by raising IPL to or above the level at which that event occurred. For example, process rescheduling was done by means of an IPL 3 software interrupt. Code threads that modified a process's context always executed at IPL 3 (or higher) to prevent a reschedule. Another example is the manipulation of device controller registers. These registers were always manipulated at the device's hardware interrupt level; thus other system activity of a lesser importance was blocked out while the time-critical code path was executed.

The second synchronization method, mutexes, was used to lock purely software constructs, such as global section descriptors. Mutexes provided a mechanism for defining many locks without assigning a unique software IPL to each lock. A mutex was acquired by the operating system on behalf of a process and was considered "owned" by that process. Rescheduling could occur while a process "owned" a mutex; however, process deletion could not occur. Lock requests made by

a process of higher priority for an already owned mutex were handled by placing the requesting process into a wait state, thus avoiding deadlocks.

In a multiprocessing system, each VAX CPU has its own interrupt priority level, independent of the others. Thus raising IPL would synchronize on a single CPU but not across the entire system. IPLs, then, could not be used to synchronize all CPUs. Neither were mutexes a viable solution, since they could only be used within process context and at low IPLs. Therefore, the SMP team created a new VMS mechanism that they termed a "spinlock." Anywhere VMS code had previously synchronized by raising IPL, the code would now acquire a spinlock; wherever VMS code had lowered IPL, it would now release a spinlock. Use of mutexes remained unchanged save that the code to acquire and release mutexes was protected by a spinlock.

The design for spinlocks included a number of critical concepts. First, a spinlock is "owned" by a CPU, not by a process (as mutexes are). Second, each spinlock is acquired and released at a particular IPL that is associated with the spinlock. Raising IPL when a spinlock is acquired prevents other activities from interrupting time-critical code. Third, CPUs "spin-wait" when blocked from obtaining a spinlock resource held by another CPU, since spinlocks are only assigned to time-critical resources that cannot be locked for long periods of time. Lastly, the design of spinlocks includes a mechanism for deadlock prevention or detection since the debugging of "hung" systems is too costly. Therefore, each spinlock is assigned a rank. Because spinlocks must be acquired in order of rank, deadlocks are thus prevented. Further, a debugging aid was built into the spinlock design. A part of each spinlock data structure is set aside to hold the last eight program counters (PCs) that acquired or released each spinlock. When enabled, these consistency checks proved invaluable in determining interactions between different components in the VMS executive, such as memory management and scheduling.

The VMS engineers implemented routines for acquiring and releasing spinlocks rather than scatter in-line code through the VMS kernel. The first step in acquiring a spinlock is to synchronize the local processor by raising to the IPL of the spinlock, just as if it were a uniprocessor system. The actual locking of a spinlock is accomplished

with an interlocked test-and-set memory operation, the BBSSI (Branch on Bit Set and Set Interlocked) instruction. The spinlock interlock bit is contained in a separate byte within the spinlock structure. Unlocking a spinlock is done with the inverse BBCCI (Branch on Bit Clear and Clear Interlocked) instruction. These interlocked operations are atomic memory transactions across all processors in a VAX multiprocessor configuration. Furthermore, since memory is common to all processors, the interlocked memory test-and-set operations provide a sufficient method of extending synchronization to all processors within a multiprocessor system.

The use of multiple IPLs as a synchronization method in VMS provides the capability to schedule events in a prioritized fashion. The inclusion of IPLs in the spinlock structure allows the SMP synchronization mechanism to appear as an added dimension to IPLs. Moreover, this SMP mechanism preserves the ability to schedule events in a prioritized manner.

For uniprocessor systems, the SMP design also includes the ability to optimize the routines that acquire and release spinlocks. For example, on a single CPU system, the spinlock acquire-and-release routines are never called. Instead, only a move-to-processor register (MTPR) instruction is executed, thus raising IPL. System performance of a single CPU has been measured as only a tiny percentage less than VMS version 4 performance.

Mutex synchronization is still the second synchronization method used in VMS. In the SMP design, mutexes are used for locks that are held for long periods of time and for situations in which the IPL has to be lowered. Mutexes are still owned by processes, not by CPUs, under the SMP design.

Spinlock Granularity, Device Locks

One aspect of the SMP design that directly affects system performance is the granularity of the spinlocks. A coarse granularity (fewer spinlocks) is easy to implement and debug; however, a coarse granularity provides fewer synchronization points, and thus processors are blocked for longer periods. A finer granularity (more spinlocks) provides more parallelism and thus shorter blocking times; however, a fine granularity is much more complicated to design and implement, and requires more synchronization points. An important concept to remember is that, while the system is in a noncontending

situation, a synchronization point only adds unnecessary overhead. That is, if there is never any possibility of processors contending for the same resource, then synchronization is not required. Therefore, the SMP team decided that a manageable number of spinlocks for the initial design was no more than 32. The SMP design provides designers the ability to create a finer granularity of locks in future releases of VMS as performance measurements identify time-critical resources.

As the SMP development evolved, it became clear that a finer granularity of spinlocks for the I/O subsystem would be easy to implement. With multiple VAXBI buses, multiple CPUs could handle different device interrupts simultaneously. This further improved the parallelism of the system and resulted in a new characteristic for spinlocks: dynamic versus static spinlocks. A static spinlock protects those resources common to all VAX/VMS systems. Therefore, static spinlocks are assembled into the VMS source code. Dynamic spinlocks synchronize device-specific code and so are created at boot time, depending upon the I/O configuration of the particular VAX system. Thus the number of dynamic spinlocks varies from system to system, whereas the number of static spinlocks is consistent across all systems. The dynamic spinlocks used to lock particular devices were named "devicelocks" to differentiate them from static spinlocks. A devicelock is used wherever device-specific code previously raised IPL to a device's IPL to block interrupts.

Identifying Resources Requiring Spinlocks

One of the first SMP development tasks was to identify each VMS resource that needed synchronization and then determine the proper locking mechanism — spinlock, mutex, interlocked queue, etc. Once this work was complete, the added dimension provided by spinlocks allowed multiple resources to be protected by a single IPL. For example, IPL 8 (SYNCH) had protected the following resources: memory management, scheduling, the I/O database, the file system, and the timer queue. By adding a new dimension, namely spinlocks, each of these resources could be protected by a different spinlock but share the same IPL. Therefore, in a multiprocessor configuration, it was now possible to run more than one processor at the same IPL. However, the processors must be executing different critical regions

of code. The spinlock design, therefore, has the advantage of providing more synchronization levels than could be provided by IPLs alone. Hence, the granularity of spinlocks can be much finer than that allowed by software IPLs alone. This finer granularity in turn provides more concurrency of execution in the VMS kernel.

For example, IPL SYNCH had protected a large number of resources and thus would be a good candidate for a finer granularity of spinlocks. Where VMS code had previously raised IPL to SYNCH, the SMP team had to determine which spinlocks had to be acquired and then perform the conversion.

In summary, IPL SYNCH became the following spinlocks:

FILSYS	File system structures (such as file control blocks)
IOLOCK8	Fork IPL 8 (map registers, data paths and System Communication Services resources)
TIMER	Timer queue
MMG	Memory management, page description database, swapper, and modified page writer
JIB	Portions of the job information block
SCHED	Process control blocks, scheduling database, acquisition/release of mutexes

Per-CPU Context Areas and Interrupt Stacks

Another development task was to identify the context that had to be maintained for each processor — independent of the general system structures. This "per-CPU" context area had to include such items as identification of the current process, a unique CPU identification field, and CPU-specific work queues. In addition, design requirements specified that a processor be able to locate its private CPU context area with minimal overhead.

The easiest solution would have been to include an internal processor register (IPR) into which software could load the virtual address of the context area. Since IPRs are part of the processor hardware, each CPU could have pointed to its own context area without confusion. However, such a processor register did not exist in the

VAX architecture. Therefore another solution was needed in order for SMP to execute on existing VAX systems.

A creative alternative to inventing a new IPR was to find a method to use an existing IPR for multiple purposes. The VAX architecture includes an interrupt stack pointer (ISP) which software loads with the virtual address of the interrupt stack. Since each processor must have its own stack for handling interrupts, this area was already CPU-specific. Under the SMP design, the interrupt stack area and the CPU context area are treated as one virtually contiguous context block. When the virtual address of this new context area is rounded to an appropriate power of two, a simple clearing of the low order bits of the virtual address of the ISP yields the base address of the private CPU context area.

This solution provided two similar ways to find the private CPU context area:

```
MFPR # PR$_ISP,Rx
BICL # mask,Rx
```

or

```
BICL3 # mask,SP,Rx (when running on the
interrupt stack)
```

Both methods return the virtual address of the private CPU context area. However, the latter case provides the faster mechanism.

Translation Buffer Invalidation — A Form of Cache Coherency

As was already mentioned, the VMS SMP design required that cache coherency be maintained in the hardware. However, the VAX architecture includes one hardware cache that is maintained by software, the translation buffer. The translation buffer caches page table entries (PTEs) to speed up address translation from virtual to physical memory addresses.

Software monitoring of the translation buffer is appropriate for two reasons. Since page table pages are only "virtually contiguous" and not "physically contiguous" portions of VAX main memory, monitoring changes to the PTEs would be difficult for hardware. Also, since modification of page table contents is usually an infrequent event, this cache is more suitably maintained by the software.

Therefore, as part of its monitoring function, the operating system software must notify the processor whenever it changes the contents of a

PTE. In case the PTE is cached in the translation buffer. This notification is called a translation buffer invalidation request and is accomplished by a write to an IPR. Since PTEs can be cached on any processor in a multiprocessor system, one possible implementation would be for all CPUs to perform a translation buffer invalidation request when any PTE is changed. Since translation buffer invalidation must be carefully coordinated among all CPUs, however, this simple approach would have significantly affected system performance if left unmodified.

Two other features of VAX/VMS memory management play significant roles in the design for translation buffer invalidation in the SMP environment. First, a user-process address space cannot be executing on multiple processors simultaneously. Second, the cached user-process PTEs are invalidated when a LDPCTX (load process context) instruction is executed as part of process rescheduling.

Using these features, engineers optimized the design to require system-wide translation buffer invalidation only for system address space and not for user address space. Since system addresses change less frequently than user space addresses, this new design allowed for a major reduction in the interprocessor communication traffic.

Process Affinity

Certain operations in a multiprocessor system must execute on particular CPUs. The VMS SMP designers termed the binding of a process to a particular CPU as "process affinity." Affinity for a process is implemented by means of a 32-bit mask (one bit per CPU) in the process control block (PCB). Once a process is assigned affinity, the process may only execute on CPUs for which it has affinity. Process affinity is enforced by the VMS scheduler during a reschedule event. (Note that only for real-time priority processes does VMS SMP guarantee to run the N-highest priority processes on an N-processor system.)

The VMS SMP design currently implements two levels of process affinity: hard affinity and capabilities. Hard affinity forces selection of a single CPU in the affinity mask. This level of affinity is used when a process must be guaranteed execution on a particular CPU, which is specified by the CPU identification field in the PCB. Specifically, hard affinity is used to implement CPU diagnostics and to halt a CPU. When hard affinity is being enforced, the process affinity

mask is reduced to a single bit, which represents the one CPU on which the process may execute. The selection of hard affinity is a very static operation. The selection of which CPU to run on is determined prior to scheduling the process, and the selection remains enforced until otherwise requested.

Capabilities provide a logical mapping of processes to services. These services may only be available on certain CPUs in the SMP environment; for example, primariness is a logical capability. A capability may be serviced by one or more CPUs in the SMP environment. For example, primariness is a capability that is only offered by at most one CPU in the SMP environment.

When a process requires capabilities, the process indicates the desired capabilities in a 32-bit mask in the PCB. When the process is scheduled, a comparison is made of the current requested capabilities and the capabilities offered by the CPU being rescheduled. If the CPU has the required capabilities, then the process is executed; otherwise, the process is ignored and another process is chosen for execution. Any active CPU offering a particular capability may service any process requiring that capability. Once the capability is no longer required by a process, the capability bit in the PCB is cleared and the process can execute on any CPU in the multiprocessing system. Thus, capabilities offer a much more dynamic load-leveling of processes across the CPUs in the system than does hard affinity.

Device Affinity

The VMS SMP design requires that the primary CPU have access to all I/O devices on the system. Due to hardware asymmetry for certain devices in some existing multiprocessing systems, the VMS SMP design also had to include provisions for device affinity. For example, usually both devices in the console subsystem — the console terminal and the console block storage device — can only be accessed by the primary CPU. This is especially evident on 8300 systems, where a physical backplane cable connection from one of the VAXBI slots (usually slot 2, which contains the primary CPU) limits access to the console subsystem to the primary CPU.

Device affinity models the hardware asymmetry by allowing only a subset of the processors to access these I/O devices. Only the portions of

VMS software that access the hardware itself (such as device driver routines that alter control and status registers) must execute on one of the CPUs in the device affinity set for that device. For example, most of the initial processing of a `$QIO` request can execute on any CPU. The driver code actually starts the I/O transfer by controlling the device by means of the control and status registers. Only this portion of the driver code must execute on a member of that device's affinity set.

Under the SMP design, all forking and postprocessing occur on the same CPU that received the device interrupt. The device affinity implementation uses a "trickle down" method that requires no affinity checks for any of the queues. Instead, fork threads are queued to the appropriate CPU in the first place. The SMP implementation queues the fork threads by replicating the I/O postprocessing queue and the fork queues for each CPU in the per-CPU context area. Thus each CPU can process its own fork and I/O postprocessing queue without acquiring the various spinlocks that would be required for system-wide queues. Further, under this design, the set of CPUs to which a particular device is bound under device affinity is a proper subset of the CPUs that can service interrupts for that device.

The affinity field for a device is stored as a bit mask in the unit control block (in the field `UCBSL_AFFINITY`). This bit mask represents those CPUs that are allowed to access the specified device. The default value for `UCBSL_AFFINITY` is `-1`, allowing access from any CPU to the device. As already mentioned, the console subsystem devices are accessible only from the primary CPU; therefore, the `UCBSL_AFFINITY` mask for these devices is initialized to the primary CPU only.

The affinity field for a device is checked on entry to only two of the seven driver entry points:

- `STARTIO`
- `ALT_STARTIO`

If the affinity check fails, the I/O request packet (IRP) is queued as a fork block to another CPU from which access is allowed. The fork block in the `CDRP` portion of the IRP is used to fork the request to another CPU. The fork block is queued to a work request queue in the selected CPU's per-CPU context area. An interprocessor interrupt is then delivered to notify the CPU that work is now present in its work request queue.

All other entry points into device drivers are serviced by the primary CPU, which must be guaranteed access to all devices. These entry points are normally called only during device initialization and include the following entry points:

- TIMEOUT
- UNIT INIT
- CONTROLLER INIT
- CLONED UCB
- UNIT DELIVERY

Process affinity is used to provide the device affinity requirements for the \$CANCEL system service. When the \$CANCEL request is serviced, the UCBSL_AFFINITY field may not allow access from the CPU on which the request was initiated. If access is not allowed, then the process affinity is changed to force the process to execute on a CPU compatible with the affinity requirements of the device.

Some VMS routines are always called when I/O completes on the same processor that serviced the device and fork level interrupt dispatching. Therefore, device affinity is implicit for these routines, and no affinity checks are made prior to calling the routines REGISTER_DUMP and MOUNT_VERIFICATION.

Future Investigations

The initial VMS SMP design is finished, but many interesting areas invite further investigation. These include

- Performance improvements, perhaps finer granularity spinlocks
- Enhancements for parallel processing
- Provisions for higher availability

The key to the VMS SMP design is the new synchronization primitives, that is, spinlocks. The flexibility of the spinlock design will be important in future enhancements to SMP, as already proven in the evolution from static to dynamic spinlocks.

Granularity is another important attribute of spinlocks, which are synchronization points. All synchronization points must be factored into the design of any multiprocessor system. Each spinlock represents at most a single thread of execu-

tion. Therefore, each section of code protected by a spinlock can be executed by only one processor at a time. If two processors attempt to access the same section of code (termed a critical region), then only one processor will proceed while the other(s) spin-waits. To restate Amdahl's Law: You cannot get more than one CPU's worth of work out of any synchronization point.

The ability to increase the number of spinlocks should prove invaluable in future enhancements to SMP, as performance measurements indicate which spinlocks need to change their granularity.

Performance Evaluation of the VAX 6200 Systems

Performance evaluation is an essential element in the development of a computer system. An effort was made to accurately evaluate the performance of the VAX 6200 system under workloads that represent real customer environments. Workloads were developed to represent three major target markets — Engineering/Scientific, Commercial, and General Timesharing. These workloads were used to drive the VAX 6200 systems and thus to evaluate system performance in these environments. Performance measurement results indicate that the VAX 6200 system is a well-balanced multiprocessor system and that the multiprocessor performance is fairly linear across these workloads.

Introduction

The VAX 6240 system is a tightly coupled multiprocessor system based on the CVAX microprocessor. The system consists of four processors sharing memory through a single, high-speed bus. This paper describes the process by which performance of the VAX 6240 system was evaluated under various workloads that represent target markets. The method used to develop and verify these workloads is discussed along with the evaluation of system performance. We use the multiprocessor efficiency measure, defined as the relative throughput obtained by the addition of each processor, to characterize multiprocessor performance. Measurement of the VAX 6240 system indicates that the multiprocessor efficiency measure is directly dependent on the contention for shared resources generated by a workload.

Workload Development

One of the major issues in evaluating the performance of a computer system has been in the workload area. In the context of this paper, workloads are software tools used to create interactive multiuser environments in which the interactive throughput and responsiveness of the system are the key performance metrics. Conversely, benchmarks are either single or multiple copies of programs run in batch mode; the amount of time to complete execution of these programs is the performance metric. The ques-

tion continually debated is how well the benchmarks and workloads represent current user environments. Since there are many different kinds of computing environments and both the applications and computing styles are continually changing, it is very difficult to develop representative workloads accurately. The approach taken here was to first survey the current customer population and identify a few major target markets. Table 1 consists of three surveys obtained from different sources, with n being the sample size.

Table 1 Survey of Customer Environments

Environment	Survey 1 n = 110	Survey 2 n = 200	Survey 3 n = 55K
Engineering/Scientific	46%	50%	31%
Commercial	40%	23%	35%
Education	8%	15%	8%
Software Development	6%	12%	4%
Miscellaneous	--	--	11%

Table 2 Distribution of Customer Environments

Engineering/Scientific	40%
Commercial	40%
General Timesharing	20%

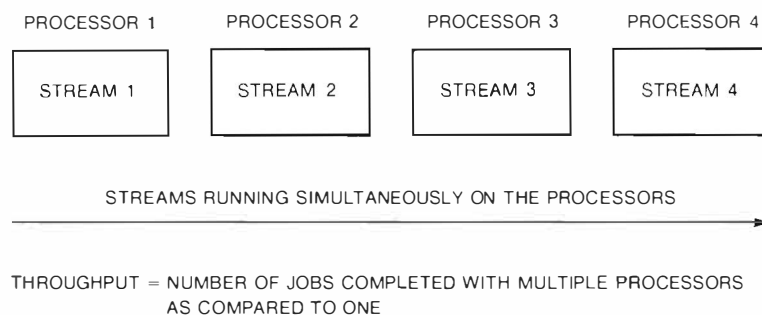


Figure 1 Execution of Multiple Programs Run in Parallel

Clearly, Engineering/Scientific and Commercial environments dominate the market, with Education, Software Development, and General Timesharing applications accounting for the rest. Further examination of the Software Development and the Education environments showed much similarity in function, except that Software Development is slightly more compute intensive. Thus we further simplified the application categories, as shown in Table 2.

We identified typical environments in each of these categories by evaluating system resource consumption in these environments rather than by evaluating what an end user does on the system. Thus we could simplify the number of parameters to CPU, memory, and I/O resource utilizations. Having identified these typical environments, we collected or developed benchmarks and workloads to represent them.

Single Stream

Acquiring single stream benchmarks was not as difficult as developing multiuser workloads. Most of Digital's customers have benchmarks that represent their environments. Therefore, we acquired a collection of benchmarks to represent Engineering/Scientific, Commercial, and General Timesharing from various customer sites. These benchmarks are used to evaluate the single-processor speed.

Multistream Batch Jobs

A stream of well-known benchmarks was selected that represented each of the above-mentioned Engineering/Scientific, Commercial, and General Timesharing markets.

- The engineering stream consists of typical programs used in electrical circuit simulation,

oil reservoir simulation, flight simulation, and linear equation solvers.

- The scientific stream contains simulation programs that use Monte Carlo techniques to track particle movement, along with commonly used routines from national laboratories.
- The commercial stream contains the activities done by a personnel department to support salary planning.
- The general timesharing stream represents the activities done in a software development or education environment.

Multiple copies of this stream were run simultaneously to take advantage of multiprocessor compute resources (Figure 1). To capture the maximum throughput, we ensured that all of the processors were 100 percent busy while the multiple streams were running on the system.

Multiuser Workload Development

The overall process of workload development is shown in Figure 2. Our goal was to represent typical timesharing environments for the different target markets. The entire strategy consisted of

- Identifying typical real sites
- Collecting data on resource utilization and image usage patterns
- Deriving a packaged workload to represent the real site environment
- Validating the workloads by comparing the resource utilization of the workload against the resource utilization at various customer sites and modifying the workloads as required

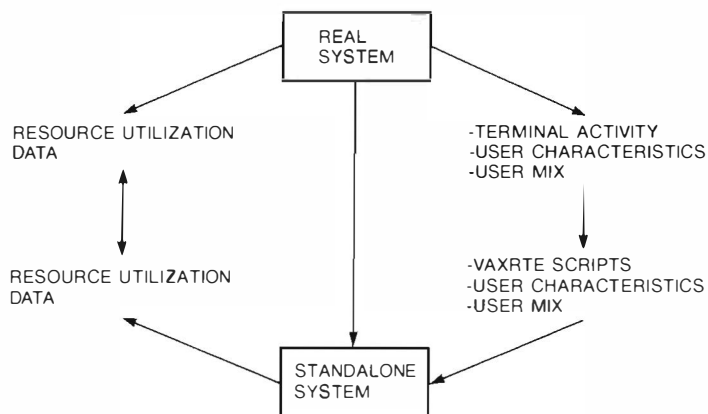


Figure 2 Interactive Multiuser Workload Development

In the following sections, we describe how we used this strategy to develop two multiuser workloads: the engineering workload, which represents an Electronic Computer-Aided Engineering environment (ECAE); and the Software Development Environment Workload (SDEW).

Data Collection

Two Digital sites were chosen to represent the ECAE and SDEW environments. Internal sites were chosen initially to facilitate the data collection process. Both sites had clustered environments that consisted of a variety of VAX systems along with some workstations.

We collected information on these clustered systems to capture their behavior under the load generated by the environment over a period of one week. VAX SPM software was used to collect resource utilization data (CPU, I/O, and memory utilization) on all the systems at both user level and system level. VMS Image Accounting was used to obtain resource utilization data on an image basis. Using the SET HOST/LOG Digital Command Language (DCL) command, we collected log files of user sessions to study user habits. Other user characteristics, such as think time and type rates, were obtained through interviews and observations.

Data Analysis

The performance team studied the cluster-wide resource utilization profiles in order to select the time when the interactive activities were predominant. We compared resource utilization profiles of individual systems against the cluster-

wide average over a week's accumulation of data. Based on this comparison, we selected a typical day and a typical system. One hour was chosen from the typical system on a typical day during the period of peak interactive use to characterize the system at full load.

Further, based on the user profiles, we classified users according to computer usage, that is, heavy or light computing (for ECAE workload) and heavy, medium, or light computing (for SDEW workload). We then used the image accounting data and user log files to classify users according to the type of activity they performed.

Once several user classes were identified, the number of users in each class, or user mix, was determined. We defined the user mix by looking at (1) the number of users in each class at the

Table 3 ECAE and SDEW User Mix

ECAE User Mix	
Type of User	No. of Users
Engineer: Heavy	3
Engineer: Light	3
SDEW User Mix	
Type of User	No. of Users
Heavy software development	1
Light software development	3
Secretary	1
Technical writer	1

one-hour peak, and (2) the organization structure at the real sites. Table 3 shows the user mix for ECAE and SDEW workloads. In addition to interactive users, these workloads also have batch jobs running in the background.

Developing the Workload

Having identified the user classes and activities, we then developed an intermediate workload using DCL command procedures. This intermediate step allowed easier translation to the final workload, which was based on VAXRTE (VAX/VMS Remote Terminal Emulator) scripts. Individual user scripts were developed and validated. We then packaged the entire workload by integrating all of the user scripts and the batch jobs. Once development was complete, the workload was validated at both system and user levels against the real internal site. Further validation was done at the user level against Digital's customer sites.

Workload Validation

This section describes the workload validation process using the ECAE workload as an example of the validation methodology.

Validation against "real" internal site — The workload was tested using the same hardware configuration as the real system. For the ECAE workload, a VAX-11/780 system with 32 megabytes (MB) of memory, RA81 disks, and six interactive users was tested. The purpose of this test was to compare the resource utilization of the workload in an hour-long experiment to the resource utilization of the real system during the typical hour. System- and process-level resource utilization data of several different resources were compared.

User-level validation — To validate the workload at the user level, we compared the average CPU and direct I/O (DIO) utilizations computed for 1 hour for the different user classes. The results are shown in Table 4.

CPU utilization for all three user classes validated to within approximately 10 percent, which was considered to be well within acceptable limits. Validation of the DIO rate was made somewhat difficult because (1) the DIO rate on a per-user basis was very low (0.3 DIO per second for the heavy user), and (2) measurement of the DIO rate is only accurate to 0.1 DIO per second. For all three user classes, the workload came to

Table 4 User Resource Utilization for Real Internal System and ECAE Workload

User Class	CPU minutes/hour		DIO/second	
	Real	ECAE	Real	ECAE
Heavy	1.6	1.5	0.3	0.4
Light	0.5	0.5	0.2	0.1
Batch	42.8	48.5	0.0	0.1

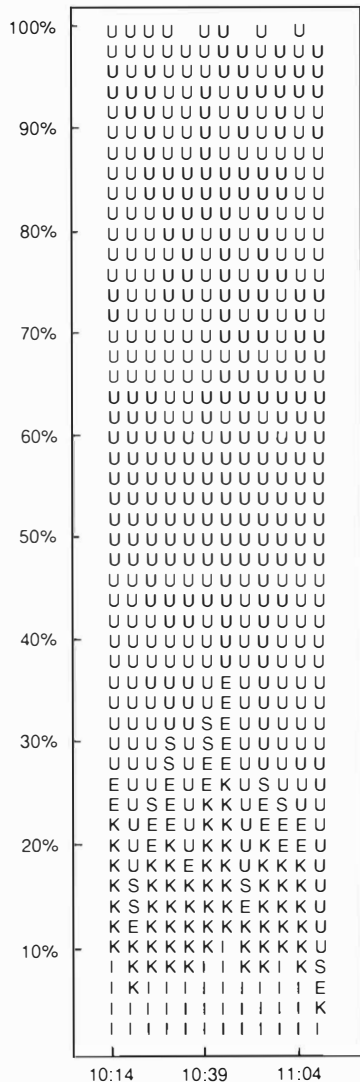
within 0.1 DIO per second of the values measured from the real site.

System-level validation — For system-level validation, we compared the system-level usage of CPU, disk I/O, and memory for the 1-hour ECAE test experiment to the peak hour of the real system. Figure 3 shows that the CPU was used 100 percent of the time on the real system during the 1 hour; whereas the CPU utilization in the workload tended to vary slightly more, but was always between 90 percent and 100 percent saturated. The average CPU utilizations of the real system and the ECAE workload are very close at 100 percent and 93 percent, respectively.

The DIO utilization over a 1-hour period for the two systems is compared in Figure 4. For both systems there is significant variability in the DIO rate over the 1 hour period. The ECAE workload was slightly more bursty, but the average DIO rates for the real system and the ECAE workload were very close at 3.3 and 3.0 DIO operations per second, respectively.

Memory utilization on the two systems did not vary substantially over the 1-hour period. However, total average memory usage with the workload, 23MB, was less than on the real system, 29MB, as depicted in Figure 5.

Although the workload validated very well for CPU and DIO resource utilization, the workload used 20 percent less memory than was used at the real site. This was in part due to the fact that during the development of the workload the CPU and disk I/O utilization of subprocesses was added to the resource utilization of the parent process. Although the workload represents the work done by those subprocesses and the load placed on CPU and disk I/O resources, the workload does not represent the additional memory required by those subprocesses. As will be described in subsequent sections, the lower memory utilization of the workload did not constitute a problem.



REAL SYSTEM

CPU UTILIZATION (PERCENT)
VERSUS TIME OF DAY

FROM: 6-NOV-1986 10:14:06.45
TO: 6-NOV-1986 11:14:39.91

EACH COLUMN = 300 SECONDS
(5 MINUTES)

CPU IDLE

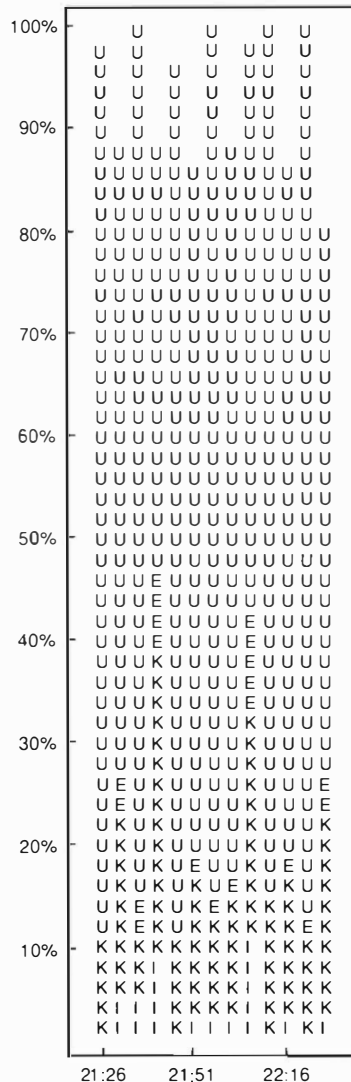
TOTAL IDLE	PAGE WAIT	SWAP WAIT	PAGE & SWAP WAIT
0.1%	0.0%	0.0%	0.0%

CPU BUSY

INTER STACK 6.4%	KERNEL 11.2%
EXECUTIVE 4.0%	SUPERVISOR 1.9%
USER 76.6%	COMPATIBILITY 0.0%
SYSTEM 21.5%	TASK 78.4%

KEY:

- I — INTERRUPT
- E — EXECUTIVE
- U — USER
- K — KERNEL
- S — SUPERVISOR



ECAE

CPU UTILIZATION (PERCENT)
VERSUS TIME OF DAY

FROM: 2-FEB-1987 21:26:13.08
TO: 2-FEB-1987 22:30:13.07

EACH COLUMN = 300 SECONDS
(5 MINUTES)

CPU IDLE

TOTAL IDLE	PAGE WAIT	SWAP WAIT	PAGE & SWAP WAIT
6.7%	6.5%	0.0%	6.5%

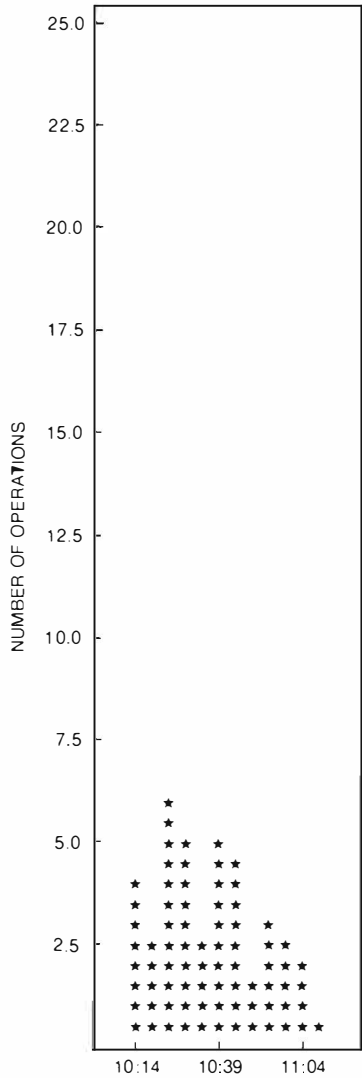
CPU BUSY

INTER STACK 2.8%	KERNEL 14.4%
EXECUTIVE 2.8%	SUPERVISOR 0.1%
USER 73.2%	COMPATIBILITY 0.0%
SYSTEM 20.0%	TASK 73.3%

KEY:

- I — INTERRUPT
- E — EXECUTIVE
- U — USER
- K — KERNEL
- S — SUPERVISOR

Figure 3 CPU Utilization for Real Internal System and ECAE Workload for 1 Hour



REAL SYSTEM

DIRECT I/Os (RATE/SECOND)
 VERSUS TIME OF DAY

FROM: 6-NOV-1986 10:14:06.45
 TO: 6-NOV-1986 11:14:39.91

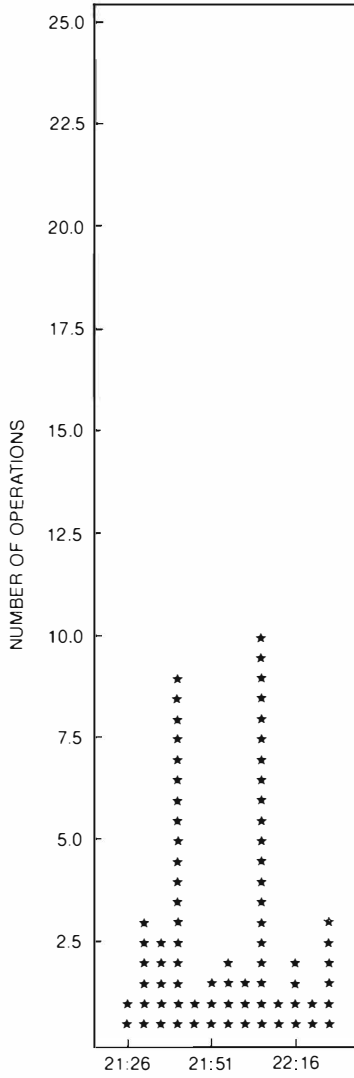
EACH COLUMN = 300 SECONDS
 (5 MINUTES)

I/O RATES (PER SECOND)

DIRECT I/Os 3.3	BUFFERED I/Os 8.5	MAILBOX WRITES 0.6
MAILBOX READS 0.6		LOGICAL NAME TRANSLATIONS 4.4

KEY:

* = DIRECT I/O



ECAE

DIRECT I/Os (RATE/SECOND)
 VERSUS TIME OF DAY

FROM: 2-FEB-1987 21:26:13.08
 TO: 2-FEB-1987 22:30:13.07

EACH COLUMN = 300 SECONDS
 (5 MINUTES)

I/O RATES (PER SECOND)

DIRECT I/Os 3.0	BUFFERED I/Os 4.5	MAILBOX WRITES 0.3
MAILBOX READS 0.3		LOGICAL NAME TRANSLATIONS 3.6

KEY:

* = DIRECT I/O

Figure 4 DIO Utilization for Real Internal System and ECAE Workload for 1 Hour

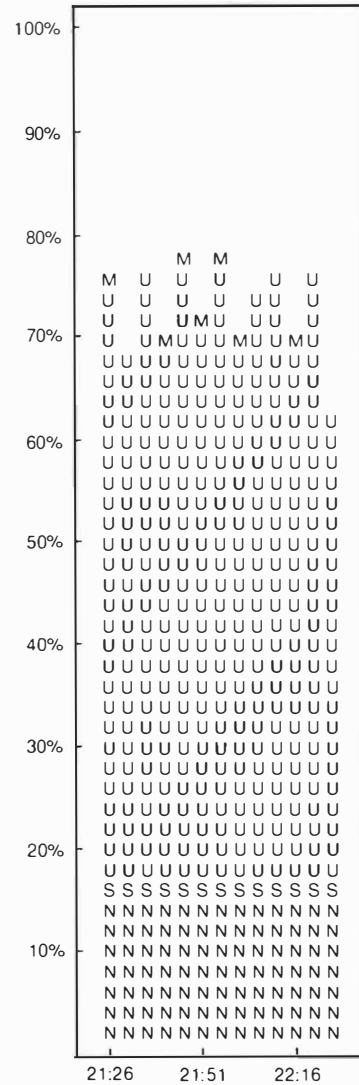
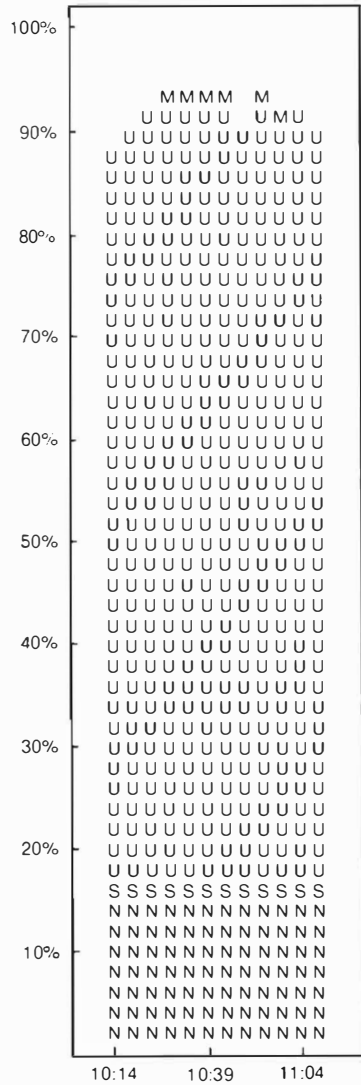


Figure 5 Memory Utilization for Real Internal System and ECAE Workload for 1 Hour

A summary of the comparisons of the average resource utilizations for the real system and the workload is presented in Table 5.

Validation against customer sites — This validation of the workload against the internal system was followed by validation against customer systems. The goal of this additional validation was to determine if the workload was representative of the load placed on systems by Digital's customers.

Two semiconductor manufacturers in California were used as validation sites for the ECAE workload. Initially, it was determined that there were significant differences between the work performed at these customer sites and the work performed at the internal Digital site. The Digital internal VAX systems were used for logic design of gate arrays, circuit boards, and systems; whereas at the external sites, the VAX systems were used for the design of integrated circuits. Specifically, the work differed in the following ways:

- DECSIM is used extensively within Digital, whereas SPICE is the predominant simulation software used by external semiconductor developers. DECSIM simulations require very large amounts of memory as compared to the SPICE simulations done by customers.
- Design rule checking is both a time-critical and disk I/O-intensive task done by semiconductor designers. Design rule checking and the load it places on the I/O subsystem were not executed at the internal Digital site at the time resource utilization data was collected.

As a result, we modified the ECAE workload to include the load placed on the system by design rule checking and replaced the use of DECSIM with SPICE.

System resource utilization data was collected on VAX 8800 systems for one week at these customer sites. In a manner very similar to the process used for the initial development of the workload, the data from these sites was reduced to a typical peak period. Table 6 presents the comparison of resource utilization on a per-user basis in the workload and at customer sites.

The ECAE workload falls within the range of utilizations observed at these customer sites for both disk and memory utilizations. The workload is slightly (approximately 10 percent) more CPU intensive on a per-user basis than was observed at

Table 5 System-Level Resource Utilization for Real Internal System and ECAE Workload

Resource	ECAE	Real System
CPU busy	93%	100%
DIO/second	3.0	3.3
Memory	23MB	29MB

Table 6 Comparison of Resource Utilization on Customer System and in ECAE Workload

Resource Utilization per Hour	Customer Sites	ECAE Workload
CPU (minutes/hour)	3.8-4.5	5.0
DIO operations/second	1.4-2.3	1.8
Memory (MB)	0.7-0.8	0.8

customer sites. This workload will put a 10 percent heavier load on the system, making the performance numbers slightly conservative for the computer-aided electrical engineering market.

Performance Measurement and Analysis

This section discusses the performance of the systems in three major applications: Engineering/Scientific, Commercial, and General Time-sharing. In each of the environments, single stream, multistream, batch, and multiuser workloads were tested.

Single-Stream Performance

The first step in evaluating the performance of a multiprocessor system is to establish the base-level performance of the uniprocessor relative to a well-known system such as the VAX-11/780. A large number of single-user benchmarks were used to establish this base level.

Single-User Performance

Single-user performance was evaluated by using traditional synthetic benchmarks, well-known industry standards, and real application programs from engineering, scientific, commercial, and general timesharing environments. Most of the synthetic benchmarks are in FORTRAN; industry standards are Whetstones, Dhrystones, Linpack, and others. The real applications, as mentioned, represent four environments.

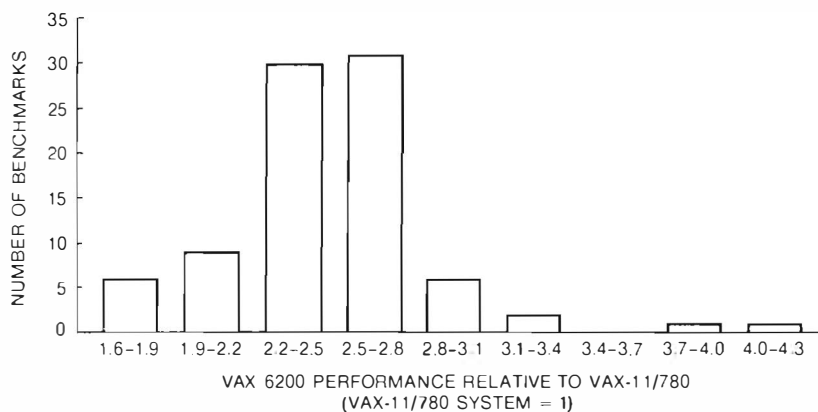


Figure 6 Frequency Distribution of the VAX 6210 Performance on the Single-User Benchmark Set

These benchmarks were used to evaluate uniprocessor speed compared to a VAX-11/780 system. A frequency distribution of the speedup factors on all these benchmarks was plotted, and the central tendency was examined. (See Figure 6.) A high percentage of the benchmarks fell between 2.2 and 2.8.

Table 7 summarizes the performance of the VAX 6210 in the single-user environment relative to a VAX-11/780 system. The performance average of the VAX 6210 system, across all these benchmarks, is 2.8 times the performance of a VAX-11/780 system.

Decomposed Single-user Performance

VAX 6200 performance on decomposed programs was evaluated through the use of manual and directed decomposition techniques. To begin with, a program is evaluated to see if some

segments can be separated into parallel threads that can be run independently. Then the program is decomposed and run, either manually or through directives. The program is initiated as a single job; then the segments of the program that lend themselves to decomposition are divided into subprocesses and executed in parallel on different processors. In the manual decomposition method, the optimal number of subprocesses for various levels of multiprocessor systems is evaluated by varying the number of subprocesses and calculating the speedup factors. In the directive decomposition method, the compiler takes care of various optimization factors. These programs were run standalone with no interference from any other programs on the system. Figure 7 illustrates the decomposition process.

The benchmark description is as follows. To evaluate the maximum speedup factors that can be achieved through decomposition, code segments were selected. Such segments as matrix multiplication and convolution are widely used in engineering/scientific applications. Different array sizes (from 100 to 1000) were used with various arithmetic data types such as integer, and single and double precision.

An image processing program and the Linpack1000D program were used to represent real application programs, where only certain segments can be decomposed.

The performance results are as follows. The multiprocessor efficiency measure, defined as the relative speedup obtained by the addition of each processor, is the key metric used here to evaluate

Table 7 Performance of the VAX 6210 in the Single-User Environment

Synthetic Benchmark Set:	
Single-user set	2.5
Industry-standard Benchmarks:	
Whet-s & -d	2.3
Linpack-s	2.7
Linpack-d	3.2
Dhrystone	2.8
Real Application Benchmark Set:	
Engineering set	2.8
Scientific set	2.6

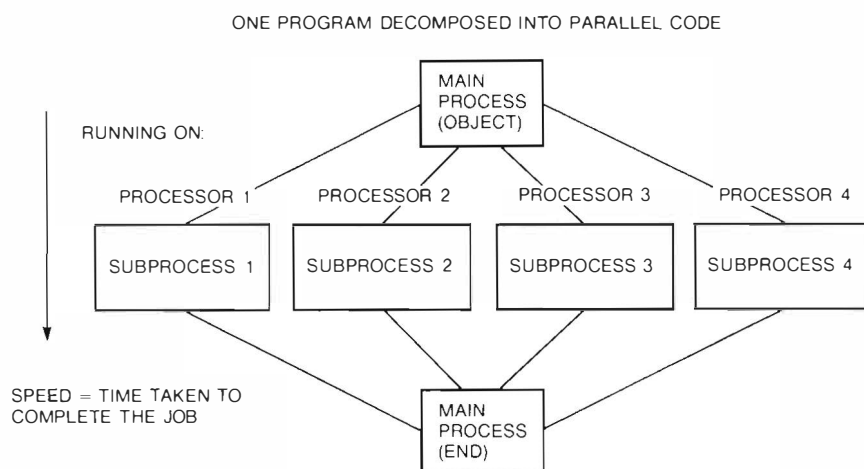


Figure 7 Program Decomposition Process

performance. As seen in Figure 8, the multiprocessor efficiency measure on the program kernels is fairly linear. Multiprocessor synchronization is minimal in this computing environment. The performance was very close to the theoretical maximum. A speedup of 3.9 times the uniprocessor performance was achieved on the four-processor 6240 system. The performance on the image processing program is slightly lower than what was observed on the program kernels. Thus performance gained by decomposition depends directly on the amount of code that can be run in parallel. (Note: On the Linpack1000D program, directed decomposition was used; whereas on the other programs, manual decomposition was used.)

Multistream Batch Performance Measurement and Analysis

The multistream jobs were used to measure the system-level batch performance on the multiprocessor systems. As shown in Figure 1, these multiple streams were run in parallel to allow concurrency in the execution of these streams. Maximum concurrency is achieved since each of these streams is identical. No single stream runs any faster; however, the number of jobs completed increases almost linearly with the addition of processors. Adequate memory was allocated to the jobs to avoid unnecessary paging and swapping. In addition, sufficient I/O resources were present on the system to preclude I/O bottlenecks. The elapsed time to complete these jobs

was recorded and used to evaluate the multiprocessor batch throughput performance. It is important that all the streams run simultaneously and share resources equally. Large differences in the completion times of streams would imply that maximum concurrency was not achieved because of some bottleneck in the system.

Multiprocessor performance on multistream batch jobs was very close to linear across all environments. Results for the commercial stream, representing personnel administration, were only

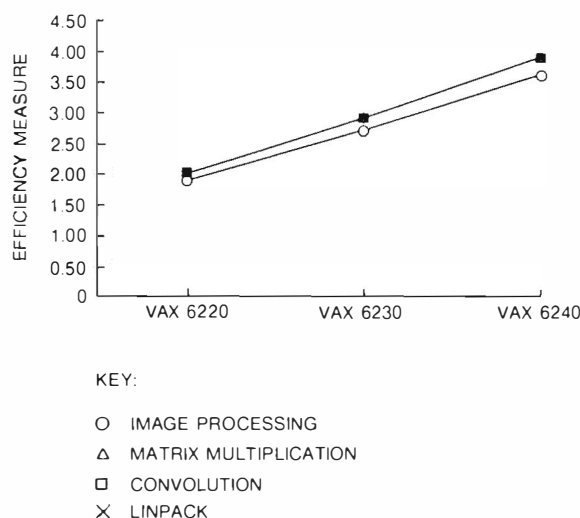


Figure 8 Multiprocessor Efficiency through Parallel Processing

slightly lower — probably because of the higher amount of I/O on this stream. (See Figure 9.)

Interactive Multiuser Performance Measurement and Analysis

In the interactive multiuser environment, the system must support the activities of a substantially higher number of users and their frequent interaction with the system. The number of users on the system increases the amount of context switching, and the contention for shared resources is also much higher in this environment.

The test methodology included the use of a remote terminal emulator, VAXRTE, to create the interactive multiuser environment. The VAXRTE generated the input for the system under test and received consequent output. The VAXRTE also logged and time-stamped all interactions and maintained the job mix throughout the experiment. To run a multiuser experiment, the system under test and the VAXRTE system were booted and running. Using scripts, every few seconds the VAXRTE logged a user on to the system under test. After all logins were completed, sufficient time was allowed for the system to reach a steady state. The experiment was then run long enough to execute the longest script cycle for the specific workload. While the experiment was running, VMS monitor and other monitoring tools were used to capture the resource utilization data. When the experiment was completed, data was reduced and analyzed.

Workload description — Three interactive multiuser workloads were used to evaluate the multiprocessor performance in the three major environments: Engineering, Commercial, and General Timesharing.

The Engineering environment was represented by an ECAE workload. This workload consists of the types of tasks done by design engineers developing electronic circuits: circuit simulation, design rule checking, schematic file transfers from workstations, and tasks supported by VMS utilities.

The multiuser Commercial (Compu-Share) workload is based on the Compu-Share Order Processing software package. This workload consists of three major types of transactions: order entry, order inquiry, and accounts receivable reporting.

The General Timesharing SDEW represents the types of tasks done by software engineers. The

major tasks executed in this workload are compile-link-execute-debug cycle using FORTRAN, BLISS, and MACRO; utilities used include CMS, RUNOFF, and text editors.

Hardware/software setup — Table 8 summarizes the hardware and software configurations.

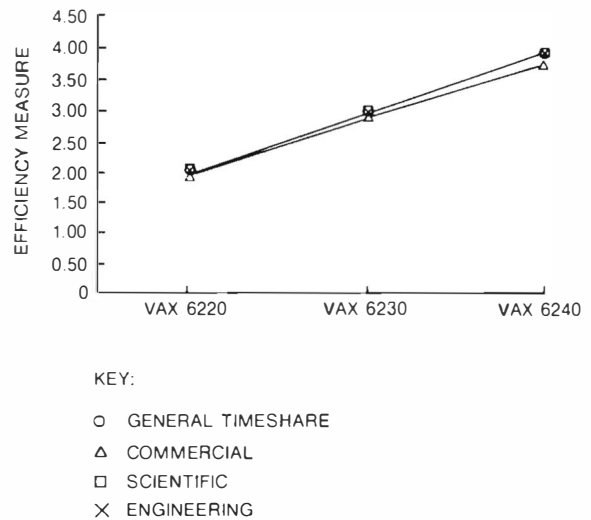


Figure 9 Multistream Efficiency Measures

Table 8 Summary of Hardware and Software Configurations

Hardware Configuration			
Processor	VAX 6240		
Memory	128MB		
Disk controller	2 HSC70		
Disks	(Disk configurations differed for each workload; see below.)		
Number of RA82 Disks per Workload			
Dedicated Use	Number of RA82 Disks per Workload		
	ECAE	Compu-Share	SDEW
System	1	1	1
Page/swap	1	1	1
Library	—	—	1
Interactive	2	2	4
Batch	3	—	2
Database	—	6	—
Note: Where necessary, software was distributed over multiple disks to avoid disk bottlenecks.			
Software Configuration			
VMS V5.0 - FT2.1 (A single-processor system was run with multiprocessing turned off.)			

Performance metric — The multiprocessor efficiency measure is defined to be the relative multiprocessor interactive throughput compared to the uniprocessor throughput.

Also considered in this metric is system responsiveness, based on acceptable service criteria for light, medium, and heavy tasks. This metric is used to evaluate the number of users supported at peak throughput while the system maintains the service time criteria. System resources required to support each application are also identified.

Performance results — The multiprocessor efficiency measure is very close to linear in both the ECAE and SDEW environments (Figure 10). This result shows that even in the multiuser interactive environments near-linear performance can be expected if the system is well balanced in terms of processor speed and the memory-to-processor bus speed. It also indicates the efficiency of the VMS SMP software. In the Compu-Share environment, the performance was slightly lower because of the high amount of disk and terminal I/O generated by this workload. The performance of the multiprocessor systems under symmetric multiprocessing (SMP) depends directly on the amount of I/O. It is important to note that even with high amounts of I/O, the multiprocessor efficiency measure is well over three for the four-processor system.

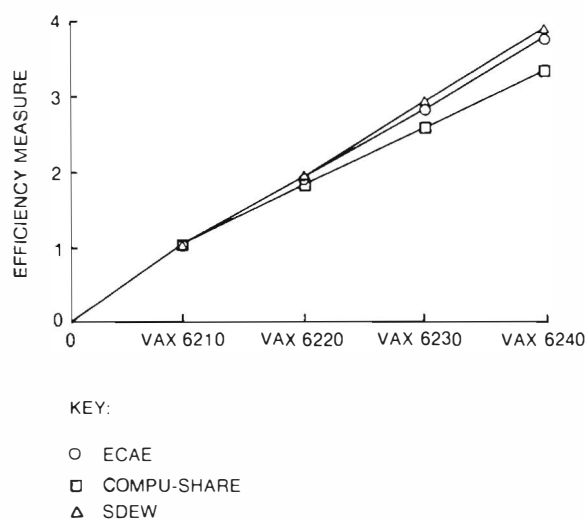


Figure 10 Multiprocessor Efficiency Measure for All Multiuser Workloads

At the peak throughput levels, response time criteria were maintained in each workload. Table 9 compares users supported and resources used by each of these workloads. The maximum number of users supported on the VAX 6240 are 38, 120, and 126 users for ECAE, Compu-Share, and SDEW, respectively.

In terms of resource utilization, it should be noted that the multiprocessor synchronization

Table 9 Summary of Workload Resource Utilizations

Multuser	ECAE	Compu-Share	SDEW
Number of users supported at the peak	10, 20, 28, 38	30, 60, -, 120	36, 66, 90, 126
Resource utilization			
Number of users	38	120	126
CPU - 6240			
Percent utilized	100%	100%	100%
Interrupt	2%	6%	4%
Kernel	12%	29%	20%
Executive	3%	7%	7%
MP synch	1%	7%	2%
User	82%	51%	67%
I/O			
Disk I/O profile	Bursty	Uniform	Bursty
Average disk I/O per second	24	113	68
Average buffered I/O per second	82	112	76
Memory			
Maximum used (MB)	32	60	57

under SMP is handled by spinlocks. A spinlock is a bit in shared memory that is accessible by means of interlocked instructions by all processes through mutual agreement. Mutual agreement implies that a process can set the bit and gain access to the scheduler database if no other process has access to it. If a process tries to set the bit and the bit is already set, then the process continues to "spin" using a sequence of instructions to continue checking to see if the bit is clear. MP synch is the amount of CPU time spent waiting to change the bit or acquire the spinlock and thus gain access to the scheduler database. MP synch is 1 percent for ECAE, 7 percent for the Compu-Share workload, and 2 percent for the SDEW workload. Since MP synch is the CPU time spent waiting to acquire spinlocks and indicates the amount of spinlock collisions, it shows the level of contention for shared resources experienced by SMP under each workload. For the Compu-Share workload, this level is significantly higher. The Compu-Share workload generates the most disk I/O compared to the other workloads, which may be the reason for a higher amount of time spent by this workload in MP synch.

The following three graphs, Figures 11, 12, 13, present the CPU modes usage profiles. The

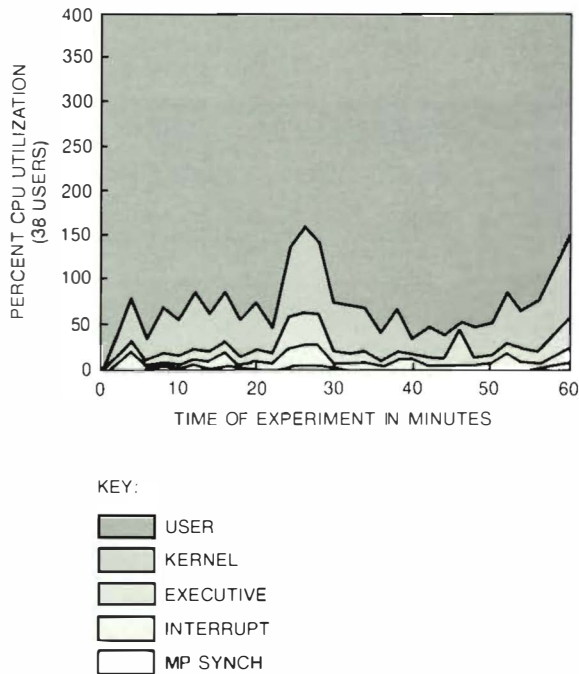


Figure 11 CPU Utilization over Time — ECAE Workload

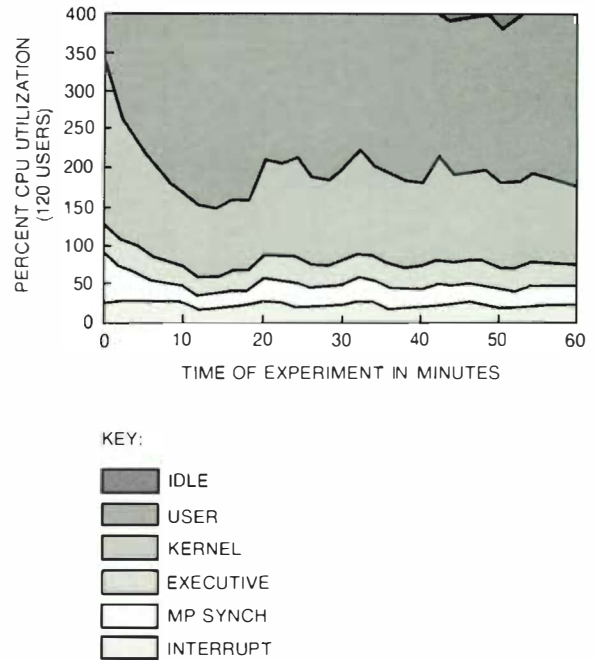


Figure 12 CPU Utilization over Time — Compu-Share Workload

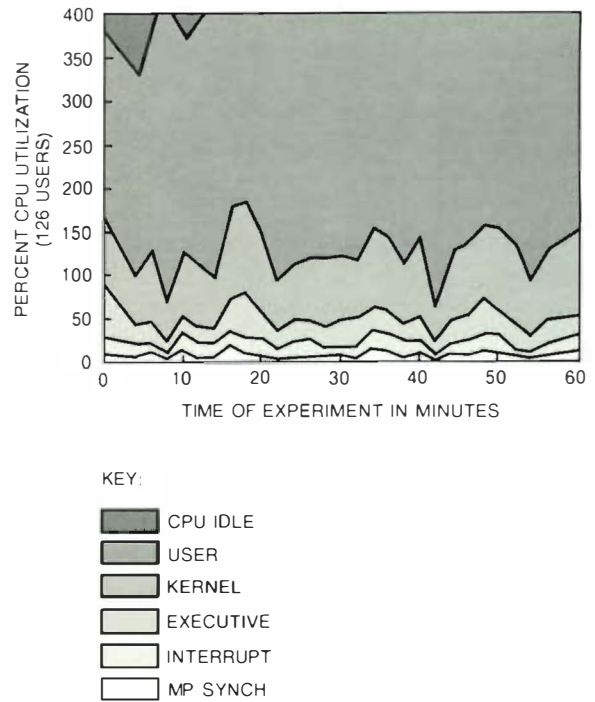


Figure 13 CPU Utilization over Time — SDEW Workload

Compu-Share workload shows higher but more uniform interrupt and kernel mode activities. Compu-Share's use of databases, which generates heavy I/O and local locking, is manifested in the heavy kernel and interrupt mode activity. SDEW does a fair amount of file manipulation. ECAE has much lower I/O activity than both Compu-Share and SDEW.

The next three graphs in Figures 14, 15, and 16 compare the I/O profiles. The disk I/O on ECAE and SDEW is very bursty, and it is interesting to note that their relative CPU mode profiles correlate well, showing a relationship between the two. The I/O on Compu-Share is high but not as bursty.

Comparing the disk I/O generated by the workloads and the effect it has on CPU utilization. Compu-Share puts the heaviest load on the multiprocessor system. However, even with all the synchronization necessary on this workload, the multiprocessor efficiency measure is fairly high (3.3). The ECAE and SDEW workloads show high multiprocessor efficiency measures of 3.8 and 3.9, respectively. This level of gain in the

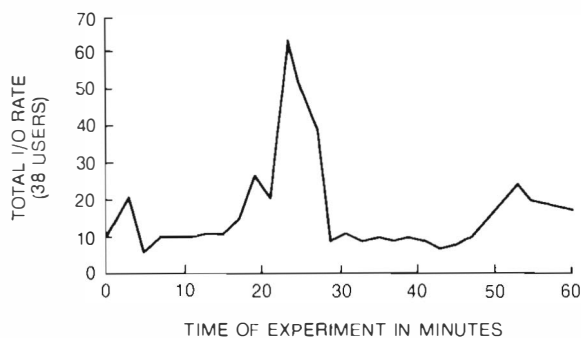


Figure 14 Disk I/O Utilization over Time — ECAE Workload

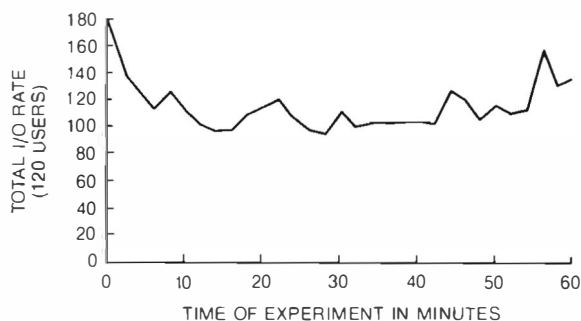


Figure 15 Disk I/O Utilization over Time — Compu-Share Workload

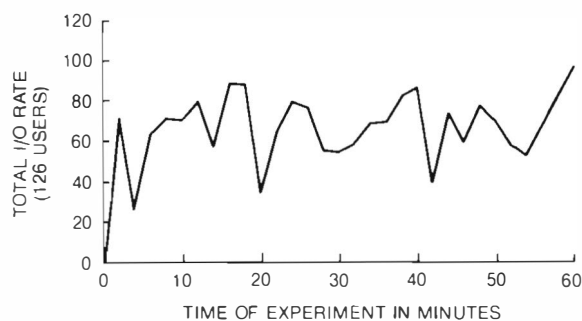


Figure 16 Disk I/O Utilization over Time — SDEW Workload

multiuser environment on the multiprocessor systems shows that VMS SMP is working efficiently and that the VAX 6240 system is a well-balanced system in terms of the processor and bus speeds.

Application Characteristics Affecting Multiprocessor Performance

This section discusses some of the characteristics in applications that directly affect multiprocessor performance.

Memory-to-Processor Traffic

Since these multiprocessor systems share memory, contention to access memory could be a factor that affects multiprocessor efficiency. Therefore applications that generate lower memory-to-processor traffic do perform better, assuming there are no other bottlenecks in the system. One way to reduce this traffic is to organize the data to improve locality of reference. Data that is accessed together should be placed together.

Disk I/O Operations

With the symmetric multiprocessing software, I/O operations can be handled by each of the processors. As a result, the I/O-intensive applications perform much better on the symmetric multiprocessor systems as compared to the asymmetric multiprocessing systems. However, the I/O device interrupts are still handled by the primary processor, even under SMP. By reducing the rate at which device interrupts are made, any contention for the primary processor can be reduced. To reduce the number of I/O interrupts, larger block transfers may be better in I/O-intensive applications. Thus, an application that will lend itself to making larger block transfers

With minimum bus cycle time down by more than a factor of two and dynamic random-access memory (RAM) access time remaining relatively constant, the opportunity arose to increase performance by using static RAM to add a cache. Static RAMs with 35-ns access times and 64-kilo-bit (Kb) densities could be used for this purpose at reasonable cost.

Design Partitioning and Functionality

To facilitate implementation of the processor module using custom VLSI, the design was partitioned into seven major parts: the central processing unit with first-level cache, the floating point unit, the second-level cache, the memory controller, the Q22-bus interface, the system support functions, and the clock circuitry. Each of these partitions was implemented by a single chip, with the exception of the second-level cache. This cache was implemented by programmable array logic (PAL) and static RAMs. Five of the parts are connected directly to a 32-bit multiplexed address/data (CDAL) bus: the central processing unit with first-level cache (CVAX), floating point unit (CFPA), second-level cache, memory controller (CMCTL), and Q22-bus interface (CQBIC). To reduce loading, the chip containing the system support functions (SSC) connects to a buffered version of this bus, the BCDAL. The clock circuitry (CCLK) was separated from the processor chip to conserve pins as well as to allow designers more flexibility in choosing a clock rate.

To maximize performance, the CVAX, CFPA, second-level cache, and CMCTL operate synchronously from a four-phase clock generated by the CCLK. The SSC and CQBIC operate asynchronously on a 40-MHz oscillator. The processor module was designed to allow the CCLK to be fed either from the 40-MHz oscillator or from a separate oscillator. The separate oscillator allowed the central processor and memory subsystems to be sped up when it was determined that the CVAX, CFPA, and CMCTL chips were capable of running ten percent faster than originally projected.

Each of the major parts of the processor module is described in following sections.

The Central Processing Unit and First-level Cache

The CVAX chip is a microcoded 32-bit VAX CPU. To implement the entire VAX architecture using a

single chip, the CVAX designers selected a subset of the full VAX instruction set and data types. The implementation includes 175 instructions and six data types (also implemented by the MicroVAX II system), plus 6 additional string instructions: CMPC3, CMPC5, LOCC, SCANC, SPANC and SKPC. The CVAX also provides microcode support for emulation of 53 additional instructions (six less than the MicroVAX II) and five data types. When any of these instructions is decoded, an emulated instruction exception is generated. This exception causes a set of instruction-specific parameters to be pushed on the stack and control to be passed to operating system emulation routines by the emulated instruction vector in the system control block. As in the MicroVAX II, the remaining 70 instructions and three data types are handled by the CFPA chip.

The CVAX implements the following registers:

- Sixteen, 32-bit, general-purpose registers
- Twelve VAX standard internal processor registers to support memory management, process control, interrupts and system identification (SBR, SLR, MAPEN, TBIA, TBIS, TBCHK, PCBB, SCBB, IPL, SIRR, SISR, and SID)¹
- Five internal processor registers specific to the CVAX to support the interval clock, first level cache, error reporting and console emulation (ICCS, CADR, MSER, SAVPC, and SAVPSL)²

The CVAX also provides a means for accessing six additional VAX standard internal processor registers to support the time-of-year clock, console serial line, and I/O bus (TODR, RXCS, RXDB, TXCS, TXDB, and IORESET).¹ These registers are implemented in the SSC.

The registers in the SSC are referred to as "external" internal processor registers and are accessed by software in the same manner as other internal processor registers, that is, by means of MTPR and MFPR instructions. However, the CVAX chip generates a special cycle on the CDAL bus with the register number as an address. The SSC responds to these cycles by either supplying the CVAX with the register contents (MFPR) or performing the register update (MTPR). Accesses to other unimplemented VAX internal processor registers will also cause these cycles to be generated, but the cycles will terminate with an error condition. (The cycles are timed out after four microseconds by a CDAL bus timer in the SSC.) When a register write is made to an unimple-

mented internal processor register, the CVAX ignores the error signal; the result is a long no-operation. When a register read of an unimplemented internal processor register is attempted, the results are undefined.

Also like the MicroVAX II system, the CVAX processor implements a memory management unit. The unit supports full VAX demand-paged virtual memory, with single-level page tables for system space addresses and double-level page tables for process space addresses. In addition, four levels of access protection are supported by the memory management unit. A 28-entry, fully associative address translation buffer is provided for storing recent virtual-to-physical address translation (as opposed to an 8-entry translation buffer in the MicroVAX II).

Unlike the MicroVAX II system, the CVAX includes an on-chip (first-level), physical instruction and data cache. Because chip area was at a premium, a 1 KB, two-way set associative organization was chosen. In contrast to the second-level cache, this organization achieves a high hit rate for the available chip area through increased control logic complexity instead of increased storage array size. The extra control logic complexity of the first-level cache is more efficiently implemented in custom VLSI, whereas the large storage arrays of the second-level cache are more efficiently implemented with off-the-shelf parts. Since the first-level cache organization yields a set size equal to the memory page size, cache look-up and virtual-to-physical address translation can be overlapped. Thus a cache cycle time equal to the processor microcycle time is achieved.

The first-level cache is look-through; that is, cache hits on read cycles result in no activity on the CDAL bus, thus preserving its bandwidth for DMA transfers. The block size is one quadword so that cache misses on cacheable read cycles cause the CVAX to generate a quadword transfer on the CDAL bus. This transfer results in two longwords of data being returned in response to a single address. The minimum transfer time is two microcycles for the first longword and one for the second, which increases the effective CDAL bus bandwidth. Further, the first-level cache is write-through. However, to improve performance, the CVAX also contains a longword write buffer which allows the CPU execute out of the first-level cache while the write operation is being completed.³

The Floating Point Accelerator

The CFP chip works in conjunction with the CVAX chip to process floating point instructions and to accelerate the execution of some integer instructions (MULL, DIVL, and EMUL). The CVAX decodes the instructions and sends the CFP control and opcode information by means of a dedicated eight-line control bus. The CFP gets its operands from the CDAL bus. Unlike the MicroVAX II, all operands do not have to come from the CPU. Operands come from the CVAX only if they reside in the general-purpose registers or first-level cache. If the operands reside in the second-level cache or main memory, the CFP takes them directly off the CDAL bus. When the CFP has completed the operation, it returns condition codes and exception status by means of the control bus, and the unaligned result by the CDAL bus. One, two, or three longword transfers may be required to transfer the result, depending on the type of operation. The CVAX aligns and sends the result to its ultimate destination. To improve DMA latency, the CVAX will grant the CDAL bus requests while waiting for the CFP to return the result.⁴

The Second-level Cache

The second-level cache sits directly on the CDAL bus and bridges the 4-microcycle gap in access time between the first-level cache and main memory. The project goal for the second-level cache was to maximize system performance without placing the schedule at risk. Consequently, designers chose to use large storage arrays to achieve the desired level of performance (hit rate) rather than complex control logic. By keeping the control logic simple, the cache could be implemented in PALs rather than custom VLSI. Thus the chance of design errors was reduced as well as the time needed to correct any errors found during design qualification.

The large storage arrays were easily implemented using off-the-shelf static RAMs. The resulting design was a 64KB, direct-mapped, physical instruction and data cache with write-through. The implementation called for six PALs for control logic, eight 16K-by-4 static RAMs and four 16K-by-1 static RAMs for the data store, and three 16K-by-4 static RAMs for the tag store.

In keeping with the philosophy of simple control logic, the second-level cache is look-aside; that is, address decoding occurs in parallel in the cache controller and the memory controller.

Therefore, the cache does not have to regenerate CDAL bus cycles in the event of a cache miss. The second-level cache control logic simply

- Watches the CDAL bus cycles
- Returns data to the CVAX on cacheable read cycles that miss the first-level cache but hit the second-level cache
- Allocates a block on cacheable quadword CVAX read cycles that miss both caches
- Updates an entry on CVAX write cycles that hit the second-level cache
- Invalidates a block on DMA write cycles that hit the second-level cache
- Ignores DMA read cycles

Because the second-level cache stores the same types of references as the first-level cache, very little control logic is required to determine which CVAX references are cacheable. The CVAX will only generate quadword CDAL bus cycles on cacheable CPU references that miss the first-level cache. Therefore, the second-level cache control logic only considers quadword read cycles cacheable.

To respond within the minimum CVAX bus cycle time (one microcycle for the second longword of a quadword cycle), the second-level cache control logic uses an overlap scheme. The second-level cache overlaps the address generation and the tag look-up for the second longword portion of the cycle with the data access for the first longword portion of the cycle.⁵

The Memory Controller

The CMCTL chip is the interface between the CDAL bus and the memory array. The chip is a full 32-bit, single-ported, synchronous memory controller with 7-bit error-correcting code (ECC) and supports up to four memory array modules (two more than the MicroVAX II).

The CMCTL longword write buffer minimizes the effect of write operations on CPU performance. (Both caches are write-through.) The CMCTL also supports multiword transfers on the CDAL bus. On these transfers, the CMCTL utilizes page mode in the dynamic RAMs to achieve the performance of an eight-way interleaved memory subsystem without the use of additional banks or interconnect complexity. The size of the transfer is encoded in bits 31 through 30 of the physical address (up to four longwords). Thus with only a single address, the memory controller can fetch

sequential longwords in less time. Both the CVAX and the CQBIC utilize this feature to improve performance. The CVAX generates quadword transfers to fill cache blocks on a cache miss; and the CQBIC generates quadword, hexaword, or octaword transfers on block-mode DMA by devices on the Q22-bus. The combination of multiword transfers and the look-through first-level cache made the added complexity of dual ports (as used in the MicroVAX II) unnecessary. To work effectively with the look-aside second-level cache, the CMCTL must monitor the CDAL bus after starting a memory operation. If the second-level cache responds with the data first, the CMCTL aborts its operation before completion.

To support a range of CVAX microcycle times and also maintain the performance advantage of synchronous operation, the CMCTL includes a programmable wait-state bit. This bit controls the number of CPU microcycles used to access the RAM array. Moreover this bit allows the same array modules to be used for processors with different microcycle times.⁶

The memory controller was not designed to support battery back-up because of the added design complexity and cost. For those applications that require support during power outages, standby uninterruptable power supplies are a better solution and are available for small systems at low cost.

The Q22-bus Interface

The CQBIC interfaces the CDAL bus to the Q22-bus. This chip provides address translation between the 26-bit CDAL bus and 22-bit Q22-bus. In addition, CQBIC handles data buffering between the 32-bit synchronous/asynchronous CDAL bus and the 16-bit asynchronous Q22-bus. Q22-bus addresses are translated to CDAL bus addresses by a programmable mapping function (scatter-gather map), which is software compatible with the MicroVAX II system. This function gives the CPU the capability to map any page of the 4 megabyte (MB) Q22-bus address space to any page of the main memory address space. Thus Q22-bus DMA devices can transfer directly to or from discontinuous pages of main memory. CDAL bus addresses are translated into Q22-bus addresses by a direct mapping function. This function maps the 4MB Q22-bus memory space and the 8KB Q22-bus I/O space into the VAX I/O space. Thus the CPU can directly access Q22-bus memory or device registers by means of two ranges of I/O page addresses.

DMA write references are buffered in two naturally aligned octaword buffers and transferred to main memory by the most efficient combination of multiword transfers. The two octaword buffers allow an entire block-mode transfer (up to 16 words) to be buffered by the CQBIC. After the first buffer has been filled by the Q22-bus device, it is emptied into main memory while the Q22-bus device fills the second buffer. Since the CDAL bus is faster than the Q22-bus, the first buffer is emptied and ready for input from the Q22-bus device before the second buffer has been filled. This arrangement allows the interface to provide sustained throughput at maximum Q22-bus transfer rates with no additional latency.

Q22-bus block-mode DMA read references are translated into quadword transfers on the CDAL bus. The four words are buffered in a single quadword buffer and supplied to the DMA device on demand. Before the buffer is emptied, the next quadword is prefetched. This prefetch eliminates additional latency on all but the first transfer. To keep the latency of the first transfer at a minimum, the CQBIC responds to the DMA device after receiving the first longword of a quadword CDAL bus cycle, rather than waiting for the entire quadword transfer to complete.

To fit the entire Q22-bus interface in a single chip, some changes had to be made to the bus interface architecture of the MicroVAX II system. On the MicroVAX II, the scatter-gather map was stored in a dedicated 32KB static RAM array within the bus interface. On the CQBIC, not enough space was available to implement this storage array internal to the chip. Moreover, not enough pins were available to provide a dedicated bus to an external static RAM array. The solution was to store the scatter-gather map in a 32KB block of main memory and to implement a 16-entry fully associative cache for map entries in the CQBIC. The cache functions in the same manner as an address translation buffer. When translating a Q22-bus address, the cache is checked for the appropriate map entry. If the entry is found, the translation takes place at maximum speed. If the entry is not found, then there is a delay while the entry is fetched from main memory. The translation is then performed. This delay is eliminated on DMA transfers that cross a page boundary, because the entry that maps the next page is prefetched when the DMA operation reaches a page boundary. On most DMA transfers, this delay is negligible because it is amortized over a large number of Q22-bus transfers. The

design ensures that the operating system does not attempt to use the block of memory where the scatter-gather map resides. The on-board firmware does not include these pages in a list of good memory pages that is passed to the operating system at boot time. An interesting side effect of putting the scatter-gather map in main memory was that the relatively long latency on some Q22-bus DMA cycles uncovered latent design bugs in several Q22-bus DMA devices. The designs of these devices had been verified by empirical testing with existing processors rather than by testing to the Q22-bus specification.

To maintain software compatibility with the MicroVAX II system, the scatter-gather map is referenced through a 32KB block of I/O space addresses. The CQBIC responds to writes in this address range by buffering the data so the CVAX cycle can complete, updating the cache if there is a hit, requesting the CDAL bus, and updating the entry in main memory. If any DMA operations are pending, they are completed before CQBIC gives up the CDAL bus. This prevents multiple successive map updates by the CPU from locking out DMA activity long enough to cause Q22-bus devices to timeout (in 10 microseconds).

On reads to this address range that miss the cache, the CQBIC has to latch the address and force the CVAX to retry the cycle. In this way, CQBIC can acquire the CDAL bus to fetch the entry from main memory. When the CQBIC relinquishes the CDAL bus, the CVAX retries the cycle, and the CQBIC provides the processor with the requested map entry. This retry mechanism is also used to implement the interlocked instructions in the VAX instruction set.

On all interlocked instructions, the CVAX generates one or more sequences of a read-lock cycle followed immediately by a write unlock cycle. The CVAX identifies these special locked cycles by placing a unique code on the parity lines at address time. The CQBIC recognizes the read-lock code and forces the CVAX to retry until the CQBIC can become master of the Q22-bus. Once the CQBIC has mastership of the Q22-bus, memory is effectively locked and the cycle proceeds. The CQBIC releases the Q22-bus (unlocking memory) on the next CVAX bus transaction even if it is not a write unlock cycle. This release prevents memory from staying locked if the CVAX has to abort the instruction due to an error encountered on the read-lock cycle.

Like the MicroVAX II Q22-bus interface, the CQBIC gives the CPU the highest rather than the

lowest priority when arbitrating the Q22-bus. This priority assignment reduces interrupt latency, since the processor is delayed for a maximum of one DMA transaction before being granted the bus to acknowledge the interrupt. Because the CPU accesses memory over a dedicated interconnect rather than through the Q22-bus, CPU references to the Q22-bus are very infrequent. Therefore this priority scheme does not have a negative impact on DMA performance.

To support a range of CVAX microcycle times and fixed Q22-bus timing, the CQBIC was designed to run at a fixed clock rate, asynchronously to the CPU/memory subsystem. This design made it easier for engineers to optimize performance of the slower asynchronous Q22-bus (where bandwidth is at a premium). These optimizations are made at the expense of lower performance on the faster CDAL bus (where there is extra bandwidth) due to synchronization delays.⁷

System Support Functions

The SSC contains all those functions required to support the on-board firmware, the time-of-year clock, and the console serial line. The chip provides the logic necessary to interface the two 64KB read-only memories (ROMs) containing the firmware with the BCDAL bus. Since the ROMs are organized as a 64K by 16-bit array, the SSC must generate two ROM cycles to satisfy each 32-bit CDAL bus cycle. This ROM unpacking function saves board space as well as the costs related to a 32-bit-wide ROM array.

The SSC assists in the firmware emulation of a VAX console processor by providing two address spaces through which the ROM may be accessed — the halt-mode ROM space, and the run-mode ROM space. Any I-stream read from the halt-mode ROM space protects the processor from external halt conditions and extinguishes the front panel run light. Any I-stream read outside the halt-mode ROM space, including reads from the run-mode ROM space, enables external halt conditions. Under this condition, the front panel run light is illuminated. The firmware is organized so that console emulation code is executed from the halt-mode ROM space, and diagnostics and boot code are executed out of the run-mode ROM space. The SSC also provides the firmware with 1KB of battery-backed up RAM for storage of data structures and stack space, and a register for controlling four diagnostic LEDs.

The SSC also contains a VAX standard console serial line and a VAX standard battery backed up

time-of-year clock. (The VAX standard serial line replaces the serial line chip used as the console on the MicroVAX II. The clock replaces the off-the-shelf clock chip.) Since the console control/status registers (RXCS and TXCS), console data buffers (RXDB and TXDB), and the time-of-year clock (TODR) are VAX internal processor registers, they are accessed by means of special CDAL bus cycles as described in the section The Central Processing Unit and First-Level Cache.¹

To save board space and cost, the SSC provides two programmable address strobes for decoding additional board-level registers. These address strobes decode the second-level cache control register (CACR) and the MicroVAX II-compatible boot and diagnostic register (BDR).²

To prevent the processor from “hanging” on unanswered CDAL bus cycles the SSC provides a programmable watchdog timer for the CDAL bus. The timer starts at the beginning of a CDAL bus cycle. If the timer expires before the cycle completes, the SSC asserts the error line, causing the CQBIC or CVAX to abort the cycle. This timer could not be used for all CDAL bus cycles. To do so, the timer would have to be set to a value greater than the Q22-bus timeout value (10 microseconds) so that CPU accesses to the Q22-bus would not be timed out prematurely. Moreover, the timer would have to be set to a value much less than the Q22-bus timeout value so that unanswered CDAL bus cycles would not cause Q22-bus timeouts during DMA. Since the CQBIC contains a 10-microsecond Q22-bus watchdog timer, the CDAL bus timer was set to 2 microseconds (greater than the longest CDAL bus cycle) and disabled on all Q22-bus references.

To support a range of CVAX microcycle times, the SSC was designed to run at a fixed clock rate, asynchronously to the CPU/memory subsystem. Since the performance of the functions in the SSC was not critical, the performance impact was not a concern.⁸

Hardware Interrupts

The interrupt logic is spread among three chips: CVAX, SSC, and CQBIC. The CVAX provides four interrupt request pins that correspond to standard VAX hardware interrupt request levels 14 through 17. The CVAX does not provide an interrupt-acknowledge pin. The CVAX acknowledges interrupts when the processor's priority level is below the interrupt level by generating an interrupt acknowledge cycle on the CDAL bus.

The "address" used is the level of the interrupt request being serviced. The data read is the offset of the vector within the system control block.

The SSC contains the interrupt-acknowledge pin. The SSC responds to interrupt-acknowledge cycles whenever it has an interrupt pending at the level being acknowledged. If the SSC does not have an interrupt pending at that level, it asserts the interrupt-acknowledge signal. The CQBIC passes interrupt-acknowledge cycles on to the Q22-bus only when the SSC asserts the interrupt-acknowledge signal. This interrupt-acknowledge scheme saves a CVAX pin, at the expense of requiring the devices in the SSC to have the highest interrupt priority at their level (IRQ 14).

The CQBIC uses all four CVAX interrupt request lines to support the four Q22-bus interrupt request levels. (BR4 through BR7 are connected to the pins corresponding to IRQ levels 14 through 17.) Since the Q22-bus has only one interrupt-acknowledge line, it is possible for a level 7 (17) device to steal an interrupt-acknowledge cycle intended for a level 4 (14) device. (This "steal" can occur if the level 7 device is closer to the processor and posts an interrupt after the level 4 interrupt was acknowledged but before the acknowledgment reached it.) To prevent this situation from causing a level 7 (17) device driver from running at a lower IPL, the CQBIC sets a bit that is returned along with the vector offset. This bit causes the CVAX to set the processor IPL to 17 before passing control to the driver. If the bit is not set, the processor IPL is set to the level at which the interrupt request was received. The CQBIC also adds an offset of 200 (hex) to the vector returned by the Q22-bus device so there is no conflict with existing VAX system control block entries.

Performance Relative to the MicroVAX II Processor Module

The reduction in gate delays due to the new chip technology allowed the processor microcycle time to be reduced to 90 ns (versus 200 ns for MicroVAX II) and the minimum bus cycle time to be reduced to 180 ns (versus 400 ns for MicroVAX II). The increase in the number of transistors made available by the new technology allowed the following architectural mechanisms to be used to increase performance:

- A larger prefetch buffer (12 versus 8 bytes)
- A larger translation buffer (28 versus 8 entries)
- A 1KB, 90-ns, first-level cache
- A 64KB, 180-ns, second-level cache (instead of 1MB of memory)
- Multiword transfers (longword, quadword, hexaword, and octaword versus longword)
- CPU write buffers (one longword) in the CPU, memory controller and Q22-bus interface
- Larger DMA buffers (16 words versus 2 words for writes, 4 words versus 2 words for reads)
- A 16-entry scatter-gather map cache

The combination of reduced cycle times and architectural mechanisms produced a CPU performance 3.2 times that of the MicroVAX II (as measured by the mean of the distribution of results from over 150 CPU benchmarks). Additionally, a slight increase in maximum I/O bandwidth was achieved (as measured by simulation with an ideal Q-bus master).

Reliability

Both the MicroVAX II design and the MicroVAX 3500/3600 design were subjected to extensive thermal analysis. This analysis contributed to a board layout and chip packaging scheme that would minimize junction temperatures, thereby improving reliability. Both designs also ensure a high level of reliability by using preconditioned components that have passed a rigorous qualification program.

Because of its increased complexity, the MicroVAX 3500/3600 was designed to be more tolerant of intermittent and transient failure mechanisms. ECC rather than parity is used to protect main memory, and the data path between the CPU and main memory (including both caches) is protected by byte parity. There are also four timers (three for the Q22-bus and one for the CDAL bus) to detect unanswered bus cycles. The CVAX can detect four types of CFPA errors, four types of memory management unit errors, one type of interrupt error and one type of microcode error. Errors that are detected synchronous to CPU execution are reported by means of a machine check on the same cycle on which the errors are detected. (Comparatively, the MicroVAX II reports the errors on the subsequent cycle.) Unique machine check frames or hardware error flags are provided so that the proper error recovery routine can be invoked. The recovery routines typically log the error, clear the error condition, retry the operation a

specified number of times, and continue if successful. If the routine is unsuccessful and the faulty hardware can be disabled, the system runs in a degraded mode until repaired. Otherwise, the system will crash. Errors detected asynchronously to CPU execution are reported by a high priority interrupt and are logged, but in most cases are nonrecoverable. Errors that are corrected by hardware are reported via a lower priority interrupt, so they can be logged.

Data from reliability qualification testing verified that the predominant failure mode was intermittent, suggesting that the error recovery capabilities built into the system would significantly increase the uptime of the system.

Testability

Most of the architectural mechanisms used to increase the speed of computer systems (such as caches and special purpose buffers) present testability problems. These mechanisms are almost always designed to be software transparent, which makes them invisible to diagnostic software. To solve this problem, special diagnostic modes are provided for the both the first- and second-level caches. The first-level cache diagnostic mode provides a way for the CPU to explicitly write the tag store and clear the valid bits by using selected instructions. The second-level cache diagnostic mode provides explicit access to both the tag and data stores through two blocks of I/O addresses (the cache diagnostic space and the cache tag diagnostic space). Through the cache diagnostic space, the data store can be read or written, the tag store can be written and the valid bits can be cleared. When not in diagnostic mode, cache appears in this space as high speed RAM. During power-up self-test, diagnostic code is transferred from ROM to this RAM to allow fast execution of the code without requiring that main memory be functional. Through the cache tag diagnostic space, the state of the cache tag bits, parity bits, valid bits, and several points within the cache control logic can be read.

The MicroVAX 3500/3600 processor module design also provides a diagnostic mode for main memory and a means of writing to main memory through the Q22-bus interface. The main memory diagnostic mode allows memory test times to be significantly reduced. Further, writing to main memory through the Q22-bus interface allows the scatter-gather map functionality to be tested

without the assistance of another device on the Q22-bus.²

Summary

Having met performance goals, MicroVAX 3500/3600 systems were shipping in volume within three years of the first shipments of MicroVAX II. At that time, two system packages, over twenty mass storage and communications options, three operating systems, and over 200 software products (for VMS alone) had been qualified and were available from Digital. Scores of hardware and software products were also available from third-party vendors. This offering would never have been possible without the level of compatibility that results from strict adherence to existing CPU (VAX) and I/O bus (Q22-bus) specifications.

References

1. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1987).
2. *KA650-AA CPU Module Reference Manual* (Maynard: Digital Equipment Corporation, Order No. EY-KA650-UG, 1987).
3. T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital Technical Journal* (August 1988, this issue): 95-108.
4. E. McLellan, G. Wolrich, and R. Yodlowski, "Development of the CVAX Floating Point Chip," *Digital Technical Journal* (August 1988, this issue): 109-120.
5. C. DeVane, "Design of the MicroVAX 3500/3600 Second-level Cache," *Digital Technical Journal* (August 1988, this issue): 87-94.
6. D. Morgan, "The CVAX CMCTL — A CMOS Memory Controller Chip," *Digital Technical Journal* (August 1988, this issue): 139-143.
7. B. Maskas, "Development of the CVAX Q22-bus Interface Chip," *Digital Technical Journal* (August 1988, this issue): 129-138.
8. J. Winston, "The System Support Chip, a Multifunction Chip for CVAX Systems," *Digital Technical Journal* (August 1988, this issue): 121-128.

Design of the MicroVAX 3500/3600 Second-level Cache

The MicroVAX 3500/3600 processor module, the KA650, is a CVAX-based uniprocessor that incorporates an unusual cache architecture: a two-level cache. The first level is a small fast cache on the CPU chip, and the second level is a large, somewhat slower cache on the processor module. Along with high quality and high performance, time-to-market was a crucial goal for this third-generation MicroVAX system product. Consequently, project engineers adhered to a philosophy of design simplicity for the second-level cache. Cache performance measurements support their design decisions.

The MicroVAX 3500/3600 Project

The primary goal of the MicroVAX 3500/3600 project was simple. The chip designers in the Semiconductor Engineering Group (SEG) were working on a new single-chip VAX, CVAX.¹ The chip would have its own on-chip cache and was projected to achieve a performance level three times the original MicroVAX chip used in the MicroVAX II system. The MicroVAX Development Group would work in concert with the SEG effort. Our goal was to ship a high-quality, high-performance CVAX-based uniprocessor, which would be upward compatible with MicroVAX II systems. This new product must be available as soon as CVAX chips could be produced in volume.

Given the objectives of high quality and MicroVAX II system compatibility, the remaining design goals were carefully prioritized as listed below:

1. Time to market
2. Raw computational performance
3. Memory expansion
4. Direct-memory access (DMA)/real-time performance
5. System cost and price
6. Additional functionality

The importance of quickly delivering the MicroVAX 3500/3600 to market led to a close working relationship between the engineers in SEG and MicroVAX Development. We designed and built the MicroVAX CPU and memory mod-

ules in parallel with the CVAX project, a process that relied heavily on simulation. In turn, the MicroVAX project team provided the initial debug testbed for CVAX: CVAX first booted VMS in a MicroVAX 3500/3600 system.

Overview of the KA650 Processor Module

The system functional partition (Figure 1) shows how the KA650 processor module fits into the entire computer system. The processor module communicates to mass storage, communication, and other I/O devices over the Q22-bus. Main memory connects to the processor on a private memory bus which uses both the backplane and "over-the-top" ribbon cable. A console panel carries bit rate and configuration switches, a single-digit hexadecimal display, a connector for the console serial line, and a NiCd battery for the processor's time-of-year (TOY) clock.

The module functional partition in Figure 2 shows the basic parts of the KA650 processor module. All memory traffic flows over the CDAL bus (CVAX data/address lines). Only I/O space registers reside on the BCDAL bus (buffered CVAX data/address lines).

The memory controller subsystem and the Q22-bus interface subsystem are each single chips: the CMCTL (CVAX memory controller) and CQBIC (CVAX Q22-bus interface chip).^{2,3} Most of the system support functions are contained in another chip, the SSC (system support chip).⁴ Each of these was designed in parallel with CVAX, as part of a complete CVAX chip set.

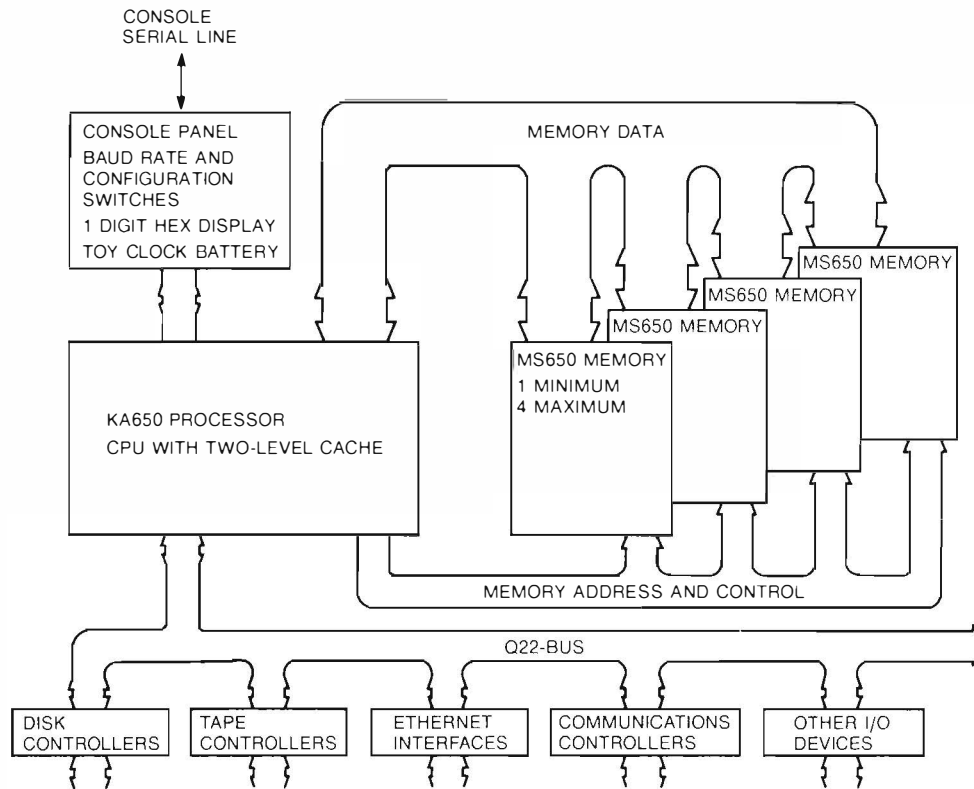


Figure 1 MicroVAX 3500/3600 System Functional Partition

The primary problem left to the KA650 module designers was to balance two key goals: to design the board-level cache for the highest performance possible and to do so without endangering the project's time-to-market goal.

Two-level Cache Architecture Description

The KA650 is Digital's first commercially available processor to incorporate a two-level cache. The first level is a small cache on the CPU chip with a cycle time of one microcycle, or 90 nanoseconds (ns). The second level is a large cache on the processor module with a cycle time of two microcycles, or 180 ns. In comparison, the cycle time of main memory system is five microcycles, or 450 ns.

The goal of each level of cache is to reduce effective memory access time on processor read cycles. At the chip level, the CVAX processor would prefer to use just one microcycle to access memory. However, the CVAX bus interface unit (BIU) requires two microcycles to access memory off the chip. To compensate for this gap, the

CVAX designers included an on-chip cache that could be accessed in one microcycle, and made the cache as large as practical. From the module perspective, CVAX can run a bus cycle as quickly as two microcycles. However, the memory system requires five microcycles to access main memory. To compensate for this second gap, the module designers included a module level cache that could be accessed in two microcycles, and made the cache as large as practical.

First-level Cache

The first-level cache is a 1 kilobyte (KB), two-way set associative cache with a quadword block size. The cache is organized as 64 rows, each row containing two sets, and each set containing 8 bytes.

Two bits in the cache disable register (CADR) select whether the first-level cache stores I-stream only, D-stream only (ordinarily used only for diagnostics), or both I-stream and D-stream references. The cache allocates a block whenever a cacheable read reference misses the cache. The CVAX BIU then generates a quadword read cycle to fill the allocated block.

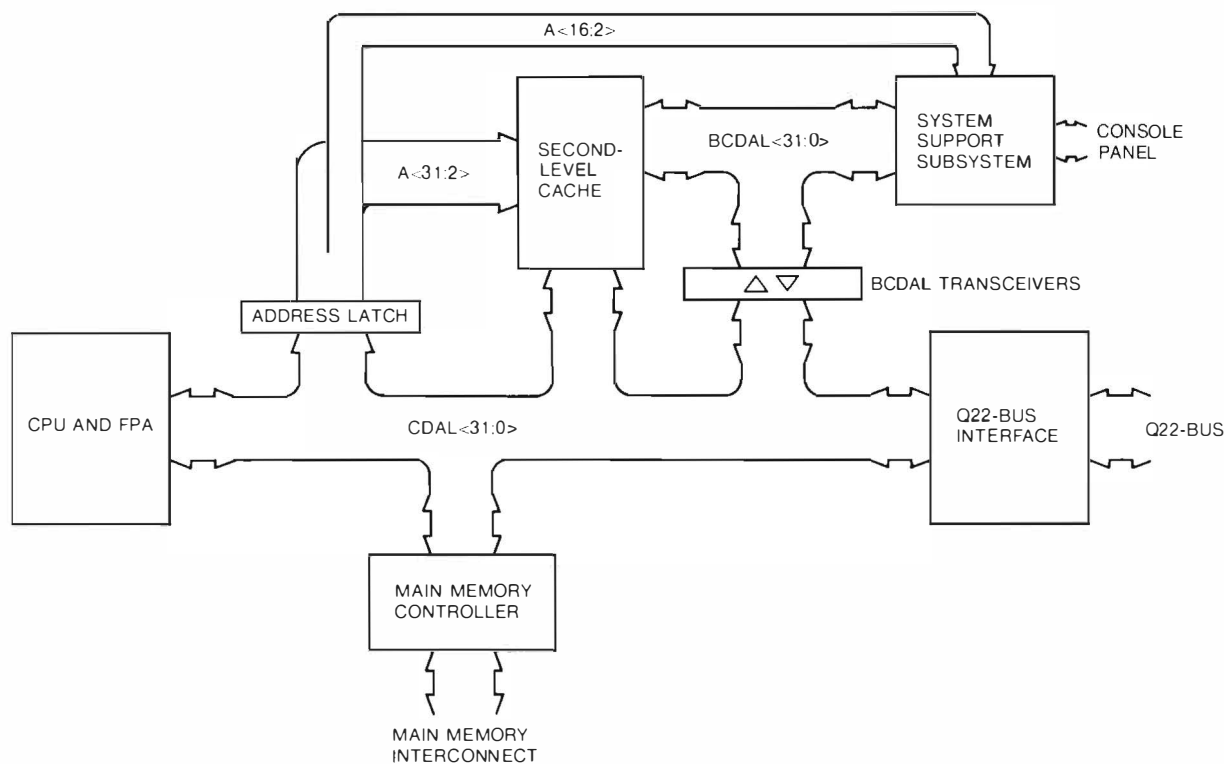


Figure 2 MicroVAX 3500/3600 Module Functional Partition

The CVAX BIU waits to determine whether a read reference hits in the cache before starting the bus cycle to access memory. This wait helps free the processor bus for use by DMA devices, but requires faster RAMs in the second-level cache.

The processor writes directly through the cache to memory. Therefore, when a cache block is allocated, the block being replaced need not be written back to memory. The CVAX BIU also incorporates a write buffer to support dump-and-run writes by the processor. If the CDAL bus is busy when CVAX needs to write, the BIU will buffer one write cycle. The buffering allows the processor to continue execution, reading from the first-level cache. Thus, some write cycles require only one microcycle.

When DMA devices write to main memory, the cache must be updated to reflect the change in main memory. Cache data that is no longer consistent with the contents of main memory is called stale data. To prevent stale data from accumulating in the cache when DMA devices write to memory, the cache will check and invalidate one or two blocks as necessary. Invalidation ties up

the first-level cache for three microcycles per quadword block and six microcycles for an octaword. However, these delays stall CPU execution only if the CPU requires access to the cache during those microcycles.

Second-level Cache

The second-level cache is a 64KB direct-mapped cache, which like the first level, also has a quadword block size. This cache is organized as 8K rows, each row containing one set of 8 bytes.

The second-level cache allocates a quadword block whenever CVAX reads a quadword that misses the second-level cache. (Quadword reads are ordinarily the result of allocation in the first-level cache. Unusual bit settings in the CADR, however, can cause the CVAX BIU to generate quadword cycles on reads without actually enabling the first-level cache.) Thus, the second-level cache will include the same kind of data as the first-level cache: I-stream only, D-stream only, or I- and D-stream references.

Instead of waiting to determine whether a read reference hits in the cache, the memory con-

troller begins accessing memory in parallel with the tag look-up in the second-level cache. If the reference hits in the cache, the memory controller will abort its response to CVAX (although the control cycle to the memory modules completes normally).

Like the first-level cache, the second-level cache also writes directly through to memory. The memory controller will perform a dump-and-run write if the write is an unmasked longword. Therefore many write cycles can complete in two microcycles. This completion time assumes the memory modules are not busy completing a previous dump-and-run write, aborted read cycle, or refresh cycle.

During DMA the second-level cache will also check and invalidate one or two blocks as necessary. These checks prevent stale data from accumulating during DMA write cycles to memory.

Design of the KA650 Second-level Cache

The importance of minimizing time taken to deliver the product to market made simplicity a high priority. For most major design decisions, we chose the simplest implementation.

Cache Speed

The cache speed was determined by the fastest CVAX bus cycle. CVAX can read or write a single longword in two microcycles (180 ns) and read a naturally aligned quadword in three microcycles (270 ns). Each added wait state costs another microcycle (90 ns). For example, a typical quadword read from main memory requires five microcycles for the first longword and three microcycles for the second longword — a total of 720 ns. Therefore the goal of the second-level cache was to allow CVAX to execute from memory with no wait states. Preliminary timing diagrams determined that to keep up with a 100-ns CVAX the cache would require 45-ns static RAMs. When later in the project KA650 module designers changed the clock speed from 100 ns to 90 ns, they also replaced the 45-ns cache RAMs with 35-ns RAMs.

Cache Size

Increasing a cache's size always improves its performance. Since high performance was a major priority, choosing the cache size was simply a matter of finding the largest RAM that would run fast enough, fit on the board, and not risk the

schedule. At the beginning of the project, we doubted that 256-kilobit (Kb) static RAMs with 45-ns access time would be available soon enough. However, we expected 64Kb RAMs to be mature when Manufacturing would need production volumes of the parts for the MicroVAX 3500/3600 system.

The 64Kb RAMs were available in three organizations: 64K by 1, 16K by 4, and 8K by 8. We could have arranged these to form a 256KB cache (using 32 64K-by-1 RAMs), a 64KB cache (using 8 16K-by-4 RAMs) or a 32KB cache (using 4 8K-by-8 RAMs). The 256KB cache would not have even fit on the module, and so was not considered. The 64KB cache would fit (requiring only slightly more module space than the 32KB cache) and was actually cheaper than the 32KB cache. So naturally we chose the 64KB cache. We then added four 16K-by-1 RAMs for byte parity.

Cache Organization

We quickly ruled out organizing the cache with more than one set. More than one set would either require too much logic or run too slowly. To get data fast enough from the correct set on a read hit would require a multiplexer and a separate set of RAMs for each set. This additional logic would take more space than we had available. Another possibility was to use a "select set" signal generated from the tag-store match signals as an address bit into the data store RAMs. This organization, however, would run too slowly.

The cache performance simulation data available to us assumed the cache was flushed on every context switch. We felt this assumption might be overly pessimistic for caches as large as 64KB. Furthermore, we expected that more realistic data would not show a large performance advantage for a two-way set associative cache over a direct-mapped cache. We therefore chose the simpler direct-mapped organization.

Block Size

When choosing the block size for the second-level cache, we again decided in favor of simplicity. We chose to make the second-level cache use the same size block as the first-level, which was already set at a quadword. At quadword block size, the second-level cache can allocate a block simultaneously with the first-level cache. The second-level cache simply captures the data from the quadword read as it comes from memory over the CDAL bus.

We had several additional reasons for not choosing either a longword block size or a size larger than a quadword. Use of a longword block size in the second-level cache complicates the control logic and potentially degrades performance. To respond to a CVAX quadword read, the cache would require two separate tag look-ups. If the first look-up hit but the second look-up missed, the cache would have to retry the bus cycle. The retry would invalidate the block in the first-level cache and waste bus bandwidth. On the other hand, use of a block size larger than a quadword would require extra data path and control to perform block fill operations.

Tag Store Organization

Once we knew the data store size (64KB), organization (direct-mapped), and block size (quadword), we could determine the organization of the tag store.

The tag store requires one row for each of the 8,192 quadword blocks of the data store. Of the CVAX 30 bit physical address, 13 address bits (bits 15 through 3) are used to select the quadword block and associated tag store row. Each tag row must store a parity bit, a valid bit, and enough of the memory address to specify where in main memory the quadword block of data came from. Since the KA650 would architecturally support no more than 64MB, address bits 29 through 26 would always be zero to access main memory. This left 10 address bits (bits 25 through 16) to be stored in the tag row. Therefore the tag store would require 8,192 words of RAM, each word consisting of 10 tag bits plus a valid bit and a parity bit.

To make this 8K-by-12 array, we used three of the same 16K-by-4 RAMs used in the data store.

We did examine the special 2K-by-9 tag-store RAMs being developed by some memory vendors. We concluded that these RAMs were too small and their availability too risky for the KA650.

Look-aside Architecture

The design of the first-level cache keeps most of the processor memory traffic off the CDAL bus. Instead of this "look-through" design, the second-level cache uses a "look-aside" architecture which simplifies the bus data path and control and improves performance on cache misses.

In the look-aside architecture, both the second-level cache and the memory controller reside on the same bus. When CVAX starts a read

cycle, the memory controller begins accessing main memory in parallel with the tag check in the second-level cache. If the cycle misses the cache, then main memory is prepared to respond as quickly as possible. If the cycle hits the cache, the memory controller senses the hit and aborts its response to the bus cycle. A drawback of this scheme is that the memory controller must still complete the control cycle to the dynamic RAMs of main memory. Consequently, the controller cannot respond as quickly as it had initially if the cache hit is immediately followed by a cache miss. We expected this penalty to be insignificant.

The alternative to a look-aside architecture would be to place the memory controller on a separate bus. The bus cycle would pass to the controller only after the cycle missed the second-level cache. This design would have improved the efficiency of main memory usage. However, this design requires additional data path and control to create the separate memory bus, and reduces processor performance by adding at least one additional microcycle to the penalty for a cache miss.

Handling of Write Cycles

We chose a simple write-through design for the second-level cache instead of a more complex write-back design. The penalty of not using write-back is reduced by the CMCTL dump-and-run write feature. When CVAX writes an unmasked longword to main memory, the CMCTL latches the address and data and terminates the bus cycle before the write to main memory is actually completed. If write cycles occur back to back (which is common for VAX processors), then the second write will be delayed while the first one completes. However, many write cycles can still complete in the minimum two microcycles.

DMA Access to the Cache

To maintain design simplicity, we decided not to allow DMA to read or write the second-level cache. This section discusses several of the possibilities we considered and rejected. These include DMA reads, DMA write-through, and a cache without valid bits.

First, we considered allowing the CQBIC (which is the only DMA device on the CDAL bus) to read from the second-level cache. However, the cache control logic is synchronous with the

CVAX clocks. The control logic design would have been significantly complicated if that logic had to respond to the CQBIC, which runs asynchronous to the CVAX clocks.

Second, the cache must recognize DMA writes to memory to prevent stale data from accumulating in the cache. We considered letting DMA write cycles write through the cache, but again concluded the timing was too complex to be practical. (Bus parity was also a concern, which is discussed in the section Cache Parity.) Instead, the second-level cache latches the address and simply invalidates one or two blocks if the address hits in the cache.

Finally, while considering DMA write-through, we thought about designing the cache without any valid bits. Power-up routines in the read-only memory (ROM) code could initialize the cache to match main memory. The cache would then remain consistent with memory unless an uncorrectable ECC (error correcting code) error was encountered in main memory. When that error occurred, the cache would simply disable read hits until the operating system could restore consistency with main memory by writing the quadword block containing the error. Of course once we decided against DMA write-through, we had to include valid bits.

Cache Parity

To improve the integrity of the second-level cache, both the data store and the tag store of the second-level cache are protected by parity.

Data Store Parity — Data store parity was simplified by taking advantage of the CDAL bus parity supported by CVAX and CMCTL. The data store simply stores and returns parity captured off the bus, and asserts CDPE (CVAX data parity enable) to have CVAX check the parity.

This parity checking scheme was another reason we rejected DMA write-through, since CQBIC neither generates nor checks CDAL bus parity.

One drawback to this simple scheme is that the processor cannot easily determine the source of a CDAL bus parity error. A CDAL bus parity error can be caused by a cache failure, a CMCTL failure, or an actual bus fault (such as open etch). This lack of isolation makes error diagnosis difficult or impossible when CVAX detects a CDAL bus parity error.

One useful feature we did not think to include was a control register bit to disable the assertion of CDPE and the subsequent parity checking by

CVAX. Such a bit would allow a machine check handler to isolate a failing bit in the data store. Without this control register bit, software can at best determine in which byte the error resides; if multiple bytes have errors, only one byte can be identified.

Tag Store Parity — The tag store parity must be generated and checked by the tag store itself. Two separate parity trees are used:

- The predictive parity tree
- The error-checking tree

The predictive parity tree generates the parity of the tag field of the address. This tree predicts what the parity stored in the RAM must be for the bus cycle to hit in the cache. Predictive parity is fast because the parity is calculated while the tag RAMs are looking up the tag. This scheme does not delay the tag comparison and is sufficient to guarantee that bad parity stored in the tag RAMs will force a cache miss. However, it is not sufficient to determine whether the parity in the RAMs is actually bad. Thus, a second parity tree, the error-checking tree, is needed.

The error-checking tree identifies bad parity in the cache tag RAMs. The output of this second tree is checked after the hit/miss decision is made, to determine whether a miss was caused by bad parity. If bad parity is detected, the cache control register error bit is set, the cache-enable bit is cleared, and an interrupt is posted to the processor. Since the bad parity forced a miss, no state is corrupted, and a process or system crash is averted.

Second-level cache tag parity covers both the 10 tag bits and the valid bit to protect against erroneously set valid bits.

Cache Diagnostic Space

Early in the project we recognized the value of being able to directly access the cache as 64KB of fast RAM. Thus we created "cache diagnostic space" in the 64MB address range from 1000 0000 to 13FF FFFF. In cache diagnostic space, the cache RAM appears as 1,024 copies of the 64KB of cache. The cache responds to all CVAX read and write cycles in this address range, effectively forcing a cache hit. For simplicity, DMA access to cache diagnostic space is not permitted.

During power-up self-test, some diagnostics are relocated from the boot/diagnostic ROM to cache diagnostic space for faster execution.

Cache diagnostic space was also useful at initial debug of the CVAX chip set. We were able to downline-load diagnostic programs through the console serial line and execute them from the cache diagnostic space. With the diagnostic programs in this cache space, we could continue debug work on the module without relying on either main memory or the Q-bus interface.

Writing to cache diagnostic space could corrupt normal cache operation by creating stale data in the cache. To prevent this, write cycles to cache diagnostic space normally invalidate the tag for that address. This invalidation also provides a simple means for flushing all or part of the cache. To simplify diagnosis of cache faults, a diagnostic mode bit in the cache control register can be set to cause writes to cache diagnostic space to set the valid bit instead of clearing it. Setting the diagnostic mode bit also clears the cache enable bit. Thus normal allocation and DMA invalidation are prevented from accidentally upsetting a diagnostic pattern being written into the cache. These features simplify the task of putting the cache in a specific state for diagnostic purposes.

Performance Measurements

Measurements of second-level cache performance bear out that the fundamental architectural decisions were sound.

The measurements were performed on a small system consisting of a KA650 CPU with 16MB of main memory, an RQDX3 disk controller with an RD54 hard disk, and a DEQNA Ethernet interface. The CPU module was modified with additional circuitry to detect various kinds of cacheable bus cycles. The system ran VMS version 4.7A. To heavily load the system with reasonably realistic workloads, we used varying combinations of three basic tasks:

- Assembling and linking a large program written in VAX MACRO
- Running a CAD program that compares the topology of two large net lists
- Copying large files (greater than 8,000 blocks) across the network

Four 16-bit counters and a logic analyzer were used to log the occurrence of particular bus cycles. For each measurement, the cache performance was monitored continuously for 5 to 30 minutes (depending on the workload and type of

bus cycle) to collect a total of 268 million sequential bus cycles of interest. For example, to study the read hit rate, the four counters simultaneously collected:

- The total number of cacheable quadword read cycles
- The number of cacheable quadword read cycles that hit in the second-level cache
- The number of cacheable quadword read cycles that missed the second-level cache (Counting both the cache hits and misses provides a useful error check.)
- The number of cacheable quadword read cycles that hit in the cache, or that would have hit if the valid bit had been set

Since CVAX gives no external indication when a memory read is satisfied by the internal cache, only reads that miss the first-level cache (and therefore generate a bus cycle) can be directly measured. Thus, it is very important to note that the read hit rate of the second-level cache alone is not the same as the read hit rate of both caches taken together as a single whole (which is beyond the scope of this paper). This is not a problem for write cycles because the first-level cache is write through.

Test Results

For the workloads tested, the read hit rate was typically 85 percent and ranged between 82 percent and 91 percent. This is what we intuitively expected: the large size of the cache would keep the hit rate high, even though the first-level cache tends to strip off much of the memory access locality.

We measured the read hit rate of the second-level cache with the first-level cache turned off, just to get an idea of how well a simple but large cache can perform. The memory read hit rate ranged between 96 percent and 99 percent when the first-level cache was turned off. This demonstrates that even a simple direct-mapped cache performs well if it is large enough. However, note that turning on the first-level cache tends to radically alter the bus traffic seen by the second-level cache. Therefore a direct comparison between hit rates with and without the first-level cache can be misleading.

The "would have" hit rate is a measure of what the read hit rate would have been if DMA write

cycles wrote through the cache instead of invalidating the cache. The modifications to the CPU module included an extra tag comparator that ignores the valid bit. Once the cache has initially filled, valid bits are cleared only by DMA invalidates. If the tag matches but the valid bit is cleared, then the cache miss was caused by a DMA invalidate and would have been a hit if the DMA cycle had written through.

The "would have" hit rate showed the benefit of DMA write through would have been negligible. The incremental improvement in hit rate was typically 0.1 percent, though in one case it rose to about 1.3 percent (copying large files over the network, with no other computational tasks). This improvement is lost in the noise when compared to the normal task-to-task variation in hit rate. Again, this is what we intuitively expected: DMA tends not to write into memory currently in use by the processor. Clearly we made the right decision to avoid the added complexity of DMA write through.

Memory write cycles were also measured for the same tasks as memory reads. However, instead of measuring the "would have" hit rate, we counted the number of cycles that took longer than two microcycles to complete. This gives us some measure of the effectiveness of the CMCTL dump-and-run write buffer.

The memory write hit rate ranged between 77 percent and 89 percent. Of all memory write cycles, 46 percent to 63 percent took longer than two microcycles (the minimum write cycle time); and 37 percent to 44 percent took longer than two microcycles and hit in the cache.

We had hoped more cycles could take advantage of the dump and run write buffer in the CMCTL. However, this performance is still good for the relative simplicity of the CMCTL write buffer. Also remember that the CVAX internal write buffer helps shield CPU performance from the delays of many write cycles. The complexity and schedule risk of adding another write buffer or designing the cache for write-back operation would not have been justifiable.

To examine the relative impact of the two-level cache on processor performance, we ran benchmarks with both caches enabled, each cache alone, and both caches turned off. Table 1 shows some typical results normalized to the performance of the KA650 with both caches turned on. Performance of the MicroVAX II is shown for comparison.

Table 1 Comparison of Benchmark Results for First- and Second-level Caches

Benchmark	Neither Cache	Second-level Cache Only	First-level Cache Only	Both Caches	MicroVAX II
HANOI	0.45	0.70	1.00	1.00	0.42
PRIME	0.68	0.81	0.97	1.00	0.24
FFT45	0.52	0.69	0.91	1.00	0.28
JACOBI	0.47	0.65	0.93	1.00	0.27
CAE2	0.51	0.69	0.95	1.00	0.31

Each cache provides a significant performance boost, but performance with the first-level cache alone is better than performance with the second-level cache alone. The faster cycle time and two-way associativity of the first-level cache outweighs the large size of the second-level cache. An extreme example of this is the Towers of Hanoi benchmark, where the performance of both caches together is no better than that of the first-level cache alone.

Conclusions

At the project close, we had met our fundamental goals. The MicroVAX 3500/3600 CPU is compatible with the MicroVAX II but delivers three times the performance — performance attributable in part to the two-level cache. And because we adhered to a simple design approach, the new system was ready to ship as soon as CVAX chip sets were available in production volumes.

References

1. T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital Technical Journal* (August 1988, this issue): 95–108.
2. D. Morgan, "The CVAX CMCTL — A CMOS Memory Controller Chip," *Digital Technical Journal* (August 1988, this issue): 139–143.
3. B. Maskas, "Development of the CVAX Q22-bus Interface Chip," *Digital Technical Journal* (August 1988, this issue): 129–138.
4. J. Winston, "The System Support Chip, a Multifunction Chip for CVAX Systems," *Digital Technical Journal* (August 1988, this issue): 121–128.

The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor

The MicroVAX 78034 chip — also known as CVAX — is a second-generation single-chip VAX microprocessor. A primary project goal was to develop a chip with three times the performance of the first single-chip VAX processor, the MicroVAX 78032. Therefore, architecture and circuit design efforts were directed toward decreasing ticks per instruction (TPI) and machine cycle time. The designers reduced the TPI by 27 percent and achieved a 90-nanosecond (ns) cycle — a significant improvement over the 200-ns cycle time of the first-generation chip. Implemented in a 2-micron CMOS process, the chip comprises six major functional units. These include the instruction queue, execution unit, memory management unit, bus interface unit, microsequencer and control store, and a unique on-chip cache.

The CVAX 78034 CPU chip is a second-generation, single-chip VAX microprocessor. This chip is the CPU of the MicroVAX 3500 and 3600 computer systems, which have approximately three times the performance of the MicroVAX II computer system.^{1,2} The VAX 6200 family of systems uses slightly faster 80-ns (speed-binned) CVAX CPU chips in a multiprocessor configuration. In this paper, we describe the CVAX chip and explain how the increase in performance was achieved.

Project Goals

The primary project goal was to develop a single-chip CPU that implemented the VAX architecture and delivered three times the performance of the MicroVAX 78032 CPU chip used in the MicroVAX II computer systems. Of the several elements in this goal, performance presented the greatest design challenge.

The performance of a CPU is inversely proportional to the product of ticks per instruction (TPI)³ and the machine cycle time. TPI depends on the performance of the system architecture. The minimum machine cycle time depends on circuit speed and on how the architecture is

implemented. In the CVAX chip, both the TPI and the machine cycle time were improved to meet the performance goal.

Much effort went into reducing the TPI. By way of comparison, the MicroVAX II system, which is based upon the MicroVAX 78032 chip, performs at approximately 11.5 TPI; whereas the MicroVAX 3600 system, which uses the CVAX 78034 chip, performs at approximately 8.4 TPI. The TPI was lowered mainly by reducing the average number of cycles required to access memory. This reduction in the number of cycles was achieved by the inclusion of the following architectural features in the system:

- A 1-kilobyte (KB), on-chip instruction and data stream cache, which is capable of a longword read each cycle
- A 64KB, second-level cache on the board, which is capable of a longword read or write in two cycles and a quadword read in three cycles
- A 28-entry translation buffer (TB), which achieves a high hit rate for virtual-to-physical address translation

Table 1 CVAX Instruction Set Architecture

Instruction Type	Number
Implemented Fully by CPU	
Integer/logical	89
Address	8
Bit field	7
Control	39
Procedure call	3
Miscellaneous	10
Queue	6
System support	11
Character string	8
Subtotal	181
Implemented by Floating Point Chip	
F floating	24
D floating	23
G floating	23
Subtotal	70
Implemented Partially by CPU	
Character string	3
Decimal	16
Edit	1
CRC	1
Subtotal	21
Implemented Fully in Macrocode	
H floating	28
Octaword	4
Subtotal	32
Total	304

Other factors influencing the lower TPI are as follows:

- More efficient microcode was implemented for some instructions. In general, most complex instructions, such as CALLx, RET, PUSH, POP, and INSV, were coded for speed rather than for space.
- Six additional instructions were implemented in microcode. These instructions are CMPC3, CMPC5, LOCC, SKPC, SCANC, and SPANC.
- The instruction decode section decodes all specifiers instead of relying on the microcode to decode some specifiers.

The machine cycle time reduction was determined in part by the technology chosen for fabri-

cation. The first-generation chip, the MicroVAX 78032 CPU, has a 200-ns cycle time and was implemented in a 3-micron NMOS process. In comparison, the CVAX 78034 CPU chip had a goal of a 90-ns cycle time and was implemented in a 2-micron CMOS process. However, only 60 percent of the improvement in the CVAX cycle time results from the fabrication process. The remainder results from architectural and circuit innovations, which are described in the section Internal Organization.

The section following presents an overview of the CVAX architecture.

CVAX Architecture

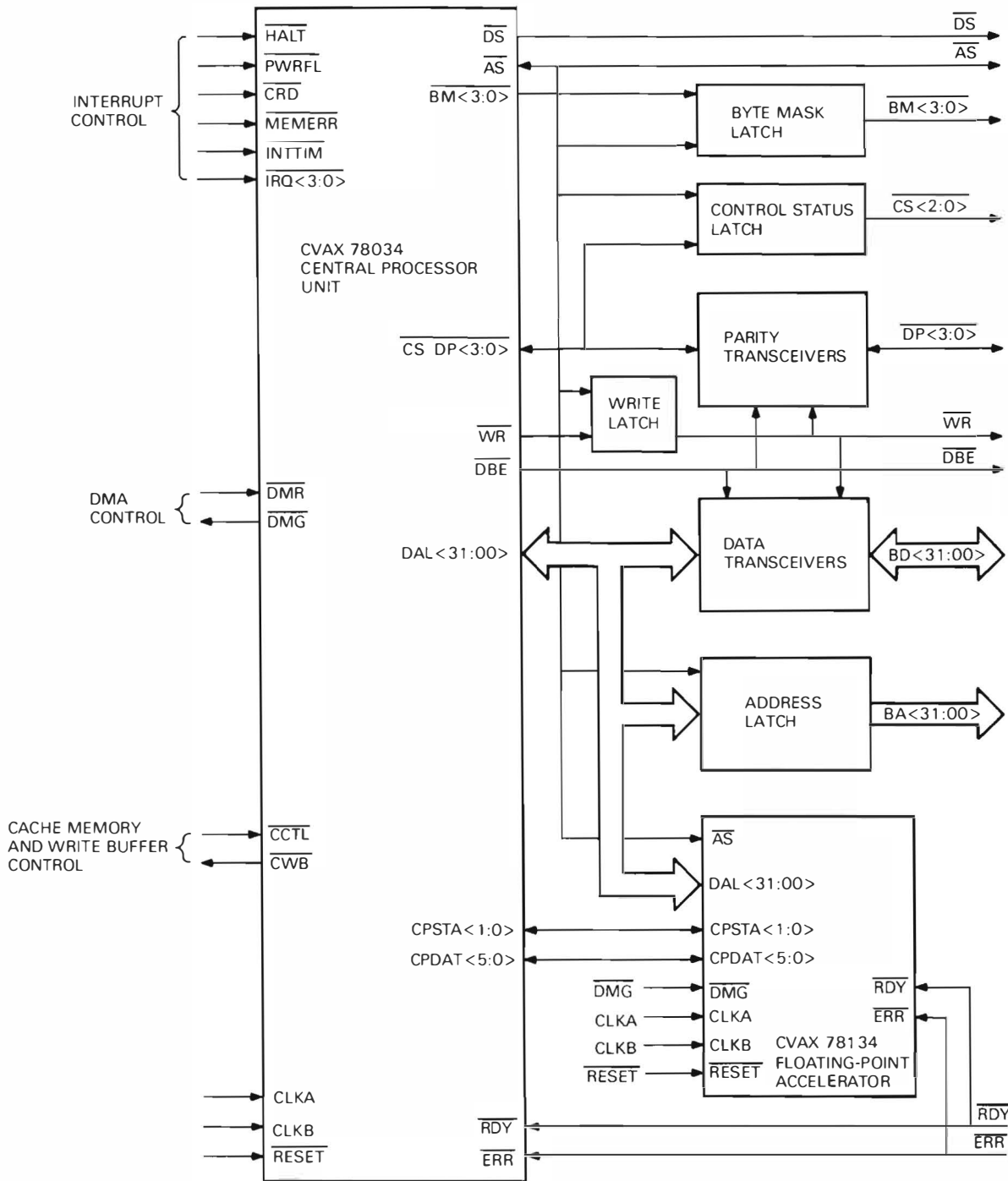
The CVAX 78034 CPU chip implements the VAX architecture, which has 16 general-purpose registers, the processor status longword, and 18 miscellaneous privileged registers. All 304 VAX instructions are supported by the system.⁴ The chip fully executes 181 instructions and provides microcode operand parsing for 21 instructions that are emulated with macrocode. The chip passes 70 F, D, and G floating point instructions to a companion floating point chip. The remaining 32 instructions are fully emulated in macrocode. Table 1 summarizes the instruction set architecture.

The chip memory management hardware and microcode provide a demand-paged virtual memory environment. The virtual memory size is 4 gigabytes, and the physical address space is 1 gigabyte.

External Interface

The CVAX bus provides a flexible interconnect protocol between all CVAX family members. The primary data bus is 32 bits wide and is time multiplexed to share addresses and data. Up to four longwords can be transferred with each address. Strokes provide timing information for synchronous and asynchronous devices. Direct memory access (DMA) request and grant signals are used to control arbitration of the data and address line (DAL) bus between the CPU and peripheral chips.

Shown in Figure 1, the CVAX 78034 CPU chip is a synchronous device on the CVAX bus. In addition to supporting the CVAX bus protocol, eight dedicated pins support a floating point coprocessor interface. These pins are time multiplexed between the CPU chip and the coprocessor chip to transfer control and status information.



MR1086-1159

Figure 1 CVAX 78034 External Interface

A clock chip generates pairs of 180-degree phase-shifted clock signals that are distributed to all synchronous MOS components in the system. The clock also generates auxiliary pairs of clocks that can be used by any non-MOS components in the external interface. Separation of the clocking for MOS and non-MOS elements provides better skew control for the critical MOS clock signals.

Microarchitecture

The CVAX 78034 CPU chip has some pipelining and is microprogrammed. The chip comprises six major functional units:^{5,6,7}

- Instruction decode and prefetch queue (I-Box)

- Execution unit (E-Box)
- Memory management unit (M-Box)
- Bus interface unit (BIU)
- Cache
- Microsequencer and control store

The photomicrograph in Figure 2 and the block diagram in Figure 3 illustrate all functional units on the chip.

Internal Organization

This section describes the six major functional units of the chip. As noted earlier, the emphasis here is on those aspects of the design that enhanced the machine's performance. In addition,

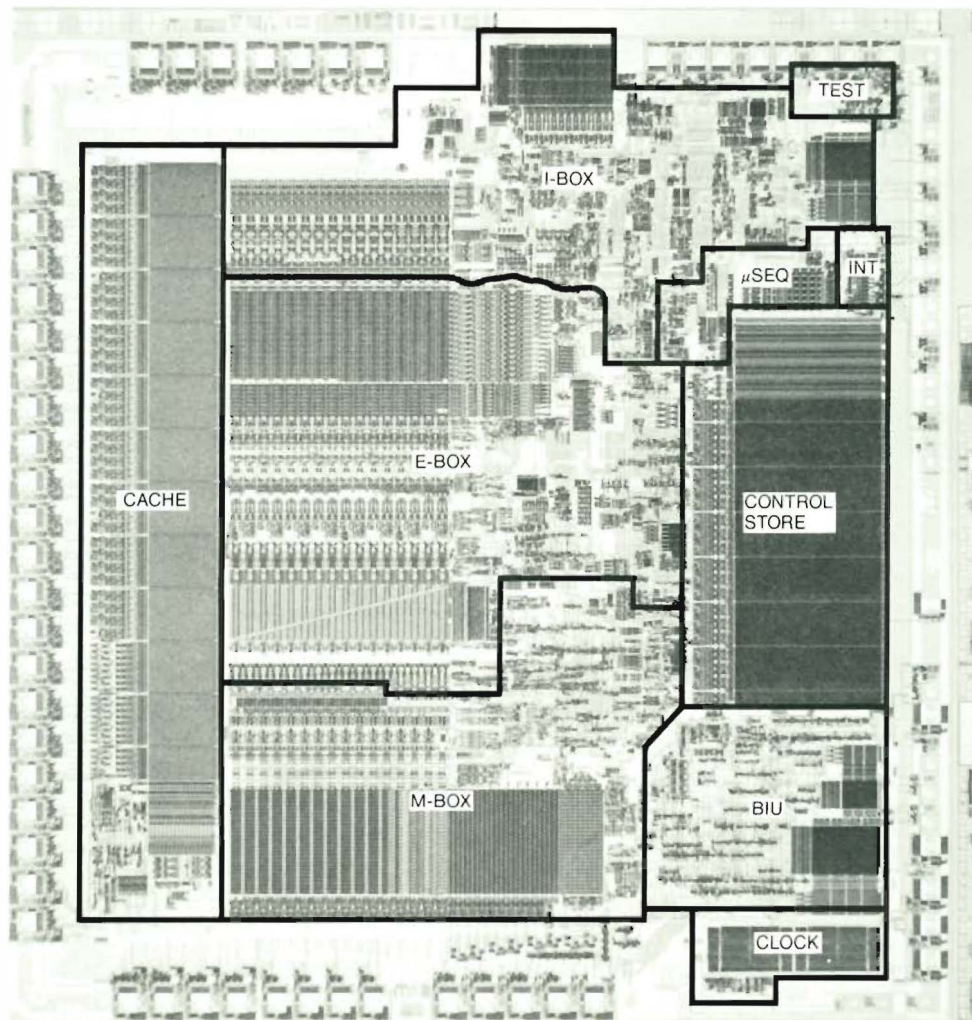
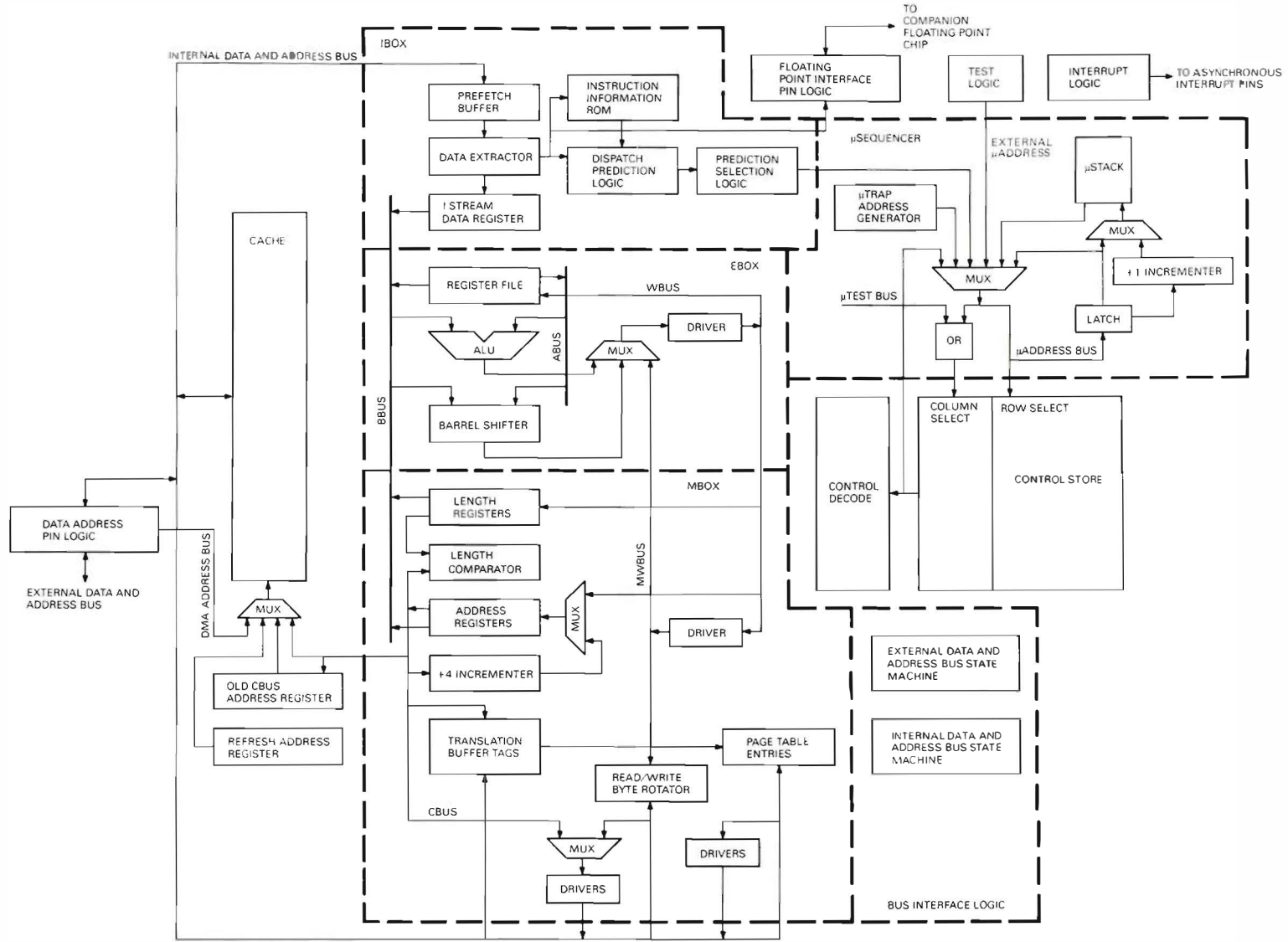


Figure 2 Photomicrograph of the CVAX CPU Chip



MD6870672A

Figure 3 CVAX CPU Block Diagram

at the end of this section we discuss the design approach taken to build in chip testability.

The flow of information between all functional units on the chip is synchronized by four on-chip clock phases of nominally equal duration. All circuits were designed to function with the partial phase overlap or underlap that can result from external clock skew and variations in the fabrication process.

Instruction Decode and Prefetch Queue

The instruction decode and prefetch queue, the I-Box, controls macroinstruction sequencing and instruction stream prefetching.⁸ During a microcycle, the I-Box determines what the next microcode dispatch will be, based on the instruction stream data and the current processor state.

The CVAX I-Box is designed to generate the microcode dispatch address for every specifier flow. This design differs from the MicroVAX CPU 78032 chip design; there, the I-Box provides the dispatch address for just the first two specifiers of a macroinstruction and relies upon the microcode to generate the dispatch address for additional specifier flows at a performance cost of one microcycle per specifier.

Primary subsections of the CVAX 78032 I-Box include the instruction decode read-only memory (ROM), the dispatch programmable logic array (PLA), and the prefetch queue.

The instruction decode ROM (IROM) contains the information about VAX macroinstructions that is required to parse the instruction stream. The IROM determines the number of specifiers for an instruction, the sizes of its operands, and a partial microaddress for the execution microcode of the instruction.

The dispatch PLA examines I-Box state, instruction stream data, and other microprocessor states to predict the next hardware-supplied microaddress for the microsequencer. This PLA is self-timed and evaluates in slightly under one clock phase.

The I-Box instruction prefetch queue operates in parallel with the instruction execution hardware on the chip. Whenever a longword in the instruction prefetch queue is empty, the I-Box issues a request to the M-Box to read the next aligned longword in the instruction stream. If the M-Box and BIU are not doing some other read or write operation, they will fetch the requested

longword and send it to the instruction prefetch queue.

When a microinstruction that loads the program counter register is detected, for example, during a branch instruction, the prefetch queue is flushed. A new instruction must then be fetched before the processor can proceed.

Up to three prefetched longwords of instruction stream data can be queued by the prefetch queue. In addition, the prefetch queue rotates the instructions to bring the opcode to the front and extracts in-line instruction stream data for use by the E-Box.

Execution Unit (E-Box)

The main functional blocks in the execution unit, the E-Box, are the register file, program counter (PC), constant generator, shifter, and arithmetic and logical unit (ALU). The data path has two precharged 32-bit read buses, called the A and B buses, and a static write bus, called the W bus. The functions performed by the E-Box during a cycle are determined by the current microinstruction and internal state. Following are descriptions of each of the main functional blocks.

The register file contains 31 single-read-port/single-write-port registers and 8 dual-read-port/single-write-port registers. The register file is used in the data path where compact layout is especially important. Therefore, to save chip area the register file cell was designed using an NMOS pass gate rather than a full transmission gate.

The 32-bit PC is located in the data path along with the program counter adder. This adder is used to increment the PC as macroinstructions are parsed.

Literals can be introduced into the data path by conditionally discharging the precharged A or B bus lines.

The shifter function is implemented as a data extractor rather than a full shifter, which would require more hardware. The extractor can extract 32 contiguous bits from a 64-bit field. When the values on the input buses are identical, the high-order bits appear to wrap around to the low-order positions, thus mimicking a full shifter.

The shifter has the two 32-bit precharged read buses (the A and B buses) as inputs and a 32-bit output. The shifter is implemented using NMOS transistors. The control diagonals are run in polysilicon strapped by metal at both ends.

Because the RC delay in asserting the control lines is long, the control lines are driven before the input data is valid. The inputs are then conditionally pulled low, discharging the outputs.

The ALU in the data path is capable of addition, subtraction, and a variety of logic operations. The ALU also includes a 1-bit left/right shifter and additional logic to support multiply and divide operations. The ALU is implemented using a carry-lookahead scheme with propagate and generate logic.

The ability to read the register file, do an ALU or shift operation, and write the result back into the register file all in one cycle is important to the machine's performance. This critical path was alleviated by partially overlapping the register file write with the next register file read. The partial overlap introduces a race between the write and the read, but the circuit delay in asserting the read select line is sufficient to ensure that the race is always won without extending the cycle time.

Memory Management Unit

When memory management is enabled, the M-Box uses a fully associative translation look-aside buffer (TB) to translate virtual addresses to physical memory addresses. The major design goal for the M-Box was to achieve a TB miss rate that was one third that of the MicroVAX 78032 CPU chip. Consequently, we increased the size of the TB from 8 to 28 page table entries (PTEs). Furthermore, we used a more efficient microcode routine to reduce the number of cycles required to fetch a PTE on a TB miss. A PTE is composed of the higher order bits of the physical address, the access protection field, and other memory management information. In the MicroVAX 78032 CPU chip, a least-recently-used algorithm was employed to replace the PTE on a TB miss. However, the implementation of this algorithm requires complex circuits and a large amount of chip area as the TB size is increased. For this reason, we implemented a simpler but almost equally efficient not-last-used algorithm in the CVAX 78034 CPU chip.

To realize a single-cycle cache read operation, both a virtual-to-physical address translation and a check of the access protection field of the PTE must occur in just two clock phases. However, there is not enough time to check the access protection field after the translation has

occurred. Therefore, all access protection fields in the TB are simultaneously compared to the current access type while the translation is in progress. This scheme requires that the access protection field be fully decoded before it is stored in the TB.

In addition to interacting with the cache, the M-Box interfaces with the BIU and the I-Box. The M-Box contains three registers: the virtual address (VA) register, the virtual address prime (VAP) register, and the virtual instruction buffer address (VIBA) register. After a data read or write using VA or VAP, VAP is loaded with the most recently used address plus four. In this way, VAP can quickly generate sequential longword addresses. During a memory operation, the M-Box sends the address to the cache and BIU. The M-Box will forward data from the E-Box during the next cycle if the operation is a write, or capture data for the E-Box if the operation is a read.

Whenever there is space available for a longword in the I-Box prefetch queue, the I-Box requests instruction stream data. If the M-Box does not decode a memory read or write request from the current microinstruction, it services the instruction stream read request using the virtual address stored in the VIBA register. After a prefetch reference succeeds, the VIBA register is incremented by four in preparation for the next prefetch.

Bus Interface Unit

The bus interface unit, the BIU, controls external chip operations, internal cache access and refresh, and arbitration for the internal data and address bus. The BIU contains two state machines.

- The internal state machine controls the arbitration for the internal data and address bus (IDALs).
- The external state machine, controls the arbitration for the external pins and DALs.

The design goal was to achieve a single-cycle read operation for hits to the internal cache and a two-cycle write operation for an ideal memory subsystem. In addition, better system reliability is achieved by providing parity protection on all the external data transfers and internal cache read/write operations.

To accomplish a single-cycle read operation, the two state machines were implemented as self-timed PLAs that require just one phase to evaluate. The separation of control operations between the two state machines allowed the PLAs to operate in different phases. Read/write-related, internal time-critical signals are generated by the internal state machine. This state machine evaluates first, stalls the CPU if necessary, controls the cache, and sets states for the external state machine. Time-critical external strobes are controlled by the external state machine. The external state machine operates next, controls the termination of external operations, clears the internal state machine flags, and grants control of external buses and strobes to external devices. On a cache miss, the external state machine unconditionally drives the external read data to the M-Box or the I-Box, and a phase later the state machine validates the data. This scheme made it possible to service the next microinstruction while the previous one was completing.

The BIU also controls all memory transactions. A memory read operation is performed in one cycle if there is a hit in the internal cache and no cache parity error is detected. However, when a cache miss occurs during a read operation, a two-longword block in the cache is allocated to store the data, which must now be read from memory. The BIU stalls the CPU until the first longword of data is received. The BIU initiates the external read cycle, sending the address of the first longword to the external memory system. When the first longword of data is received, the BIU sends it to the cache and E-Box or I-Box, and uninstalls the CPU. The fetch of the second longword is overlapped with other chip activity to minimize the effective memory access time. The second longword of data is written into the alternate longword in the allocated quadword (two longword) cache block. The cache block is validated only if both longwords in the block are fetched successfully.

The BIU contains a longword write buffer which supports a dump-and-run write mechanism. Chip activity, including cache reads, can proceed in parallel while the BIU is waiting for the completion of a write operation. The BIU may have up to three different operations in progress at once: a write to memory, a read from memory, and an internal cache entry invalidation. Descriptions of these operations in the BIU follow.

While a write to memory is awaiting completion, the internal state machine can service read

requests. If the read reference misses the cache, it is queued and serviced only after the write operation completes. This overlapping of read and write operations reduces the number of memory stall cycles, resulting in a lower TPI.

To facilitate support for multiprocessor applications and DMA activity, the BIU provides a protocol for internal cache coherency. To activate this function, an external device first gains ownership of the external address and data bus by means of the DMA request and grant protocols. The device then presents an address, qualified by certain strobes, to the processor. The processor latches the address and then performs a cache look-up. If a cache hit occurs, the matching cache entry will be invalidated.

Eight pins are dedicated to the floating point interface. To optimize the operand transfer rate between the CVAX 78034 CPU and its floating point processor, both chips read the floating point operands from memory simultaneously.

Cache

The goals for the design of the internal cache were twofold: to reduce the memory access time to one microcycle for data that is resident in the cache; and to minimize the number of cache references that miss the cache.

To achieve the one-microcycle access time, the internal cache is designed to perform the cache look-up in parallel with the translation buffer look-up. This scheme uses the 9 virtual address bits that do not change during the address translation process to index into the array. Because the cache look-up and translation buffer look-up are performed in parallel, the data for the selected cache entry is ready when the translated address is being latched into the tag comparator. The cache tag is then compared to the translated address. If a match occurs, the data is driven onto the IDAL before the end of the cycle.

To achieve our second goal — minimization of the number of cache misses — we used a two-way set associative cache with a block size of 8 bytes. This two-way set associative cache was designed to meet both performance and chip size requirements. First, a random replacement algorithm was selected to reduce circuit complexity with a minimal impact on cache performance. With reference to chip size, we determined that a cache size of 1KB was the largest that could be used. In addition, the cache is designed so that it can be configured by software to act as an instruction-only cache or as an instruction and

data cache. The instruction-only option was provided to simplify hardware in multiprocessor systems where the designers do not want to deal with DMA invalidates.

The cell chosen to implement the cache array is a one-transistor (1T) dynamic RAM. The 1T cell, illustrated in Figure 4, was chosen because of its small area. A comparable array design with either a four-transistor dynamic RAM or a six-transistor static RAM cell would have required 2.4 to 3 times as much area. The storage capacitance of the 1T cell is 110 femtofarads, resulting in a bit-line to cell-capacitance ratio of 8 to 1. With a folded bit-line structure and the use of a dummy cell (which stores half the charge of the storage cell), a voltage differential of 200 millivolts was realized at the sense amplifiers. Because of the dynamic nature of the 1T cell, a refresh counter, composed of linear feedback shift registers, was designed to control which row is refreshed during idle cache cycles.

We designed byte parity into the cache to detect data corruption resulting from either soft or hard errors. A study was done to determine the soft error rate of the cell. The soft error rate for the cache array was found to be 10 FITs, where 1 FIT is equal to 1 failure in one trillion operating hours. To protect against data corruption due to minority carrier injection, the array is surrounded by a deep N-type implant ring.

The CVAX CPU chip is the first microprocessor in the industry to include an on-chip dynamic 1T cell cache.

Control Store and Microsequencer

The operations and interactions of the five functional blocks described so far are all controlled by microcode in the control store. The microsequencer supplies the microaddress to the control store. The control store contains 1,600 words of read-only memory. Each 41-bit word is divided into a 28-bit field, which controls the execution sections of the chip, and a 13-bit field, which controls the microsequencer. Control store access is achieved in less than three clock phases.

The control store is organized into 200 rows of 8 words each. H-shaped cells, 7 by 8 microns in size, are used to implement the array.

A microaddress is supplied to the control store by the microsequencer by means of the 11-bit microaddress bus (bits 10 through 0). Eight of these bits, 10 through 4 and 0, select one of the

200 rows. Selection of a row causes all eight words to be driven onto the precharged bit lines which form the inputs of an 8 to 1 multiplexer. The three remaining microaddress bits, 3 through 1, choose one of these eight microwords to be driven onto the microinstruction bus. The final value of bits 3 through 1 can be modified by values on the microtest bus. This 3-bit bus conveys state information from other sections of the chip to the microsequencer. In this way, various processor states may be polled to enable up to an eight-way microcode branch.

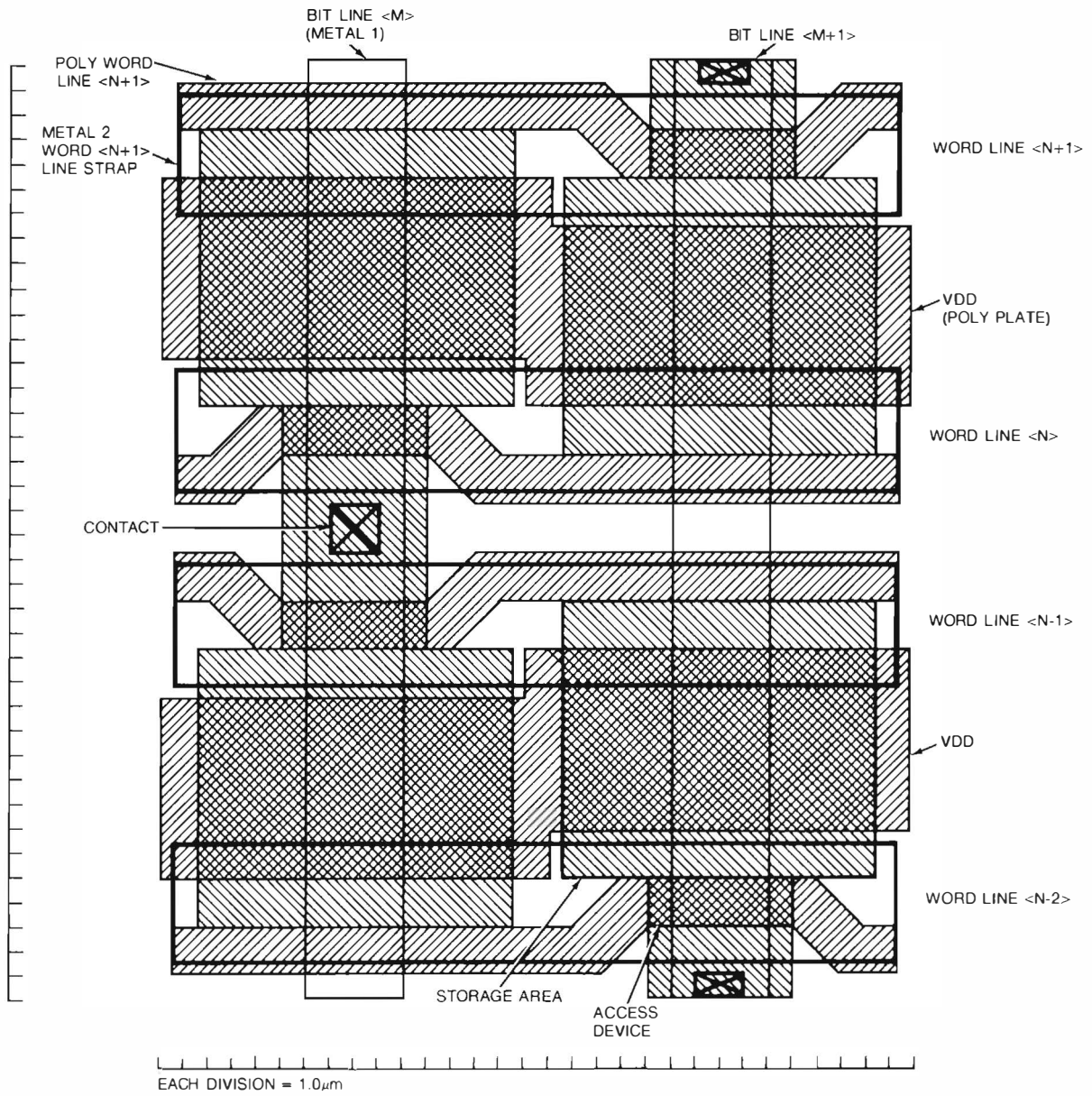
The primary function of the microsequencer is to supply microaddresses to the control store. The microsequencer selects a microaddress based on microcode control and external control from the testability logic. In addition to generating microaddresses, the microsequencer receives exception request lines from other sections, prioritizes these requests, and generates base addresses for microcode exception service routines. These base addresses can be modified by the section signaling the exception by means of the microtest bus.

The microsequencer contains a last-in-first-out (LIFO) queue of eight microaddress entries called the microstack. A latched copy of the microaddress bus is stored on the microstack when a microcode exception occurs. Once the exception has been serviced, this latched copy allows reexecution of the microinstruction that caused the exception. In the case of a microcode subroutine call, the current microaddress is incremented and stored on the microstack. This forms the address when returning from the subroutine.

Testability Issues

As a complex microprocessor chip, the CVAX 78034 CPU chip has some difficult testability issues. A large number of internal state bits and buses are not normally visible at the pins of the chip. Early in the design process, techniques such as level-sensitive scan design (LSSD) and built-in self-test were eliminated as possible testability strategies. Both of these strategies would have had a significant impact on chip area and performance. Instead, an ad hoc method of design for testability was developed.⁹

The design for testability strategy has two main themes: (1) make maximum use of existing hardware for test observability and controllability, and (2) add special test hardware to those areas








- KEY:
-  N+ SOURCE/DRAIN DIFFUSION
 -  METAL 1
 -  METAL 2
 -  POLYSILICON
 -  METAL 1 CONTACT

Figure 4 Cache 1T Dynamic RAM Cell (Four Cell Shown)

of the chip where observability or controllability would not otherwise be possible.

The chip already had some important features that could be exploited.

- The chip is controlled by the microcode contained in the control store. Thus, it is an obvious candidate for controlling the chip when in test mode.
- Many of the internal registers are readable and writable from the internal buses. By transferring this read and write data to the main bus that connects to the pins (the DALs), much of the internal state can be observed and modified.
- The interface for the floating point coprocessor chip contains a mode that broadcasts a value from the internal cache or register file to the pins. This mode is also used during test for cache and register file observability.

These features alone were not enough, however, and some specialized test hardware had to be added.

- To make use of the chip microcode in test mode, it is necessary to be able to externally choose the addresses of the microword to be executed. Thus, a test mode was added to the microsequencer. In this mode, the microsequencer ignores its normal choice for the microaddress and uses the value from a group of pins.
- The cache is difficult to test in its normal operating mode. To overcome this, a special cache diagnostics mode was developed.
- Some special test microcode was added to allow more efficient testing of some areas.
- A few major internal buses were not observable. Dual mode linear feedback shift registers (LFSRs) were added to these buses: the output of the I-Box instruction decode ROM, the microinstruction bus, and the microtest bus.

The cache refresh address counter is also implemented as an LFSR.

The dual mode LFSRs allow the data bus to be captured and scanned out serially. Alternatively, the data can be compressed every cycle using the linear feedback technique. The outputs of the LFSRs are inputs to another LFSR that combines

the data to a single-bit output stream. In this manner, all of the LFSRs may be observed at once. In addition, all of the LFSR outputs are fed into a multiplexer that allows any one of the registers to be observed.

The test logic requires only one dedicated test pin to select test mode and uses less than 2 percent of the chip area. Moreover, inclusion of this logic does not affect chip performance. When in test mode, 3 to 15 other pins are redefined for test functions. A 4-bit test-mode configuration register selects which of the LFSRs is to be observed, whether the LFSRs will be in scan or compress mode, and whether or not test broadcast mode is enabled.

The Role of Simulation and Modeling

Complexity was managed and detailed circuit behavior was predicted through the use of models and simulation. During the design, the chip was modeled at five levels of abstraction. As the design progressed from concepts to implementation, the level of abstraction was refined to reflect the increasing detail of the design.

Choosing the Microarchitecture

The performance model was the earliest and the most abstract of all the models. The performance model was used to predict the machine's performance and to quantify the speed advantage of the various microarchitectural options under consideration. Written in PL/I, the performance model was driven by trace files. These files consisted of streams of opcodes and operand specifiers derived by running typical VAX applications programs. The pseudo-microcode contained in the model approximately modeled memory request patterns and microinstruction counts for each type of VAX instruction. As we had planned, the performance model did indeed help predict the machine's TPI. Moreover, the model also helped identify performance bottlenecks in the microarchitecture.

As noted in the section Project Goals, performance is inversely proportional to the product of the TPI and cycle time. Specifically, the cycle time depends on the delay through the critical speed circuits. Therefore, to identify the critical circuits and determine the propagation delays through the circuits, we carried out cycle time feasibility studies. SPICE, a circuit-level simulator, was used in these studies. With the chip die

size as a given requirement, we determined the microarchitecture of the machine by selecting those features that minimize the product of TPI and the cycle time.

Verification of the Microarchitecture

Once the microarchitecture was defined, a detailed specification was written for each section of the chip. Next, an abstract behavioral model was written to verify that the specification described a VAX CPU. Much more detailed than the performance model, this model was controlled by microcode, ran real VAX code, and closely modeled the major chip buses, global signals, and clocks. The model was written in Digital's DECSIM behavioral modeling language. Many microcode and microarchitecture bugs were identified and fixed as a result of this behavioral model testing.

Logic and Circuit Design

The detailed logic and circuit design began while the abstract behavioral model was being written. During this phase of the design, SPICE simulations were used extensively to predict circuit behavior. Because SPICE simulates transistor behavior in detail, it requires a large amount of computer resources. Consequently only critical circuits were simulated and these were often simplified to contain only the essential elements. Circuit simulations typically involve tens of transistors rather than hundreds or thousands.

Verification of Logic — Gate Level

The abstract behavioral model had been used to verify the specification. Now it was necessary to verify the implementation of the specification. To make this verification, we wrote a schematic-level behavioral model that captured the logical and timing characteristics of every schematic. Almost every node was modeled explicitly. This essentially gate-level model was also written in the DECSIM language. The model identified many logic and timing bugs, especially between schematics designed by different engineers.

The schematic-level behavioral model was subjected to intensive verification because it offered a good compromise between implementation detail and simulation efficiency. This model of the CVAX 78034 CPU chip was used by the system designers in other design teams to model the interaction of the CPU with other chips in board designs.

Verification of Logic — Transistor Level

The DECSIM simulation tool also supports MOS transistor-level modeling. We used this tool as a switch-level simulator, that is, we modeled transistors as open or closed switches. The model was automatically generated from the schematic database.

This level of modeling reflected the true behavior of the schematics with greater subtlety than the schematic-level behavior model. However, this model was not nearly as computationally efficient as the behavioral model.

DECSIM MOS modeling identified sequencing errors, charge sharing problems, sneak paths, and race conditions that the more abstract models had failed to detect.

Physical Technology

The CVAX 78034 CPU chip is implemented in a P-EPI, N-well CMOS (complementary metal-oxide-semiconductor) process developed in-house. The process has two layers of aluminum interconnect and a single layer of polysilicon. The critical process dimensions and chip characteristics are summarized in Table 2.

The chip contains 180,000 transistor sites with 134,000 actual transistors, and measures 9.7 mm by 9.4 mm on a side. (See Figure 2.) It is packaged in an 84-pin surface-mountable ceramic chip carrier with 50-mil leads, uses a single +5 volt supply, and has a worst-case power dissipation of 1.5 watts.

Table 2 CVAX Chip Process

Fabrication Process	
Fabrication process	CMOS
Gate oxide	300 Å
Substrate	N-well in P-EPI
Device types	N-channel enhancement MOSFET; P-channel enhancement MOSFET
Interconnect Pitches (Line/space Drawn)	
Polysilicon	2 micron/2 micron
Metal 1	4 micron/2 micron
Metal 2	5 micron/2 micron
Contacts	2 micron/2 micron

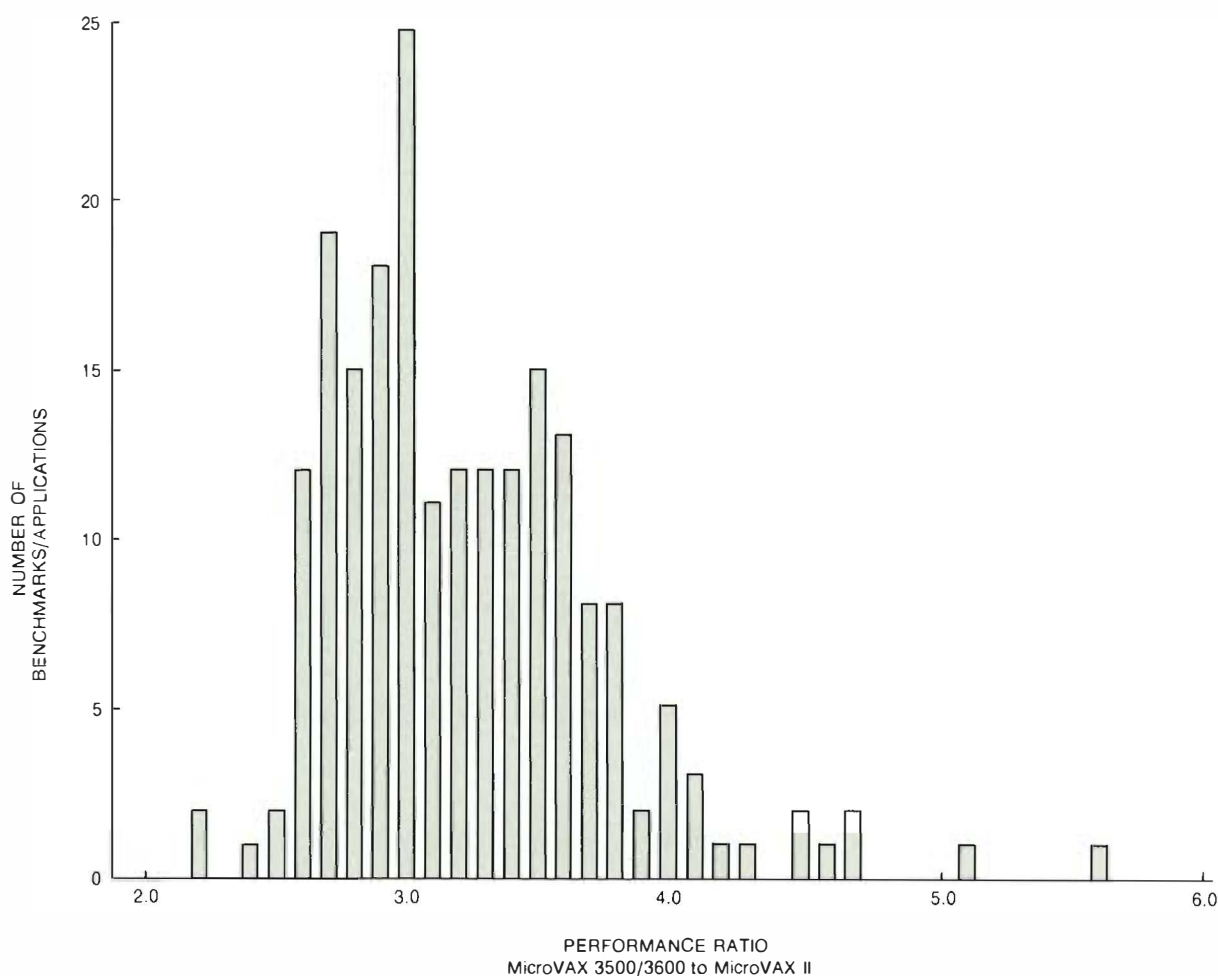


Figure 5 MicroVAX 3500/3600 and MicroVAX II System Benchmark Comparison

Summary

The CVAX 78034 CPU chip met the project design goals. Depending on the benchmark or application program being run, the performance of the MicroVAX 3500/3600 systems is 2.6 to 4.1 times that of the MicroVAX II computer. (Refer to Figure 5.) This performance increase was achieved by reducing both the TPI and the machine cycle time.

The main factors influencing TPI are the 1KB, on-chip cache; the 64KB on-board cache; and the 28-entry virtual-to-physical address translation buffer. The cycle time was reduced as a result of the advanced process technology chosen and the architectural and circuit innovations made by the design team.

Acknowledgments

The authors wish to acknowledge the technical contributions of D. Archer, D. Bhavsar, W. Bidermann, S. Carroll, D. Deverell, J. Keefe, S. Martin, A. Olesin, S. Persels, J. Reinschmidt, L. Rozek, P. Rubinfeld, M. Schenstrom, D. Schumacher, B. Supnik, J. St. Laurent, T. Thrush, and B. Worster.

Notes and References

1. D. Dobberpuhl, et al., "The MicroVAX 78032 Chip, A 32-Bit Microprocessor," *Digital Technical Journal* (March 1986): 12-23.

2. J. Beck, et al., "A 32-Bit Microprocessor with On-Chip Virtual Memory Management," *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (1984): 178-179.
3. To compute clock ticks per instruction (TPI), typical application programs and benchmarks are first run. Then the number of clock cycles required to execute these programs is divided by the total number of instructions executed. The result of the computation is the TPI.
4. *VAX Architecture Handbook* (Maynard: Digital Equipment Corporation, Order No. EB-19580, 1981).
5. D. Archer, et al., "A 32-Bit Microprocessor with On-Chip Instruction and Data Caching and Memory Management," *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (February 1987): 32-33, 329.
6. P. Rubinfeld, et al., "The CVAX CPU, A CMOS VAX Microprocessor Chip," *Proceedings of the 1987 IEEE International Conference on Computer Design: VLSI in Computers and Processor* (October 1987): 148-152.
7. D. Archer, et al., "A CMOS VAX Microprocessor with On-Chip Cache and Memory Management," *IEEE Journal of Solid State Circuits*, vol. SC-22, no. 5 (October 1987): 849-852.
8. D. Archer, "The Instruction Parsing Microarchitecture of the CVAX Microprocessor," *Proceedings of the 20th Annual Workshop on Microprogramming* (December 1987): 147-153.
9. D. Bhavsar and D. Miner, "Testability Strategy for a Second Generation VAX Microprocessor Chip," *International Test Conference* (September 1987): 818-825.

Development of the CVAX Floating Point Chip

The CVAX floating point accelerator (CFPA) chip is a CMOS floating point coprocessor for the CVAX system. The purpose of the CFPA project was to provide gains in floating point performance equal to those of the CVAX CPU for integer performance. Combined with an aggressive schedule, the primary goals required the CFPA chip to perform at three times the level of the previous generation MicroVAX floating point unit (FPU) and to be complete two years after delivery of the MicroVAX II system. Designers obtained a performance gain of only 25 percent through base technology improvements. Consequently, most gains are achieved through the use of a multiplier array, improved arithmetic algorithms, and a fast and efficient interface with the CPU.

Functional Overview

The CFPA VLSI chip is the companion floating point processor for the CVAX CPU. The chip's hardware structures and algorithms provide high overall system performance. In all, the chip executes 76 instructions.

The CFPA supports

- Three VAX floating point data types: F_floating, D_floating, and G_floating
- Floating point calculations, which include a polynomial evaluation instruction
- Integer multiply and divide instructions
- Conversion between integer and floating point data types
- Complete detection of all exception conditions

The CFPA operates synchronously with the CPU at speeds of 80 and 90 nanoseconds (ns) per cycle. Opcode, control, and status information is communicated between the coprocessor and the CVAX by means of a dedicated 8-bit bidirectional coprocessor bus.

Table 1 lists the CFPA physical characteristics.

CFPA Project Goals

The two main goals of the CFPA chip design project were (1) to provide the CVAX system with an improvement in floating point performance to

Table 1 CFPA Physical Characteristics

Number of transistors	65,000
Package	68-pin surface-mountable chip carrier with 50-mil lead spacing and heat sink
Die size	7.3 mm × 9.1 mm
Power dissipation	1 W
Fabrication process	2 micron drawn, N-well, dual aluminum CMOS

equal the central processor chip's expected performance level for integer operations, and (2) to adhere to the same development schedule set for the CVAX CPU chip. Specifically, these goals required instruction execution times to be three times faster than the MicroVAX FPU on average. Further, the schedule allowed little time to achieve these significant performance gains; the design would have to be completed only two years after the MicroVAX II system design.

In order to improve computer performance, the clock frequency and/or the amount of work completed in a cycle must be increased. The CVAX CPU uses the improved speed characteristics and greater density of the CMOS process to reduce the clock cycle time from 200 ns in the MicroVAX II design to 80 or 90 ns. A pipelined architectural approach was necessary to achieve

this reduction. In particular, while the arithmetic and logic unit (ALU) operates on one microinstruction, the register file is free to access data for the next microinstruction. This improvement allows more work to be completed in each microcycle and offers a reduction in the cycle time as well.

The previous generation floating point design, used in the J-11 FPA as well as the MicroVAX II and VAX 8200/8300 systems, already pipelined register file access with ALU operations. This pipelining was necessary to allow a 100-ns cycle time — twice the frequency of the companion CPUs — in the ZMOS process technology. Since the pipelined register/ALU operation was already achieved, the improvement in cycle time for the CFPA is limited by the speed of the ALU and does not benefit from additional pipelining. The improved technology allowed for an ALU implementation that provides a 20 percent decrease in cycle time, matching the CVAX microcycle. Therefore, the necessary performance increases for the CFPA would not be created by scaling the cycle time. Instead, CFPA designers would make improvements in the amount of work done per microcycle and in the interface between the processor and the floating point chip. This interface is described in the following section.

An overview of the chip's overall performance is presented in the section CFPA Performance at the end of this paper.

Processor-to-bus Interface

In addition to the CVAX system bus used to transfer floating point data, a dedicated 8-bit bidirectional coprocessor bus is used to communicate between the CVAX and the CFPA. An example of the CFPA system configuration is shown in Figure 1. The CFPA normally monitors the coprocessor bus for opcode and operand information until it is ready to drive a result back to the CVAX. After decoding an opcode, the CFPA monitors control signals on the bus that indicate the presence of an operand. Operands may come from a CPU general register, internal cache location, or from the memory system. When operands are transferred from CPU general registers or internal cache locations, the data is transmitted directly between the CVAX and the CFPA. Operands from external memory or cache locations are indicated on the coprocessor bus at the start of the external memory access. The CFPA then monitors the CVAX system bus and latches the returning data without CVAX intervention.

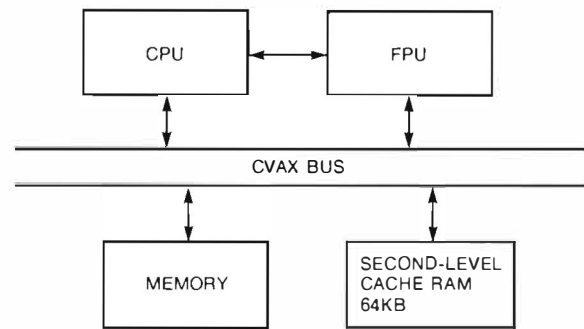


Figure 1 CFPA Example System Configuration

After supplying operands to the CFPA, the CVAX relinquishes control of the coprocessor bus to receive the result status of the floating point operation. Control of the coprocessor bus, however, does not imply control of the CVAX system bus. The CFPA ensures availability of the CVAX system bus by monitoring the direct memory access (DMA) grant signal from the CVAX. If a DMA has been granted, the floating point result status will be retransmitted until the DMA operation is complete. Receipt of the floating point status while the DMA grant signal is deasserted guarantees availability of the CVAX system bus for the next cycle. Control of the coprocessor bus is returned to the CVAX after successfully driving floating point status. The CFPA drives the result data on the CVAX system bus one cycle later, completing the operation.

Floating point instruction latency comprises overhead devoted to opcode, operand and result transfer, and actual computation, or execution time. Due solely to improvement in CVAX cycle time — from 200 ns in MicroVAX systems to 80 or 90 ns in CVAX systems — overhead times are improved by factors of 2.5 or 2.2, respectively. Designers achieved additional improvements in the interface by reducing the actual number of cycles required for these overhead transfers. As compared to the MicroVAX II system, the CVAX system requires fewer cycles to access and transmit register and internal cache operands located on the chip. Moreover, external cache and memory operands are input directly from the CVAX system bus as opposed to being fetched by the CPU and later retransmitted to the FPU as in the MicroVAX II system. The resulting interface improves performance by a factor of approximately 2.5 (90-ns cycle) to 2.8 (80-ns cycle) over the MicroVAX II system.

Despite these improvements, more than half the cycles required to execute a floating point instruction in the CVAX system can still be attributed to overhead costs. The possibility of pipelining macroinstructions — overlapping the operand fetches of the next instruction with execution of the current instruction — as well as operand forwarding was studied. In such a system the effective instruction time is determined by the longer of the operand transfers or the actual floating point execution time. Instruction time is not determined by the additive effect of the interface and execution. The one-instruction macro pipeline interface was rejected due to the risk and complexity of the design. Moreover, performance goals had already been met and development time was at a premium.

Algorithms

Although the interface figures prominently in the achievement of overall performance targets, most of our design efforts were focused on the actual execution unit. To maintain and even increase the benefits gained by the interface design improvements, we needed an equal or greater improvement in execution times. Since the most important instructions for a floating point unit are addition/subtraction, multiplication, and to a lesser extent division, designers set about optimizing these instructions. The remainder of instructions implemented by the CFPA also benefit from the shift, multiply, and divide optimizations and demonstrate performance gains relative to the MicroVAX II FPU as well. Finally, all instructions gain from microcode improvements in atypical case handling and from faster code entry and exit techniques.

Multiplication

Floating point multiplication consists of multiplication of the fractional, or mantissa, portions of the operands and the summation of the corresponding exponents. Many multiplication techniques have been developed and implemented to increase the speed of this frequently executed instruction. Perhaps the best technique for VLSI implementation at this time is the multiplier array. The array is particularly well suited for VLSI implementation due to the array's regularity of circuit connections which allow for a very compact and repeatable cell design.

The process of multiplication involves a series of additions. It is possible to delay the carry propagation necessary to complete these additions

until the final sum is formed through the use of carry save adders. Multiplier arrays consist of rows of carry save adders which add in a new multiple of the multiplicand at each row. The carry save adders produce a result, or partial product, consisting of two outputs, the carry and the sum; if added, the two outputs represent a single number equivalent to the partial product at that step obtained using full propagation addition. By deferring the final summation of the sum and carry words, the comparatively time-consuming carry propagation addition need be performed only once to produce the result.

The only drawback to the multiplier array is the large percentage of chip area devoted to this one operation. Nevertheless, the magnitude of performance gain warrants the use of an array in any high-performance computation unit.

Another common method used to improve the processing of multiplications involves multiple-bit Booth encoding. This method, which requires significantly less hardware, is aimed at reducing the number of partial products needed to be formed. The multiplier operand is encoded — or recoded — as a control pattern used to determine a sequence of shift and add or subtract operations on the multiplicand. Multiple bits of the multiplier can then be retired in a single operation. This method of reducing the number of multiplication steps can be employed either with or without an array structure.

The previous generation MicroVAX FPU executes multiplication using a fixed, 3-bit-per-cycle Booth algorithm without the use of a multiplier array. Single-precision multiplication requires 8 cycles to compute 25 product bits; *D_floating* and *G_floating* double-precision formats require 19 and 18 cycles to produce the necessary 57 or 54 product bits. Additional cycles are needed to set up the multiply loop, calculate the initial partial product based upon the multiplier least-significant bit (LSB), and round and normalize the final product.

The CFPA multiply algorithm takes advantage of the greater density and transistor count afforded by the CMOS process. The CFPA implements a multiplier array, which consists of four rows of 65 carry save adders. The multiplicand select logic associated with each row of the array as well as the interconnect between the rows is configured to implement a 2-bit Booth encoding. As a result of this configuration, 8 product bits are completed per pass through the array. Single-precision multiplication requires three passes

through the array, and double-precision requires seven passes to complete.

The array can be evaluated twice per cycle. Therefore, single-precision multiplication requires one and one-half cycles, and double-precision D-floating and G-floating formats require three and one-half cycles of processing in the array. Before running the array, one-half cycle is needed for set up and initial product calculation. After the multiplier array completes, a cycle is used to complete the full carry propagate add, which combines the final carry and sum outputs of the array. This cycle is followed by a normalization cycle during which valid status is returned to the CVAX.

When we compare the MicroVAX II system to the CFPA, the number of cycles required to complete a MULF instruction has been reduced from 14 to 4 (a ratio of 3.9 to 1 at 90 ns, 4.4 to 1 at 80 ns); to complete MULD or MULG instructions, the reduction is from 26 to 6 (4.8 to 1 at 90 ns, 5.4 to 1 at 80 ns). If we include operand transfers and count each interface cycle of the MicroVAX II system as equivalent to two CVAX cycles, however, the reduction in the total number of cycles for MULF is from 27 to 9 (3.3 to 1 at 90 ns, 3.8 to 1 at 80 ns); and for MULD, from 43 to 14 (3.4 to 1 at 90 ns, 3.8 to 1 at 80 ns) for register-mode instructions. When operands are read from or written to memory, the overhead support percentage becomes an even greater factor; and the impact of the actual CFPA multiplication speed is reduced.

To further increase performance, we considered an array of sufficient size to complete single-precision multiplication in a single pass and double-precision multiplication in two passes. However, such an array would require three times the chip area for a 2-bit algorithm. A 3-bit-per-row multiply would require 8 rows to complete single-precision multiplication in one pass and 9 or 10 rows to complete double-precision multiplication in two passes, as well as an adder to calculate the multiplicand factor of 3. Either of these alternatives, if feasible, would save only one cycle in single-precision (a reduction from 9 to 8, or 11 percent) and two cycles in double-precision multiplication (14 to 12, or 14 percent). In addition to the area requirements, the circuit design difficulty and risk involved to implement a larger array were deemed much too great for the limited gains. We therefore chose to trade off these smaller gains in favor of a partial

array of 4 rows of 2-bit-per-row retirement requiring only 1.3 mm of chip height. The result is a three and one-half to four times gain in the overall performance of multiplication.

Addition/Subtraction

Floating point addition involves a series of steps.

1. The exponents are subtracted to determine the shift amount necessary to align the fractions.
2. The fraction operand with the smaller exponent is shifted into alignment and added or subtracted.
3. The result is shifted back to the normalized form ($\leq \text{result} < 1.0$). Normalization shifting is accompanied by exponent adjustment.
4. The result is rounded and checked for overflow or underflow conditions.

Typically, the shifting operations and their control consume large amounts of chip area and potentially a large portion of the total calculation time. An analysis of these operations was used to guide trade-offs in the design of the CFPA.¹ It was noted that although large shifts are sometimes necessary to compute the final result, their frequency of occurrence is very small. Furthermore, a small shifter, capable of covering the vast majority of cases in a single operation provides the benefit of a small control circuit that can be more easily optimized for speed. It was decided that the speed and area advantages gained by designing for the most frequently occurring cases provided the best solution under project constraints.

Specifically, a small shifter that is capable of left-four to right-seven bit shifts proved to have adequate range for most alignment and normalization shifts. In up to 80 percent of the cases, additional cycles are not needed for alignment shifting. Larger alignment shifting utilizes the multiplier array for a shift capability of 16 bits per cycle. The array minimizes the worst-case shift time without requiring a large shifter. Although it rarely requires additional cycles, normalization shifting may cause a longer latency. Additional cycles, however, are not necessary for normalization in 93 percent of the cases.

To reduce the shifter control complexity, a modified ALU calculates the absolute value of the

exponent difference. The modified ALU does not require additional calculation time to accomplish this calculation. The absolute value result simplifies control logic to enable the alignment shifter to complete in the next clock phase. Only one additional generate term is needed to enable two carry chains executing simultaneously; one calculates A minus B, the other B minus A. The most significant bit (MSB) of the first carry chain determines the sign of the operation. To produce the absolute value or positive result, the MSB of the first carry chain is used to select the final output from the two carry chains. In addition, the MSB is used to select the fraction requiring alignment.

The CFPAs complete addition or subtraction operations in three cycles for most cases. This minimum execution time is exceeded for only 25 percent of all addition or subtraction operations, almost all of which require only one additional cycle.

The major improvement over the MicroVAX II FPU in the addition/subtraction algorithm is the elimination of no-operation cycles necessary for control evaluation preceding the alignment and normalization steps. The resultant reduction as compared to the MicroVAX II FPU is from eight cycles to three for both single- and double-precision additions/subtractions in the actual floating point unit calculations (3 to 1 at 90 ns, 3.3 to 1 at 80 ns).

The overall performance gain in equivalent cycles is 20 to 8 for single-precision (2.8 to 1 at 90 ns, 3.1 to 1 at 80 ns) and 26 to 11 for double-precision addition/subtraction (2.6 to 1 at 90 ns, 3.0 to 1 at 80 ns).

Division

Floating point division consists of division of the fraction or mantissa and subtraction of the exponents. Division presents a more intractable problem than multiplication when designing for high-speed performance. The difficulty arises due to the fact that the partial remainder at each step must be examined before the next operation can be determined. Various algorithms have been proposed to reduce the number of arithmetic steps, but no single solution seems to optimize both performance and size constraints.

The CFPAs use a method of division that offers an improvement over single-bit division algorithms, which perform an arithmetic operation to produce a single quotient bit per step. The

method calls for shifting over, or normalizing, multiple leading bits when the partial remainder is small. A partial remainder with multiple leading ones indicates a small negative remainder, whereas leading zeros indicate a small positive remainder. Multiple quotient bits can be determined for cycles in which the magnitude of the partial remainder is small. Shift operations replace arithmetic operations on unnormalized remainders, reducing the number of ALU cycles needed to develop the final quotient. This method of division is called normalizing, non-restoring division and is also used in the MicroVAX FPU. The difference between the two implementations is in the normalization shift range provided for partial remainder and quotient development.

Of course, this algorithm is quite data sensitive. A division that results in a partial remainder of all ones or all zeros can be completed in a minimum amount of time; whereas, if a string of alternating ones and zeros is produced at each ALU operation, the process degenerates to a one-bit-per-cycle pace. The observed average rate for an algorithm that allows unlimited shift range is 2.66 bits per cycle. Unfortunately, the shift range chosen implies a control structure directly between the shift and ALU operations. The time between these operations is critically important to the overall cycle of the chip. We chose 4 bits as the left shift range for the CFPAs to reap the maximum benefit from the technique without introducing inordinately difficult control paths between the shift and ALU operations. This amounts to an increase of 2 bits of shift range over the MicroVAX FPU. Correspondingly, the average number of quotient bits developed each cycle increased from 1.5 to 2.4. Expanding the shifter beyond a range of 4 for this method provides a diminishing improvement, as shown in Table 2.

Table 2 Average Quotient Bits per Cycle

Shifter Range	Average Speed
2	1.5
4	2.39
6	2.54
8	2.64
Unlimited	2.66

Increasing the number of quotient bits developed per cycle from 1.5 to 2.4 results in increased speeds in the CFPA divide loop relative to the MicroVAX FPU: 1.8 times greater for 90-ns cycles, and 2.0 times greater for 80-ns cycles. The overhead cycles involved in setting up the divide sequence and normalizing the quotient are reduced from 7 to 2. As a result, the CFPA realizes a performance greater than the MicroVAX II FPU in terms of number of cycles reduced for division. Including the processor-to-FPU interface cycles, the number of cycles for single-precision division is reduced from 37 to 18 cycles (2.3 at 90 ns, 2.6 at 80 ns); for D-floating double-precision division, 61 to 35 (1.94 at 90 ns, 2.2 at 80 ns).

Comparatively, this method of division is very efficient, especially when we consider the small amount of control circuitry and data path area required. Designers can increase performance additionally by using algorithms that employ multiples of the divisor, or by implementing a divider array structure. The use of multiples of the divisor requires both additional registers to hold the multiples ($3/4$, 1, $3/2$) and further expansion of the left shift capability to take advantage of the longer normalizations created by this approach (3.6 bits per cycle with left shift range expanded to 6). In addition, the control logic required to support the selection of the proper multiple is more complex and would be much more difficult to implement in the constrained cycle time. The other alternative of executing the divide step in an array structure for performance capable of 3 to 4 quotient bits per cycle involves an even greater cost in hardware and is not consistent with the project goals.

Integer division does not automatically benefit from hardware devoted to floating point division. Since floating point division relies on the normalization of the operands, integer division must either convert operands to the normalized form or accept a slower one-bit-per-cycle algorithm. The CFPA design for integer division normalizes both the divisor and dividend in order to use the 2.4-bit-per-cycle divide algorithm. Normalization of the divisor and dividend proceeds at 5 bits per cycle. The number of quotient bits needed to complete the integer division operation is determined by the difference between the normalization shift amounts of the divisor and dividend. Consequently, integer divides are typically executed at

2.5 bits per cycle as compared to 1 bit per cycle on the MicroVAX FPU.

Microcode Control Structure

The control structure for the CFPA is influenced by two opposing constraints. The complicated requirements of instructions such as extended multiply and integerize (EMOD) and polynomial evaluation (POLY) require the flexibility offered by a microcoded approach. Performance goals, however, require the speed of hardwired control structures to avoid costly delays incurred during microcode branch handling. The final implementation combines a small control PLA (programmable logic array) to provide the flexibility of microcode control with hardware control structures for speed critical paths. These control structures are enabled through the microcode to emulate complete hardwired control for important instructions. The structures provide support for alignment, normalization, multiplication and division steps. Standard microcode control supports the less critical instructions.

Functions are performed under more straightforward microcode control when the code does not penalize the instruction performance. This trade-off simplifies critical circuitry in some instances. The only exception to this rule is in the handling of exception conditions. If an exception condition can be isolated from the normal instruction flow, it is also processed in microcode rather than through the more expensive hardware control.

The use of hardware structures reduces the total number of microcode terms needed to implement the instruction set. This reduction is important to ensure that the microcode PLA can be implemented with an access time of one half cycle. Instructions generally use one code flow for all data types. In addition, similar instructions merge sections of flows to further minimize terms. For example, the add-compare-and-branch (ACB) instruction, which is one of the more complicated instructions implemented by the chip, required only three additional terms beyond the addition and compare instruction flows. Despite this effort, almost one third of the code was devoted exclusively to two instruction types, EMOD and POLY. By splitting, or "folding," the PLA into two half-height interleaved arrays, the target speed was met with a penalty of only a few duplicated terms. In total, 76 VAX floating point

as well as integer multiply and divide instructions are implemented in the CFPA. In comparison to the MicroVAX FPU, the total number of microcode states was reduced by 20 percent, to only 159.

Microprogramming

As mentioned earlier, the use of hardware support contributes to improved performance for most instructions. However, since the CFPA cycle time during execution is very similar to that of the MicroVAX FPU (80 or 90 ns versus 100 ns), we needed further improvement to meet the project goals. Algorithmic improvement in the convert-floating-to-integer (CVTFI) and EMOD instructions provides between three and four times the performance of the MicroVAX FPU for the same instructions. But these gains would hardly translate to improved overall performance when considering the frequency of use for these instructions. Therefore, to reduce cycles for all instructions, we examined transitions during code entry and exit with internal processing. Since the CFPA always receives the opcode in advance of the operands, it is possible to reduce the execution time for all instructions by performing the first step of each operation repeatedly in anticipation of receiving the last operand. In this way, as soon as the interface recognizes that the operand is valid and the control sequencer is able to act on that information, the first step of the instruction is already complete.

In the CVAX system, as in the MicroVAX II system, floating point status must be returned before data can be received. One reason for this return of status is that it prepares the write path back to the general-purpose register file located on the CPU chip. Status conditions must be checked before the result register is written; the register update can thus be inhibited in the case of an error or exception condition. Latency was reduced on almost all instructions by transmitting the result status back to the CVAX CPU in the same cycle as the last step of execution. This is accomplished by checking the result prior to the last normalization or round operation in order to determine if the possibility of an exception condition exists. Since F_floating and D_floating formats use an exponent with a range of 256 values, and G_floating format increases that range to 2,048 possible values, the exponent is in range for most results, and a no exception status

can be returned prior to determination of the final result.

CFPA Implementation

After deciding on a set of basic algorithms that appeared to meet the project goals, the development effort proceeded to actual implementation. Individual algorithms can sometimes result in a proposed hardware solution that requires modifications to either the hardware or to the algorithm in order to be implemented within design constraints. Merging the requirements of several algorithms can create implementation conflicts throughout the physical design. Care must be taken to consider the opposing requirements while incorporating the necessary features in a single design. The algorithms for the CFPA were chosen with a single hardware microarchitecture in mind. That architecture evolved as the design progressed, but the architecture maintained the basic structure that was used as a framework for early circuit design and feasibility study. The following section outlines the overall hardware microarchitecture for the CFPA. This section is followed by explanations of the more interesting circuit design issues.

Microarchitecture

The CFPA contains two main functional units:

- The execution unit, which performs all arithmetic calculations
- The bus interface unit (BIU), which controls all I/O operations

A block diagram of these units is shown in Figure 2.

The execution unit consists of two main data paths and their associated control logic. The 65-bit fraction data path contains an integral multiplier array and also processes integer data. Also included in the fraction data path are a small 4-bit left to 7-bit right shifter, a general-purpose ALU, scratch register, ROM constants, and quotient register and shifter. The second data path, the exponent data path, is 13 bits wide and contains a modified ALU design used to calculate absolute values needed in floating point addition. The exponent data path operates in parallel with the fraction data path and may be controlled independently or conditionally based upon results from the fraction data path. A 160-term PLA,

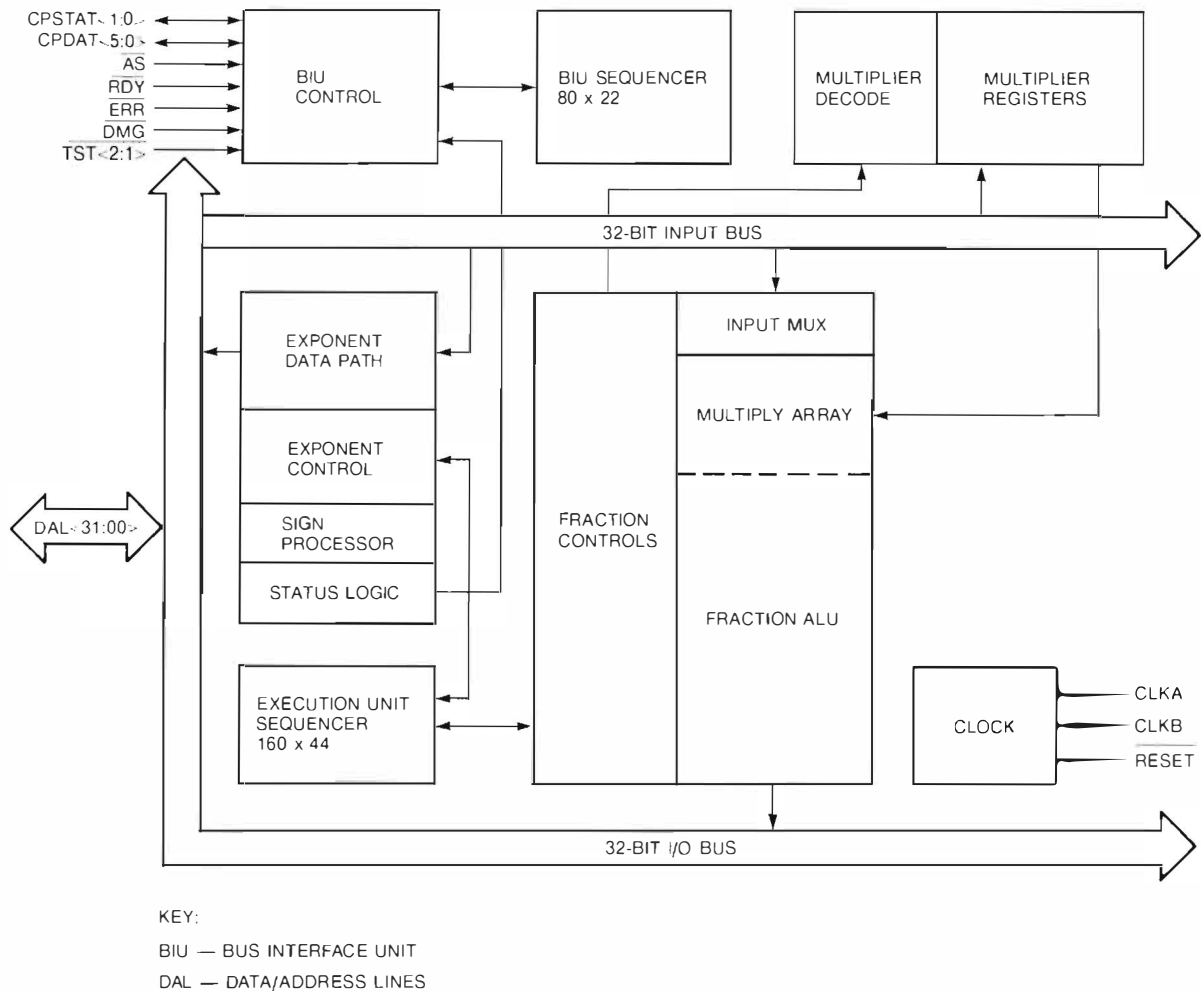


Figure 2 CFPA Block Diagram

which accesses a single 44-bit microword each cycle, controls the execution unit.

The BIU controls the interface between the CPU and memory system. A 70-term PLA in the unit controls all I/O transactions between the CVAX and CFPA. The BIU also controls the test-mode logic to allow visibility to the data paths and execution unit PLA during operation.

Figure 3 illustrates the physical layout of these structures on the CFPA die.

Circuit Design

Clocking

The CFPA chip employs a four-phase overlapping clocking scheme which provides timing resolution. Much of the control circuitry design calls for combinational circuits that operate between

latches clocked on nonconsecutive phases, which are nonoverlapping.

Multiplier

As noted in the section Multiplication, it was recognized early in the chip design that the multiplier array would be key to meeting the desired performance. The CFPA implements multiplication by using an array of carry save adders with partial product wraparound. The wraparound enables the array to be cycled as many times as necessary. The final carry and sum addition is executed in the fraction ALU. A static implementation of the carry save adders is necessary since data propagates through multiple rows of the array.

To build the carry save adders, we used a four-transistor XOR. This approach allowed for mini-

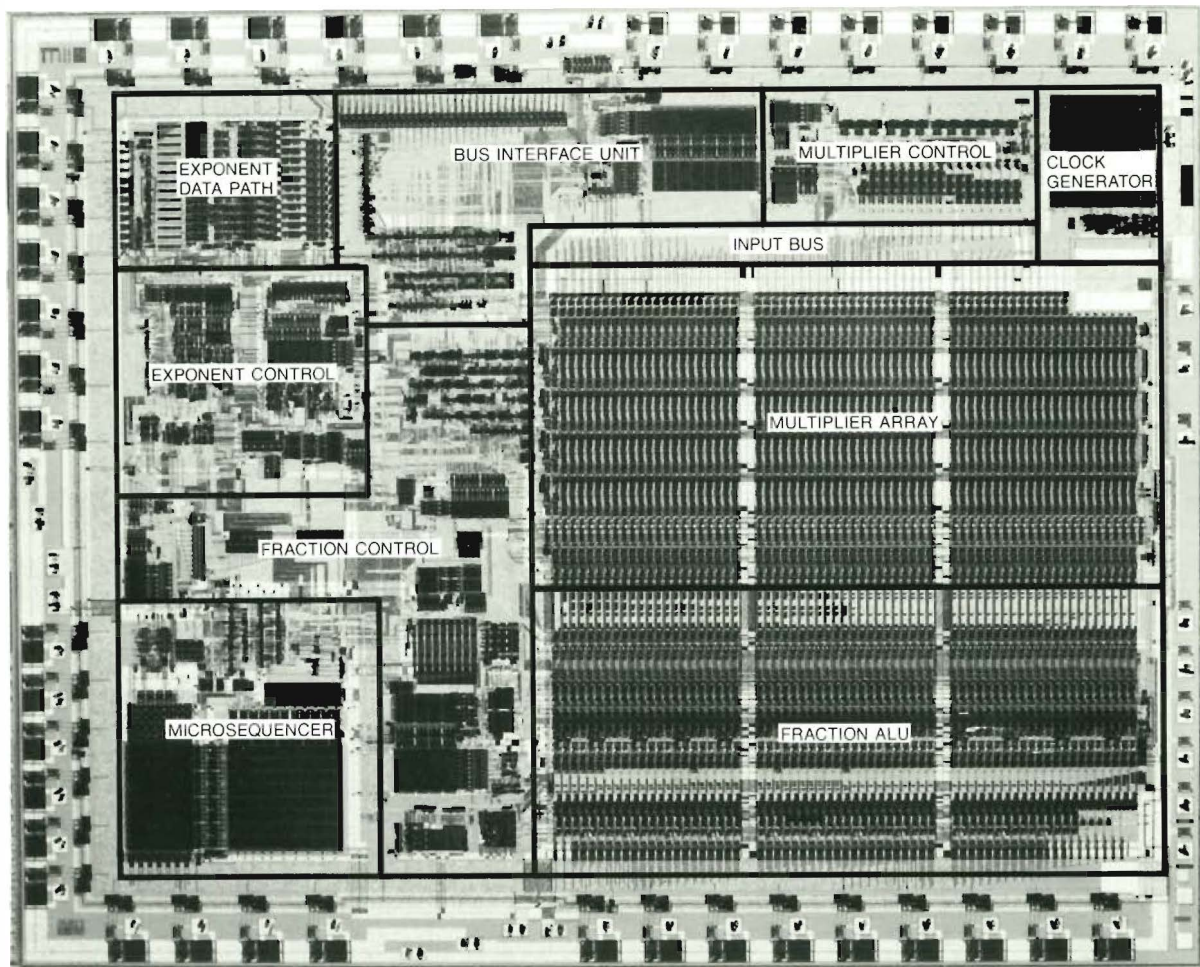


Figure 3 CFP Physical Layout

imum delay and required the least amount of chip area. As a result of SPICE simulation, we found that doubling the minimum size of the transistors in the multiplier array could provide a 20 percent speed increase. Since the cell area was constrained by the necessary interconnect in the metal layers, the device sizes were increased without affecting the cell size. Further device size increases, however, would have forced us to increase the cell size and would not have improved speed appreciably due to increased self-loading. With the approach we chose, SPICE simulation showed a worst-case delay of 6.5 ns per row and a typical delay of 4.5 ns.

To obtain the desired multiplication performance and minimize the area necessary for the multiplier array, we used a technique in which the array is cycled twice per microcycle. For

worst-case devices, a half cycle takes 45 ns. An array size of four rows takes 26 ns to propagate through the array, allowing 19 ns for latching, return of partial products, and control switching. For typical devices four rows complete in 18 ns, allowing 22 ns in an 80-ns cycle for the wraparound path.

Control PLA

We also recognized the fraction shift control PLA as a possible speed limitation. The shift control PLA was the largest PLA in the control section and had to evaluate in a single clock phase. Because no clock signals were available to control evaluation of the PLA, we used a "dummy" AND array term to start evaluation of the OR array. A "dummy" OR line controls output clocking, making the PLA self-timed. Because this PLA could be

evaluated in a single clock phase, both alignment and normalization operations were able to eliminate an unnecessary wait cycle present on the MicroVAX FPU. We were also able to expand the divide algorithm to 4 bit shifts per cycle.

As we had suspected, the limiting factor in the final chip cycle time was the multiplier array. The ALUs and the large control PLAs in both the microcode control section and the BIU easily met speed requirements in the CMOS I process.

Design Methodology

As VLSI technology improves, both chip area and density increase, allowing much larger and more complicated designs to be attempted. Critical to any large project, the ability to predict and adjust the design according to the most current information plays an important role in achieving a successful project outcome in a minimum of time. This section describes the various phases and feedback paths of the design process for the CFPA and some of the unique aspects of VLSI design.

In the first phase of design, we defined the major sections and the necessary global signals communicating between them. The major outputs of this phase were hand-drawn sets of notes on the necessary functions of each section and preliminary sketches of possible implementations. Early in the design, we recognized that certain subsections would be critical to meeting the desired performance goals. These particularly critical sections were

- The multiplier array
- The exponent input path
- The fraction shifter controls

We therefore generated more detailed preliminary designs for all of these sections. Moreover we tested their feasibility with SPICE circuit simulations. The MSB and LSB logic in the multiplier was also verified with an APL language simulation of the multiplier array.

One of the hazards in the early stages of a project is the tendency to spend too much effort perfecting one small piece of the design. If the original requirements are modified at a later date, much time is wasted. The design team, therefore, made a conscious effort to keep all parts of the design at similar levels of detail at all times throughout the project.

For purposes of design checking and chip implementation, we divided the CFPA into seven major sections: fraction data path, fraction data

path controls, exponent data path, exponent data path controls, microsequencer, bus interface unit, and clock generator. Consistent divisions and global signals between these major sections were maintained in both the behavioral and transistor modeling levels as well as in the final mask artwork. This approach allows maximum possible checking to be carried out on each section, independent of the state of other sections of the chip.

Upon completion of the initial design conception, a behavioral model was written in the DECSIM simulation language. This model helped us to refine the algorithms and further define the data path and control structures. We rewrote the model several times to improve detail and incorporate design changes. From early in the development, the behavioral model was merged with the CVAX CPU chip model and a small system environment to provide a platform for more extensive testing. Existing diagnostic programs were therefore able to be run on the model to provide early checks on the design integrity. Additional tests were written to verify specific features of the CFPA implementation before we began the detailed circuit design for critical sections. Throughout the development phase, we used the VAX Architectural Exerciser (AXE) extensively to test instruction compatibility with existing VAX implementations. Despite a degradation of approximately 1M : 1 while using the simulator to run test code, well over 500,000 test cases were run on the behavioral model before the design was considered ready for fabrication.

Using the DECSIM MOS device simulation system, we created a transistor-level model from final schematics as they were completed. By collecting test patterns from the appropriate signals in the behavioral model, the team could begin to debug the schematic in complete sections as other sections were still being designed. To do this efficiently, the DECSIM group modified their simulator to allow designers to write a binary state file and reload the file for examination. This facility gave logic designers a very efficient means to debug the transistor-level logic. Designers could run their simulations in batch mode over night, examine the resulting patterns for mismatches with the behavioral model results, and then "back up" to the area before the failure test point to find the underlying cause. They could perform all these steps without rerunning the entire simulation each time they wanted to go back in time to look at another signal.

As each section of the transistor-level schematic was developed to a satisfactory level of accuracy, the third phase of the design — creation of the physical layout artwork — began on that section. To create the artwork, a Calma GDS interactive editing system was used. Over the course of the project, three layout designers were employed full time. Toward the end of the layout phase, up to four additional designers were working on various parts of the chip. Each section was checked with the interconnect verification (IV) wirelist extraction tool and a design rule checker (DRC) program.

As all the sections were drawn and global interconnect wiring was added to the chip layout, the fourth phase of the design — the back end checks — began. The IV program was used to extract actual capacitance values for all nodes on the chip. We used these capacitance values in two ways to check the design. First, they were compiled into the DECSIM MOS transistor-level simulator. The timing feature of this tool was used to quickly check for gross timing problems over the entire chip operating as a whole. Once we identified an area as having a possible timing problem and for those areas where we believed the DECSIM MOS simulation was inaccurate, we created and ran SPICE circuit simulations. In a second use of the extracted capacitance values, a program called PATH was written in the SCAN compiler generator language. PATH allowed the circuit designers to easily and accurately create wirelists representing critical paths for submission to SPICE. The program extracts a circuit path description from the much larger wirelists generated from either the IV tool or the chip-wide schematics. Wirelists created by the IV program include interconnect and capacitance information directly from layout artwork.

Although the chip design process appears in this discussion to be a neat progression, the various aspects of the actual project quickly overlapped one another. Almost all phases were taking place simultaneously on the various sections of the chip. To keep track of all these activities and continually update the project completion date, we used a spread-sheet program as a tracking tool.

The design team of 11 people completed the project in 21 months, including 6 months for product conception and 15 months for implementation. Due to the extensive modeling and simulation prior to device fabrication, initial parts were functional at speed.

Test Features

To aid the debugging process and provide more complete test coverage, the BIU contains test logic. This logic allows visibility to both data paths or to the main PLA. A simple test load sequence allows one of 16 possible test modes to be selected. Various groups of internal data path and control bits and two test-drive timing options are allowed. The test mode can be enabled or disabled at any time by asserting a single test pin. Certain test modes are available while operating at full speed in a system configuration.

CFPA Performance

Although there is no absolute measure of performance in computer system design, the floating point performance of the CVAX system is compared at approximately three times the performance of the MicroVAX II system. Using some of the more widely publicized benchmarks of floating point system performance, the CVAX system with CFPA running at 25 MHz shows better than three times the speed of the MicroVAX II with FPU. The system calculates 3,105K single-precision Whetstone instructions per second and 1,996K double-precision Whetstone instructions per second. Linpack performance of 0.68 Mflops single-precision and 0.45 Mflops double-precision demonstrate over four times the performance of the previous generation MicroVAX implementation.

Table 3 lists the typical cycle counts for register-to-register execution of floating point addition, subtraction, multiplication, and division.

Table 3 CFPA Cycle Counts for Optimized Instructions

Instruction	CFPA Cycles	Opcode/Operand Transfers	Total Cycles
ADDF/SUBF	3	5	8
MULF	4	5	9
DIVF	13	5	18
ADDD/SUBD	4	7	11
MULD	6	7	13
DIVD	27	7	34
ADDG/SUBG	4	8	12
MULG	6	8	14
DIVG	26	8	34

Acknowledgments

In addition to the authors, members of the CFPA team were Roy Badeau, John Kowaleski, Omar Malik, and John Ellis who contributed to the logic and circuit design as well as Katie Alexandrowicz, Mike Benoit, Larry Commodore, Sharon Crafts, Bob Hicks, Dennis Hodges, Ellen Kagan, Marc Schenstrom, Tim Thrush, and Peter Wilcox who created the layout for the chip. Test, product engineering, and additional technical contributions were made by Dilip Bhavsar, Shlomo Daniely, Larry Harada, Charlie Hilman, Vinod Rai, and Nigel Scott.

Reference

1. D. Sweeney, "An Analysis of Floating Point Addition," *IBM Systems Journal*, vol. 4 (1965): 31-42.

General References

T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital*

Technical Journal (August 1988, this issue): 95-108.

E. McLellan, G. Wolrich, et al., "The CVAX FPU, A CMOS VAX Floating Point Unit," *ICCD Proceedings* (October 1987): 153-156.

P. Rubinfeld, et al., "The CVAX CPU, A CMOS VAX Microprocessor Chip," *ICCD Proceedings* (October 1987): 148-152.

W. Bidermann, et al., "The MicroVAX 78132 Floating Point Chip," *Digital Technical Journal* (March 1986): 24-36.

G. Wolrich, et al., "A High Performance Floating Point Coprocessor," *IEEE Journal of Solid-State Circuits*, vol. SC-19, no. 5 (October 1984): 690-696.

T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1987).

O. L. MacSorley, "High-speed Arithmetic in Binary Computers," *Proceedings of the IRE*, vol. 49 (1961): 67-91.

The System Support Chip, a Multifunction Chip for CVAX Systems

Developed as a general-purpose companion to the new CMOS VAX VLSI chips, the System Support Chip (SSC) contains a common core of peripheral system functions which are required to support a MicroVAX system environment. These functions include timers, VAX console support, and standby RAM. In addition, the SSC provides system designers with "hooks" to other system functions. With these peripheral functions integrated on a single chip, system designers can substantially reduce the number of components on a module and add features previously not considered cost effective. Primarily used with the CVAX CPU chip, the SSC is also compatible with the NMOS MicroVAX CPU chip.

Background and Goals

In 1984, as the VAX 8200 and MicroVAX II chip sets entered production, Digital's Semiconductor Engineering Group (SEG) directed its attention toward defining the next generation of MicroVAX systems.¹ This paper describes the project history and functionality of one of this new generation's peripheral chips, the MicroVAX System Support Chip (SSC). Developed over a period of 18 months beginning in late 1984, the SSC was designed as a general-purpose companion to the CVAX CPU. As such, the chip is used in the VAX 6200 family and in the MicroVAX 3000 family.^{2,3}

As part of the definition of the new CMOS VAX family of VLSI chips, SEG looked at the peripheral functions that surrounded the existing MicroVAX II CPU. We observed that, to build a marketable product, each system group had added a collection of timers, decoders, and other low- and mid-complexity functions to their respective modules. A high level of similarity from module to module was apparent in the makeup of these functions.

In addition to examining these existing modules, we talked with the system designers to learn what additional functions should be included on the next generation of systems. Again, we found that the various systems under development would have a significant number of overlapping functional requirements.

We decided a chip that provided the common core of these peripheral functions would be a strategic component for Digital products. This single chip would integrate many of the peripheral functions usually required on MicroVAX CPU modules. Consequently, a system designer could substantially reduce the number of components on a CPU module and add features that previously would not have been cost effective. Moreover, the chip would allow him to add features without lengthening the project schedule or requiring extra resources. As a result, the system designer could produce a more competitive Digital product at little additional cost.

From the system designer's viewpoint, the chip would

- Fully implement many functions used identically across different MicroVAX systems, such as timers, ROM support, and standby RAM
- Provide the "hooks" to support other functions that would be implemented differently in the different system environments

Thus each system group would no longer need to design, implement, and debug these important peripheral functions from scratch. Instead, they could use a readily available part that had been debugged and qualified. Further, since the SSC would use custom CMOS VLSI, this chip would contain some additional useful functions, such as

general-purpose timers, that are expensive to implement in off-the-shelf or gate array technology.

With these goals outlined, we began development of the SSC. The following section presents an overview of the chip. In the balance of the paper, we describe the chip functions in detail and discuss the trade-offs made and problems encountered during development.

SSC Overview

The SSC incorporates onto a single chip a common core of functions required to support the VAX system environment. Table 1 lists the essential physical characteristics of the chip. Figure 1, a photograph of the chip, shows the major sections. Grouped into three main categories, these sections are

- Support for power-up booting and the VAX console
- Clock and timing functions
- Features required by the VMS operating system and those commonly required on a VAX CPU module.

We begin our detailed discussions of the chip functions with the SSC console and boot code support.

Console and Boot Code Support

The peripheral support described in this section includes ROM packing, halt-protection, the UARTs, and standby RAM.

ROM Packing

When a MicroVAX CPU is powered up, it begins executing code from read-only memory (ROM). To properly communicate with an off-the-shelf ROM, the microprocessor requires additional interfacing logic. The SSC provides this logic by generating the signals needed for the ROM-to-microprocessor interface. The SSC also provides the packing support for data-width compatibility between the ROM and the microprocessor.

At project outset, SSC designers assumed the module designers would use four ROMs in parallel to provide a 32-bit-wide ROM word to the CPU. However, with ROMs becoming denser every year, it is now possible to put all boot, console, and diagnostic code in one or two 8-bit-wide ROMs. System designers therefore chose to use fewer ROMs, decreasing the number of compo-

Table 1 SSC Physical Characteristics

Total device count	84,000 (approx.)
Die size	8.0 mm × 7.5 mm
Power dissipation	Less than 1.0 W, worst case
Packaging	84-pin surface mount
Clock	40 MHz external; 20 MHz internal; 25.6 kHz for time-of-year clock

nents on the module and thus the product cost. The MicroVAX 3000 uses two 64 kilobit (Kb) ROMs in parallel, forming a 16-bit ROM word. The VAX 6200 system uses two 64Kb ROMs in series.

To provide data-width compatibility between the 32-bit-wide CVAX bus and the narrower ROMs, the SSC includes packing support for 16-bit word-wide or 8-bit byte-wide external ROM. With packing support, the SSC performs multiple reads of the narrow ROM word, assembles a 32-bit longword, and sends the longword back to the microprocessor. The SSC performs this function by directly driving the output enable and address lines 1 and 0 of the ROM. (See Figure 2.) The ROM's other address pins are driven by an external address latch, and the data lines of the ROM drive the CVAX bus directly.

To pack a ROM, the SSC asserts output enable, drives the appropriate combinations of ROM address pins 1 and 0, and receives the narrow data across the CVAX bus in consecutive ROM access cycles (unbeknownst to the microprocessor). The SSC then deasserts output enable, puts the packed longword on the CVAX bus, and completes the read transaction.

CPU Halt-request Protection

System designers requested that the SSC help prevent an undesired condition in the halt logic. When the halt pin is asserted on the microprocessor, it executes a special trap to console code stored in the ROM. A second assertion of the CPU's halt pin (typically generated when someone repeatedly presses the halt button on the system front panel) causes a second such trap, overwriting the pointer needed to return to program code upon leaving console mode. Without this pointer, normal operation of the machine cannot be resumed without booting. Obviously system designers wanted to prevent this condition.

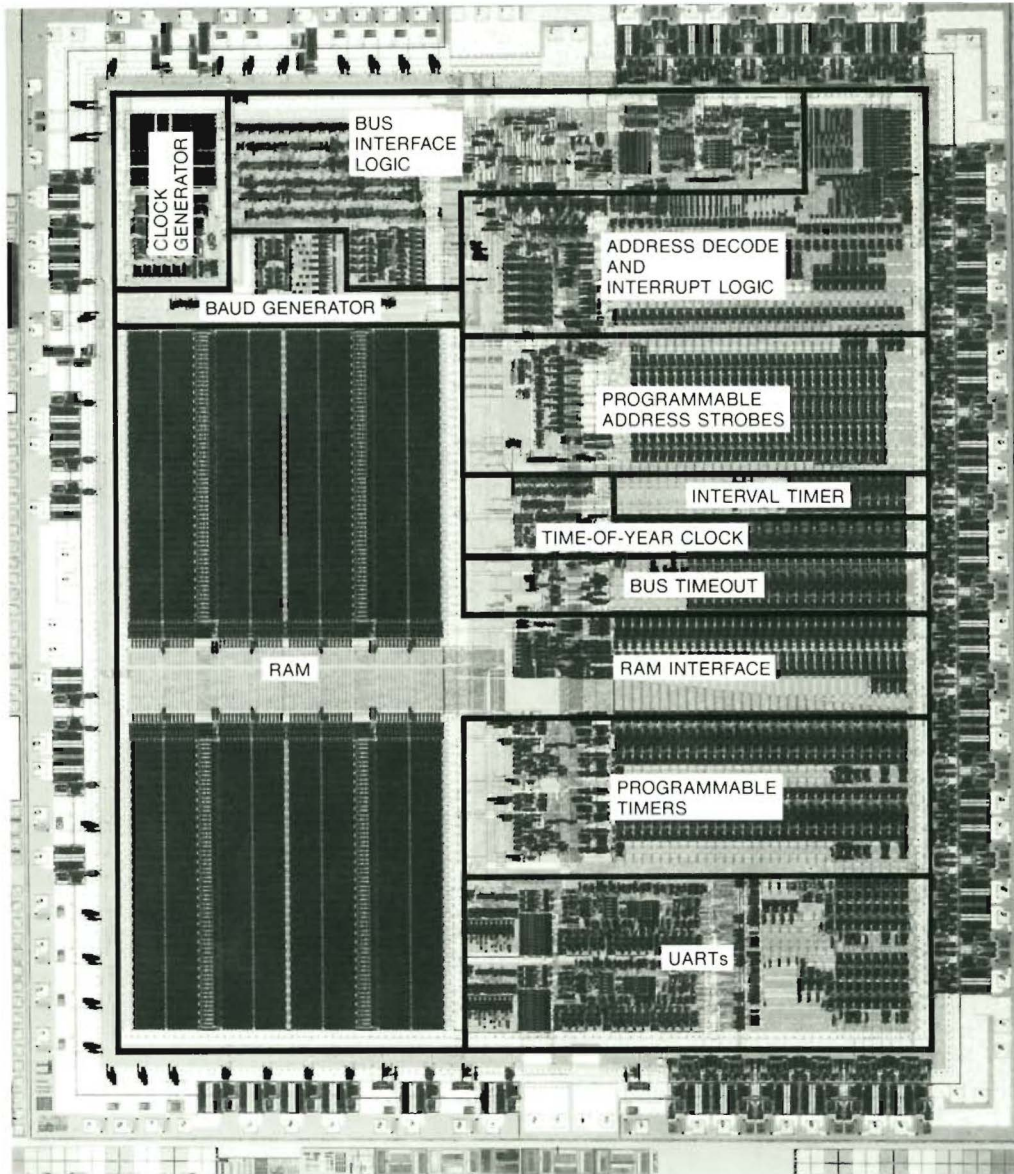


Figure 1 SSC Photograph Showing Major Sections

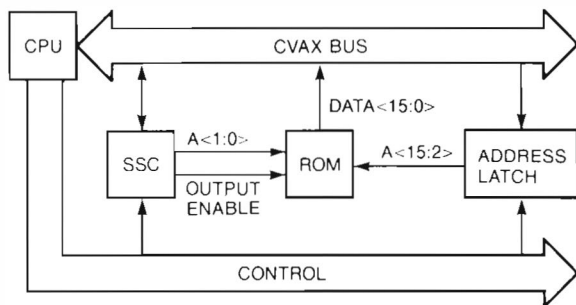


Figure 2 SSC ROM Packing Connection Diagram

The SSC prevents the second call by monitoring the addresses of all instruction reads and by intercepting all external halt requests made to the CPU. During normal CPU operation, the SSC passes an initial halt request to the microprocessor. The microprocessor immediately begins to execute from halt-protected space, which is a special address space programmed into the SSC by the user at boot time.

When the CPU reads the first instruction from console code, the SSC detects this console code address and masks further halt requests. These

requests are masked as long as the microprocessor is executing ROM console code. The console code can then run uninterrupted by halts. During console code execution, the SSC continues to monitor all instruction addresses. When an address outside halt-protected space is detected, the SSC re-enables halt requests to the CPU.

Before deciding on the design described above, we considered implementing a software-controlled bit that would enable and disable halts. This scheme would require the software to set the bit upon entering halt-protected space and to clear the bit upon re-entering normal operation. Although apparently simpler, this scheme proved to be flawed because two conditions might occur that would prevent the user from halting the system: (1) the bit could be accidentally set by non-boot code, or (2) a software error in the boot code could cause the microprocessor to start executing nonsystem code.

With the plan we chose, control is automatically returned to the user as soon as the software completes execution of the assumably bugfree halt-protected boot code. The system designer can, however, provide software control of the halt-enable function by aliasing the boot ROM into two adjacent spaces, where only one copy is halt protected. The software can then control halts by jumping between copies of the code. (This method is used on the MicroVAX II and MicroVAX 3500/3600 systems.)

UARTs

Although it was clear from the beginning that the SSC should provide UARTs, the best choice for number and design was not immediately clear. We had two choices at the time the chip was defined:

- Double-buffered DEC DLARTs (DC-319), which were in wide use, although a few problems with this design had recently surfaced
- Silo designs, which were becoming popular, though large in size

To conserve chip area, the SSC team settled on a design very similar to the DEC DLART design, making a few improvements in response to user requests. To keep from unduly complicating the design, we also decided to limit the number of UARTs to two (the number supported as console

ports within the VAX architecture).⁴ As a further simplification, we limited the number of baud rates to eight power-of-two choices (300 to 38,400 baud).

Our most significant improvement to the DLART design was the addition of hardware control-P break-detection. Control-P entered on a VAX console is interpreted as a halt request. Thus, the UART must pick out this special keystroke from the normal character stream and then signal the CPU to take appropriate action. Formerly, this function was performed by cumbersome firmware. However, the SSC hardware continuously watches for this character and, when it senses control-P, automatically signals the microprocessor.

The console code may configure the SSC such that a break is defined as a control-P or as 20 spaces; the latter is a definition still used in some console applications. At one point, we had planned to use the chip timebase to define a break as a space lasting a fixed number of milliseconds instead of 20 spaces. However, users advised us that this new idea, although more elegant, would make the UART more confusing to use.

Other improvements include better notification of overrun and framing errors, and secure console support. Console security is effected by a pin. When grounded, the pin prevents a break from halting the CPU. This pin is typically connected to a key switch on the computer's front panel. Using the switch, the user can lock out console-induced halts.

Further, the SSC allows the CPU to directly access the UARTs, time-of-year clock, and bus reset register by means of the VAX external processor register protocol. Using this protocol, the microprocessor can address system registers located outside the microprocessor by register number rather than by complete address. The SSC understands this protocol and is capable of decoding the register number and generating the desired response. Previously, VAX module designers using off-the-shelf UARTs had to implement a substantial amount of external logic to decode the register addresses and enable the UARTs to respond to this protocol.

Finally, the UARTs support break transmit and loopback, and properly respond to VAX interrupts. In products containing the SSC, one UART is used as the system console; the other is used for auxiliary functions, such as remote diagnostics, or is disabled.

Standby RAM

When a VAX system is powered off, the operating system must store some information in non-volatile memory until the system is powered up again. This stored information describes the system configuration and contains pointers to restart data stored on the disk. On the MicroVAX II CPU module, a watch chip provided 50 bytes of storage for this purpose. System designers indicated this amount was inadequate; 500 to 1000 bytes was desirable.

To meet this standby storage need, the SSC provides 1 kilobyte (KB) of battery backed-up random-access memory (RAM), organized as 256 by 32 bits. This RAM is also used as a system "scratch pad" during power-up test.

Additional standby support features are described in the section Standby Features.

Timers

The SSC timers serve to improve system reliability, meet architecture requirements, and save module space. These timers include the programmable bus timeout, the interval timer, general-purpose timers, and the time-of-year clock discussed in this section.

Bus Timeout

Since the CVAX bus is a handshake bus, incomplete bus transactions can hang the system. Some older VAX systems permit this condition; when those systems were designed, the high cost of implementing a timeout in external logic could not be justified in relation to the rarity of this event. However, the SSC improves system reliability by providing a programmable bus timeout at no additional system cost.

If a transaction lasts longer than a user-programmed interval, the chip

- Signals the microprocessor that a bus error has occurred
- Terminates the transaction
- Sets certain internal status flags based on the type of transaction that timed out

The status flags differentiate the two types of timeouts: (1) unexpected timeouts of read or write transactions, and (2) permissible timeouts caused by some unimplemented external processor registers or by certain interrupt-acknowledge transactions. After the timed-out transaction is

terminated, error-handling code reads the SSC internal status flags and takes the appropriate action.

The timeout interval may be programmed in 1-microsecond increments up to 16 seconds. The larger values are used to time out system self-test.

Interval Timer

The VAX architecture specifies a complete interval clock which the operating system uses to schedule time-critical system functions at regular intervals. On MicroVAX CPUs, logic for the clock is simplified to reduce the amount of circuitry on the microprocessor chip. On these microprocessors, only an interrupt-enable bit is implemented. The timer source is generated externally and is driven onto an input pin of the microprocessor chip. When the interrupt-enable bit is set, an interrupt request is generated on the falling edge of the timer source, which is a 100-Hz signal on MicroVAX systems.

The SSC eliminates the need for the module designer to place another oscillator on the CPU module by providing a 100-Hz output suitable for driving the interval timer input to the microprocessor chip.

General-purpose Timers

Early in the SSC development, many potential users voiced a need for general-purpose timers on future MicroVAX modules. However, no one had specific recommendations on how such functionality should be implemented. Some users requested four timers; whereas others reasoned that one timer supported with software could do the work of four or eight timers.

After some design attempts, we decided to copy, bit for bit, the VAX standard interval clock. We reasoned that it was prudent to select a design that was already well thought out and in general use. We did add one control bit to provide a one-shot capability. Our decision to include two timers was based on the amount of available chip area and a desire for some redundancy.

Each timer provides scheduled interrupts with 1-microsecond resolution. The maximum interval between interrupts is 1.2 hours. In one-shot mode, the timer stops upon generating its first interrupt. In single-step mode, a count can be caused only by writing to a specific control bit. The interrupt vector is user-programmable.

These timers are not used by the CPU module, but are available to the end user. We expect them to be very helpful to users designing embedded, time-sensitive applications.

Time-of-year Clock

The VAX architecture requires a battery-backed-up time-of-year clock with a resolution of 10 milliseconds (ms). When the MicroVAX II CPU module was designed, the best method for providing this feature involved the use of a BCD watch chip, approximately one-half gate array of logic to interface the chip to the MicroVAX bus, and some specially written operating system code. Even then the clock provided a resolution of only 1 second in standby mode.

The SSC provides a much more desirable solution. Its 32-bit VAX-standard time-of-year clock, driven by an external 25.6 kilohertz (KHz) oscillator, increments every 10 ms. As with all SSC-internal registers, the microprocessor can access the time-of-year clock without using any external logic.

To further minimize cost and module space usage in systems where battery-backed-up clock operation is not required, the user may simply ground the 25.6 KHz input pin on the SSC. During normal operation, the time-of-year clock will automatically derive its timebase from the chip's UART timebase, removing the need for the 25.6 KHz oscillator on the module.

Other Support Features

Programmable Address Strobes

As noted in the section Background and Goals, the SSC is designed to provide system designers with "hooks" to other system functions. One of these hooks is the SSC programmable address decode strobe function, which adds user flexibility and also saves module space.

Virtually every CPU module needs logic that watches the bus for particular addresses and asserts signals when these addresses are sensed. This function is typically embedded in gate array logic or in dedicated programmable array logic (PAL) chips.

The SSC has two programmable address decode strobes. The user may program each strobe for a particular address of 1s, 0s, or "don't cares." The user can also program selectively for read or write transactions. When a strobe channel is enabled, the corresponding output pin will assert during any bus transaction for which the pro-

grammed address and transaction type are matched.

The strobes can be programmed either to provide a hook for external logic or to complete a transaction after a delay. When the SSC is programmed to provide a hook, the strobe might be used to drive an external address decoder or to enable another chip. After asserting the output strobe, the SSC takes no further action, permitting another device to complete the bus transaction.

Alternatively, a strobe can be programmed to complete the transaction after a delay that permits an external device several hundred nanoseconds to respond. When configured in this way, the strobe is usually programmed to respond to reads of a single longword address. The strobe is then wired to enable three-state drivers which drive module data onto the CVAX bus. This data is often made up of external registers, or of dual in-line package switches that indicate baud rate selection and other module-specific information.

Output Port

Four pins on the chip function as an output port. The port is written as a register and is capable of driving simple output devices. This output port is another general-purpose feature that system designers need to implement various module-specific functions. Some designers use the port pins to drive LEDs, which are then flashed in a particular sequence to indicate progress of self-test. In other applications, system designers have used these signals to control external multiplexers and to provide simple modem control.

Bus Reset

The VAX architecture requires a reset of the I/O system when the CPU issues a write to a particular external processor register. This specification requires support from both decoding logic and I/O system reset logic. In the past, each module designer had to implement both logic blocks in external hardware. SSC designers saw another opportunity to simplify the CPU module by placing some of the consistently required logic on the SSC.

Although the I/O system reset logic varies among systems, the decoding logic is the same in each MicroVAX system. The SSC provides this core logic, taking three actions. First, the chip decodes the external processor register number. Then it asserts an output pin in response to the external processor register write. Finally, it

delays the completion of the write transaction for several hundred nanoseconds, so that module-specific logic, triggered by the pin assertion, may proceed to take the proper action to complete the I/O system reset.

Standby Mode: Power-sensing Features

When powered down, VAX systems are required to maintain a running real-time clock for at least 100 hours. Retention of some memory is also desirable. As noted in the section Standby RAM, the SSC satisfies these requirements by providing a standby operating mode. In this mode, the power supply to the module and to the chip pad drivers is turned off and most internal logic is disabled. However, the SSC RAM and time-of-year clock are powered by three NiCad batteries supplying between +3.1 V and +4.5 V at approximately 150 microamperes. The batteries also power the 25.6-kHz external low-power CMOS oscillator, which provides the time-of-year clock timebase. Within the SSC, special logic guarantees smooth transitions from normal operation to standby mode.

As part of providing standby operation, the SSC must reliably report at boot time whether standby power was continuously maintained during the standby period. The task of determining whether battery power had remained stable during the standby period was a difficult challenge for the SSC designers. There are two ways power can be lost during standby: The batteries may run down, or someone may replace the batteries. In either case, the SSC detects loss of power and reports such loss to the CPU during the next boot. Except for external logic used for voltage measurement, this entire function is implemented within the SSC as follows.

When the batteries run down, the unacceptably low voltage can be detected during boot. However, our CMOS process is not optimized for the design of logic that can accurately measure intermediate voltages. Thus, external circuits are used to detect whether battery voltage is currently below a minimum level. If voltage is below minimum, these circuits assert an SSC input pin dedicated to this function. However, these external circuits cannot detect temporary power losses that occur during standby mode, for example, when the batteries are replaced. To provide for these cases, a special latch on the chip, which powers up in a preferred state, detects the interruption of battery power during standby or initial power-up. This power-up

detector latch will operate for arbitrarily slow supply transitions. In addition, the latch's reset input includes internal filtering for protection against fast supply transitions or power-up noise.

If either the external circuits assert the SSC input pin or the special power-up latch indicates a loss of power, the SSC sets an internal flag bit at boot time. The bit, which indicates that the clock and RAM are not valid, is read by the microprocessor during boot.

System reliability is improved by the SSC's ability to determine the integrity of its standby logic and to notify the CPU in a software-accessible fashion. Moreover, this feature saves design time, since designers need not individually create this tricky but necessary logic.

Flexible Addressing

The designers of the SSC determined that the chip should fit into any VAX system environment with a minimum of external address decoding or system incompatibility. As a result, the SSC control and status registers and internal RAM are all situated within a relocatable 2KB address space. This arrangement eliminates the need for an external chip-enable pin and the external decoding logic that would be needed to properly assert such a pin. The power-up boot code programs the base address of the registers by writing a 2KB-aligned value to the SSC base address register.

The SSC base address register is located at a single fixed address, chosen in cooperation with our major users. The SSC RAM and registers can then be addressed by adding their specified offsets to the value in the base address register. A system designer can therefore situate the SSC registers and RAM (together) anywhere in a system's I/O space map.

Initialization

To make the SSC especially easy to use, most of the SSC configuration bits are grouped in a single register. These bits include setup for the UARTs, programmable address strobes, ROM packing, and halt-protection features. Thus, during system initialization, most SSC features can be configured with a single write.

MicroVAX and Multi-speed Compatibility

Although targeted primarily as a companion to the CMOS VAX CPU, the SSC is also compatible with the older NMOS MicroVAX CPU used in the

MicroVAX II. Thus, new low cost or low performance designs using the older microprocessor chip can also take advantage of the high integration and extra functionality provided by the SSC.

The SSC is also compatible with modules that have either high or low cycle times. Originally designed for a 100-ns microcycle, the CVAX microprocessor runs at 90 ns in the MicroVAX 3000 system and at 80 ns in the VAX 6200 system. Early in the development of the CVAX chip set, we decided that chips that were not performance-critical, like the SSC, would run at just one speed (100 ns), but would be capable of interfacing to a faster-running microprocessor. Speed conformability would simplify development, manufacturing, and field support because one SSC could be used across all MicroVAX systems.

Accordingly, the SSC bus interface, running at a 100-ns microcycle, accommodates microprocessors running at microcycles from 100 ns to 60 ns.

Summary

The SSC project yielded a CVAX microprocessor companion chip that provides a high degree of functionality, flexibility, and integration. Comprising console support, timers, decoders, and other programmable features on a single chip, the SSC permits system designers to develop smaller, more integrated modules at lower cost. Moreover, improvements made to the generalized features, such as halt protection and break detection, contribute to increased system reliability without reducing system design flexibility.

The utility of the SSC is evidenced by plans to include the chip in over a dozen different Digital products, such as the MicroVAX 3000 systems, the VAX 6200 systems, many XMI adapter boards, and various controller products.

Acknowledgments

The author wishes to acknowledge Brian R. Allison, Barry Maskas, Robert McNamara, Jay T. Nichols, Michael H. Phipps, and Robert M. Supnik, who contributed to the development of the SSC functional specification.

Further, the author acknowledges the considerable efforts of the SSC design and manufacturing team: Robert A. Anselmo, Nannette M. Fitzgerald, Robert J. Flanagan, James D. Gorr, Dennis E. Hodges, Keith D. Johnston, Balakrishna Joshi, Joseph R. Mantos, John W. May, Michael J. Saldana, and Nicholas D. Wade.

References

1. T. Fox, P. Gronowski, A. Jain, B. Leary, and D. Miner, "The CVAX 78034 Chip, a 32-bit Second-generation VAX Microprocessor," *Digital Technical Journal* (August 1988, this issue): 95-108.
2. B. Allison, "An Overview of the VAX 6200 Family of Systems," *Digital Technical Journal* (August 1988, this issue): 10-18.
3. G. Lidington, "Overview of the MicroVAX 3500/3600 Processor Module," *Digital Technical Journal* (August 1988, this issue): 79-86.
4. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1987).

Development of the CVAX Q22-bus Interface Chip

The CVAX Q22-bus interface chip (CQBIC) is a highly integrated, single chip that serves as the interface between the CVAX microprocessor and the Q22-bus I/O subsystem. The CQBIC VLSI design is the first produced by Digital's Japan Research and Development Center in coordination with teams in the U.S. Before implementing the interface design, team members built a test chip to ensure the feasibility of a CMOS Q22-bus transceiver and to test various design alternatives. Also as part of their research effort, they examined alternative designs for several functions, including the scatter-gather map cache and the data buffering functions. Project designers then implemented the CQBIC using a mix of full custom and semicustom design databases. A description of the five major functional sections is presented in this paper.

The CVAX Q22-bus Interface Chip (CQBIC) is an evolutionary step in functionality and integration from the MicroVAX II CPU module's Q22-bus interface design. The MicroVAX II CPU module's Q22-bus interface comprises 18 discrete chips and a gate array; the module design employs linked sequential controllers.¹ The advanced CQBIC design integrates these controllers and all other interface functionality in a single chip and retains the linked controller design.

Specifically, the CQBIC provides the electrical and functional interface between the 32-bit CVAX microprocessor and the 16-bit Q22-bus I/O subsystem. Integrated on the chip are the complete Q22-bus interface, data buffering, the CVAX bus,² direct memory access (DMA) interface, a scatter-gather (S/G) map cache, and complex control logic. Table 1 lists the chip's physical characteristics.

Begun in February 1985, the two-year CQBIC project was a joint venture for three of Digital's groups: Japan Research and Development Center, Large Scale Integration (JRDC/LSI); Semiconductor Engineering Advanced Peripherals Development (SEG/APD); and Micro Systems Development (MSD).³

Project Goals and Organization

A highly integrated, single-chip, CVAX bus to Q22-bus adapter was a desirable product for sev-

Table 1 CQBIC Physical Characteristics

Process	2-micron drawn, N-well, dual aluminum CMOS
Number of transistors	40,900 (approx.)
Die size	9.2 mm × 9.4 mm
Power consumption	1.5 W
Packaging	132-pin surface-mountable chip carrier with 25-mil lead spacing and heat sink
Power supply	+5 V

eral reasons. Primarily, such a chip would reduce component costs and system module size, and increase system reliability as compared with the MicroVAX II CPU module's Q22-bus interface.

Therefore, the primary goal of the CQBIC project was to develop a highly integrated chip as an interface between the CVAX microprocessor and the Q22-bus. This chip would ease the task of Digital's system designers by standardizing the interfacing to the Q22-bus and by providing the same or improved I/O bandwidth performance as the MicroVAX II CPU module Q22-bus interface.

Achievement of this performance goal was complicated by the single-port memory architecture of the first planned CPU module and its two-level instruction and data, direct-mapped cache scheme. In comparison, the MicroVAX II CPU

module has a dual-ported memory architecture with no caching. However, the DMA single-port architecture was required for the new two-level cache architecture; with a single-port organization, DMA addresses can be viewed by the caches so that the caches can invalidate valid entries during I/O-to-memory write transactions. Consequently, to both accommodate this architecture and meet its performance goals, CQBIC had to be designed to consume little CVAX bus bandwidth while performing DMA transactions. Such a design would not greatly degrade CVAX microprocessor performance.

A second important project goal was to preserve I/O performance and operating system software compatibility.⁴ Therefore, CQBIC would provide the same Q22-bus virtual to CPU physical memory address translation as contained on the MicroVAX II CPU.

In addition to meeting these goals, the CQBIC project would also serve to demonstrate the feasibility of a remote VLSI design center for the SEG organization. Moreover, through this project the JRDC/LSI Group would have an opportunity to demonstrate its VLSI design capabilities.

Further complicating the challenges presented by the design goals, the distance between the working groups, the cultural and work style differences, and the language barrier was the newness of the JRDC team. Many of the JRDC team members could read and write English, but had some difficulty speaking and listening to English. Also, the Japanese language was completely foreign to MSD and SEG. Written English served as the primary form of communication throughout the project. Further, the JRDC team members had to learn not only about Digital's products and architectures, but also the Q22-bus, the other five chip specifications under development, the SEG semicustom and custom chip design tool suites, and Digital's CMOS technology. To help with this steep learning curve, experts from each of these areas facilitated the training and information flow. These experts provided answers to specific questions and helped to solve specific problems as follow-up to formal training sessions.

Based on the MicroVAX II CPU design experience in SEG and MSD, SEG provided leadership for both the chip specification development and the project. This role involved conveying to the JRDC team the chip functional definition and detailed behavior specifications. This information had to

be presented in the context of the five other VLSI chips being designed by the SEG groups with a focus on the CPU module product. The U.S.-based project leadership had to provide budget, schedule, and task coordination for JRDC, MSD, and for other organizations within SEG.

As the initial customer, MSD performed three major specification reviews. This group continually provided direction concerning design trade-offs, and requested specific functionality revisions to tailor CQBIC more to their CPU application.

Digital's Engineering Network was the primary means of transferring written communications between groups. We also exchanged information by sending facsimile copy and by mailing magnetic tapes and documents. At times telephone discussions and personal visits were necessary.

Specification development began with a two-week visit to the JRDC facility in Tokyo. At that time, we wrote the first draft with key members of the JRDC team. This draft specification laid the foundation for subsequent architecture and functionality research, and served as a communication medium. The draft specification was then maintained by the JRDC team and SEG and was frequently revised and reviewed.

The following section presents the project research conducted to ensure the feasibility of project goals and to resolve major questions raised by the draft specification.

Project Research

Project research focused on two areas. First, we wanted to evaluate the risks involved in the implementation of a CMOS Q22-bus transceiver. For this purpose, SEG team members implemented a test chip. Second, we wanted to determine the best means to achieve our stated performance goals. The tests and studies which we conducted and their results are described below.

Q22-bus Transceiver Test Chip

To determine whether or not a CMOS Q22-bus transceiver could be implemented, several studies were performed by SEG circuit designers responsible for the cell library. These studies showed feasibility, with two major implementation risks:

- The proposed differential comparator to be used as the receiver required a stable voltage reference.

- The 33 100-milliampere (mA) peak, 70-mA steady-state sink current Q22-bus transceivers were to be on the same substrate as complex control circuitry. Three problems could result:
 - CMOS latch-up due to charge injection from input signal overshoot
 - Excessive noise due to substrate current transients
 - Excessive localized power dissipation

With several design alternatives available to us, we needed more experimental data to determine the better alternatives. To obtain this data, a Q22-bus octal transceiver test chip was designed, fabricated, and packaged by SEG circuit designers. Available after seven months, this packaged octal transceiver test chip was tested in a MicroVAX II CPU module and performed well under system conditions.

The test chip experiments showed that CMOS latch-up due to worst-case overshoots below ground did not occur. These results matched our expectations. We were not concerned with overshoots above the +5 volts (V) bias because of the Q22-bus termination voltage of 3.4 V. Tests also showed that special care would be required in the allocation of dedicated ground pins for the Q22-bus transceivers to avoid noise coupling from substrate bounce and package power-lead inductance. Also, in the chip layout, we would have to use many parallel traces of metal interconnect to prevent metal migration when sinking 100 mA of peak current. Finally, due to low channel resistance of the Q22-bus driver output pull-down device, the power dissipation of the test chip was shown to be within reliable operation limits. Therefore, CQBIC power dissipation was not a concern in terms of thermal characteristics of the planned packaging.

The test chip results did lead to a compromise concerning the stable voltage reference. Because of large variations in CMOS process materials, a precision off-chip or external resistor would better serve to establish the required voltage than would some risky process-desensitized structure in CMOS.

Prior to these tests, we designed CQBIC to facilitate the use of either integral transceivers or off-chip transceivers. Fortunately, the test data demonstrated the feasibility of a single chip with integral Q22-bus transceivers, and the project

proceeded under a plan that included integral transceivers.

Architecture and Performance Studies

As the octal transceiver test chip was being developed, MSD, JRDC and SEG conducted architecture and performance studies. These studies would answer questions about the organization of the S/G mapping function, the data buffering required to meet the performance goals, and the sequential controllers partitioning and clocking to manage the two asynchronous buses and the internal functions.

S/G Mapping

A RAM structure was first proposed to implement the S/G mapping functionality. The MicroVAX II CPU design had used such a structure, with two 8K-by-8 static RAMs. This proposal, however, was rejected since not all of the RAM would fit on a single chip with all the other required circuitry. Increasing the chip size was not an option. The chip size was limited for cost reasons as well as packaging cavity size reasons. The chip's cost is directly proportionate to its size, and the design of a new package was outside the scope of the project. Moreover, implementation of a portion of the RAM would have introduced a system software incompatibility with MicroVAX II and would have reduced the planned performance.

As the problem of S/G mapping functionality was studied, it became clear that system memory was adequate. Further, CQBIC could not implement the full 8192-entry RAM on a chip size that could be fabricated with reasonable yield. Also, a capability to prefetch S/G map entries based on expectation was considered necessary to sustain peak, as opposed to average, performance. We looked to the Q22-bus DMA devices which perform transactions with incrementing addresses. In particular, Q22-bus devices are designed to utilize the Q22-bus block-mode data transfer protocol. This protocol transfers data packets of eight-word blocks. With this protocol available, we could design the CQBIC to cache the S/G map entries from system memory on demand and on expectation.

The next two problems were how to implement the cache and how many entries to include in the cache. A 16-entry cache provided the balance we sought between several factors: appropriate chip area, implementation complexity, design risk, and DMA I/O performance impact.

Data Buffering

CVAX bus cycle times were targeted to be four or more times greater than typical Q22-bus cycle times. Also, the CVAX bus was being designed to support DMA multidata transfers. This design was consistent with the Q22-bus block-mode data transfer protocol. To bridge the bandwidth gap between the two buses and to minimize the use of CVAX bus bandwidth, data buffering techniques were investigated to optimize for Q22-bus block-mode throughput for read and write transactions. These investigations resulted not only in the determination of buffer sizes but also in a decision on how to control the buffers to optimize sustained throughput and minimize initial latency.

The MicroVAX II CPU is capable of supplying read data to the Q22-bus with a very consistent access time because memory arbitration is not required. To achieve MicroVAX II average read performance, read data prefetching was considered necessary to compensate for the memory arbitration time. For CQBIC, the first read of a Q22-bus transaction would be time delayed by the DMA request and grant time, to obtain mastership of the CVAX bus, and by the subsequent system memory access time. The delay would always be longer than MicroVAX II read latency, which had only memory access time read latency to consider. We determined that two quadword read buffers would be sufficient to sustain the required throughput because read data is prefetched based on expectations of the Q22-bus block-mode protocol. Low latency was achieved by providing a response to the Q22-bus as the first longword of the quadword read data was obtained from system memory.

Pipelining the buffered write data could be achieved with two buffers, each eight words deep. An octaword block is the packet size of the Q22-bus block-mode protocol and is the maximum multitransfer block size of the CVAX bus. The control logic would be designed to allow one buffer to be unloaded to system memory while the other was being filled. The latency would be better than that of the MicroVAX II CPU module, since the CQBIC data was packed into fast octaword buffers. The average throughput would be sustained by the four times or greater bandwidth of the CVAX bus, as compared to the Q22-bus, by the use of pipelined data buffers.

The CQBIC buffering and transaction optimizations in conjunction with the CVAX CPU internal

cache hit rate result in an insignificant DMA I/O impact on CVAX CPU performance. Given the buffering and control organization and optimizations described above, performance difference between the single-port and the dual-port memory designs cannot be detected by a Q22-bus device. The result is improvement in Q22-bus read and write throughput over the MicroVAX II CPU. The CQBIC maximizes Q22-bus performance and minimizes CVAX bus usage. Moreover, CQBIC can sustain Q22-bus block-mode transfer write data rates of 3.3 megabytes (MB) per second and read data rates of 2.5 MB per second.

Finally, to optimize the CVAX I/O write performance, a dump-and-run buffer was to be implemented in CQBIC. This buffer is used to avoid tying up the CVAX bus while the slower Q22-bus transaction completes and while deadlock situations are resolved.

Controller Partition

Given these buffering functions, the control of the data path and of the two major bus interfaces was naturally partitioned into five linked controllers and a prioritization function. Each bus interface was partitioned into a master and a slave controller. The S/G map cache also required a controller. Then to assist in coordination of control flow decisions, a priority resolver function was needed.

This partition allows the Q22-bus and the CVAX bus to operate in parallel while all deadlock conditions are resolved. Fortunately the CVAX chip team implemented a bus transaction retry capability. This retry capability proved essential to our partition and implementation of CQBIC control functionality.

Clocking

Two primary factors led us to select a 50-nanosecond (ns) two-phase nonoverlapped internal clock scheme. First, the MicroVAX II CPU module's 50-ns single-phase clocking scheme was a proven approach and mapped well to the fixed Q22-bus minimum asynchronous timing specifications. Second, we expected synchronous CVAX bus cycle timing to vary with CMOS technology improvements. The variable CVAX cycle time and four-phase overlapped clocking scheme could not be used to generate the fixed Q22-bus timing. Also, having two clocking schemes in one chip was determined to be a design too complex to manage.

The implication of the selected CQBIC clocking scheme was that, with reference to all internal controllers, the CVAX bus and the Q22-bus were asynchronous.

Research Results Summary

The result of the research was a single chip design that would achieve the stated project goals by providing

- Integral Q22-bus transceivers
- A 16-entry map cache, with prefetching
- Two octaword Q22-bus write buffers
- Two quadword Q22-bus read buffers, with prefetching
- A longword CVAX write buffer
- Transaction partitioned sequential controllers, which are optimized for look-ahead data buffering control and for utilization of multiple-transfer transactions to minimize CVAX bus and Q22-bus usage

The research results were documented in the form of a revised chip specification and a behavioral model. The chip was implemented from the revised specification with a process which was unique and unproven.

Implementation Process

CQBIC was implemented using a mix of standard library cells, custom library cells, and full custom layout sections. At the time, SEG could not offer a formal design tool suite to deal with such a mix of full custom and semicustom design databases. So the JRDC team standardized by selecting the methods of the semicustom tool suite for logic and circuit design. The semicustom schematic editor and wire lister were used to design all the logic. This wire lister facilitated interfacing to SPICE and other checking tools and most importantly to the layout tools. For layout, no automation of floor planning and cell placement and routing could be employed. This layout was all done by hand, as were the full custom designs. Interconnect verification and design rule checking were completed using the tools from the custom design suite.

A full custom layout section was required to implement the S/G map cache because of the chip-size and latency constraints. A part of the latency is due to the Q22-bus address look-up in the cache. The S/G latency had to be small to com-

pensate for the long latency that could occur, for example, when the look-up misses the cache and requires an S/G map memory read access.

The standard cell library was rejected because it did not offer a content addressable memory (CAM), which is the structure required to facilitate fast address look-ups. In addition, the use of standard library cell latches and exclusive OR gates was estimated to almost double the desired look-up time on the 16 cached entries.

Again to contain chip size and also to meet control performance, custom programmable logic array (PLA) sections were required. The PLA structures offered by the standard cell library were too slow and required a clocking scheme different from the CQBIC two-phase clocking scheme. This decision to implement custom PLA structures is credited as the reason performance goals were achieved. In fact, performance goals could not have been achieved without custom PLA structures.

At the time logic and circuit design began, the standard library cells available for this design were found to be inadequate. Many necessary functions were missing or were not tailored for the specific application. Also, in many cases the performance of library cells did not match the performance required by the two-phase clocking scheme. Hence the JRDC team developed its own extensions to the standard cell library. The common logic structures such as NAND, NOR, flip-flop, and latch were used from the standard cell library as much as possible, since these structures reduced the risk of circuit problems. Custom structures, such as counters, multiplexers, latched pad transceivers, synchronizers, PLA AND plane drivers, and PLA OR plane receivers, were designed and made available to the library.

The JRDC team accurately modeled the chip based on the specification at the behavioral and the MOS levels of abstraction using Digital's DECSIM simulator.

Initially, the JRDC team developed a behavioral system environment model based on their understanding of the CVAX bus and the Q22-bus specification. This environment model was layered around the CQBIC behavioral model to verify the design. As the design progressed, a more accurate behavioral chip model replaced the initial model after correlation.

Further, as other CVAX behavioral, structural, or MOS chip models matured, MSD incorporated them into the CPU system model. This model was

then used to test the CQBIC further in the context of the application system. System simulation proved that all CVAX bus specifications which were communicated were understood and implemented correctly. The system simulation served as an independent test of the CQBIC design. Although no CQBIC problems were found by MSD during system simulation, the testing did prove that the system would operate. We later learned that several bugs could have been found had more time-varied events been scheduled with the system simulation test cases.

When completed, the CQBIC MOS model was correlated to the behavioral chip model. The MOS chip model was then placed in the MSD system model for regression testing.

When we were confident that the CQBIC design was complete, that is, when no new bugs were found after thorough testing, the chip was released to SEG for a final design review and submittal for fabrication. The database was copied over the Engineering Network from the JRDC facility in Tokyo to the Hudson, Massachusetts, plant. After completing a final design review and subsequent problem fixes, the chip was submitted for fabrication. Eight weeks later first-pass parts were probed and found to be functional. Packaged parts were run in the MSD CPU module. This testing revealed several timing bugs related to events from both buses occurring at the same time. After extensive testing, the bugs were fixed, and a second revision was released for fabrication. When the second pass part was tested in the CPU module, another timing problem related to coincident transactions from both interfaces surfaced. This particular bug was obfuscated by a pass 2 bug. A third revision was prepared and fabricated. This third pass was available in time for the first customer shipments. The final chip functionality is briefly described below.

The CQBIC Functional Organization

CQBIC is an asynchronous CVAX bus device and requires a fixed 40-megahertz oscillator input to drive Q22-bus timing. The oscillator input is used to generate a two-phase, nonoverlapped clock which is distributed to all chip sections. The CVAX bus interface was designed to accommodate transaction cycle times from 100 ns to 60 ns. This design anticipated a CVAX CPU technology change and subsequent performance improvement.

CQBIC provides the power-up, initialization, power-fail, and power-down protocols to the system and performs Q22-bus and CVAX bus address decoding. Further, the chip performs the page address S/G mapping function for DMA devices by using its 16-entry S/G address map cache.

This cache contains a copy of the most recently used S/G pointers, which are located in system memory. The cached pointers are used to map 22-bit Q22-bus virtual to 29-bit CVAX bus physical addresses. CVAX bus and Q22-bus transactions are optimized by using a CPU dump-and-run write buffer, two pipelined Q22-bus octaword write buffers, and two pipelined Q22-bus quadword read buffers. The chip performs transparent address and data alignments, and packing and unpacking of internal buffers.

CQBIC is composed of five global control sections. A block diagram of the chip control sections is shown in Figure 1.

Each section contains an independent sequential controller:

- The Q22-bus arbiter
- The S/G map
- The Q22-bus master
- The Q22-bus slave and CVAX bus master
- The Q22-bus electrical interface.

A photomicrograph showing the floor plan of the control sections is shown in Figure 2.

Each section shown in the Figure 1 block diagram is described next.

Q22-bus Arbiter Section

As a Q22-bus arbiter, the CQBIC is the default Q22-bus master and the highest priority requester. The arbiter accepts requests from Q22-bus DMA devices and from the master section, and grants mastership with first priority to the master section. In response to a master request, the arbiter exercises a demand mastership protocol to Q22-bus devices to ensure low-latency interrupt vector or data reads. In response to interrupt requests from the Q22-bus, the arbiter receives the requests and passes them to the CPU. When the CPU acknowledges the request, CQBIC reads a vector from the Q22-bus device and supplies an acknowledge signal.

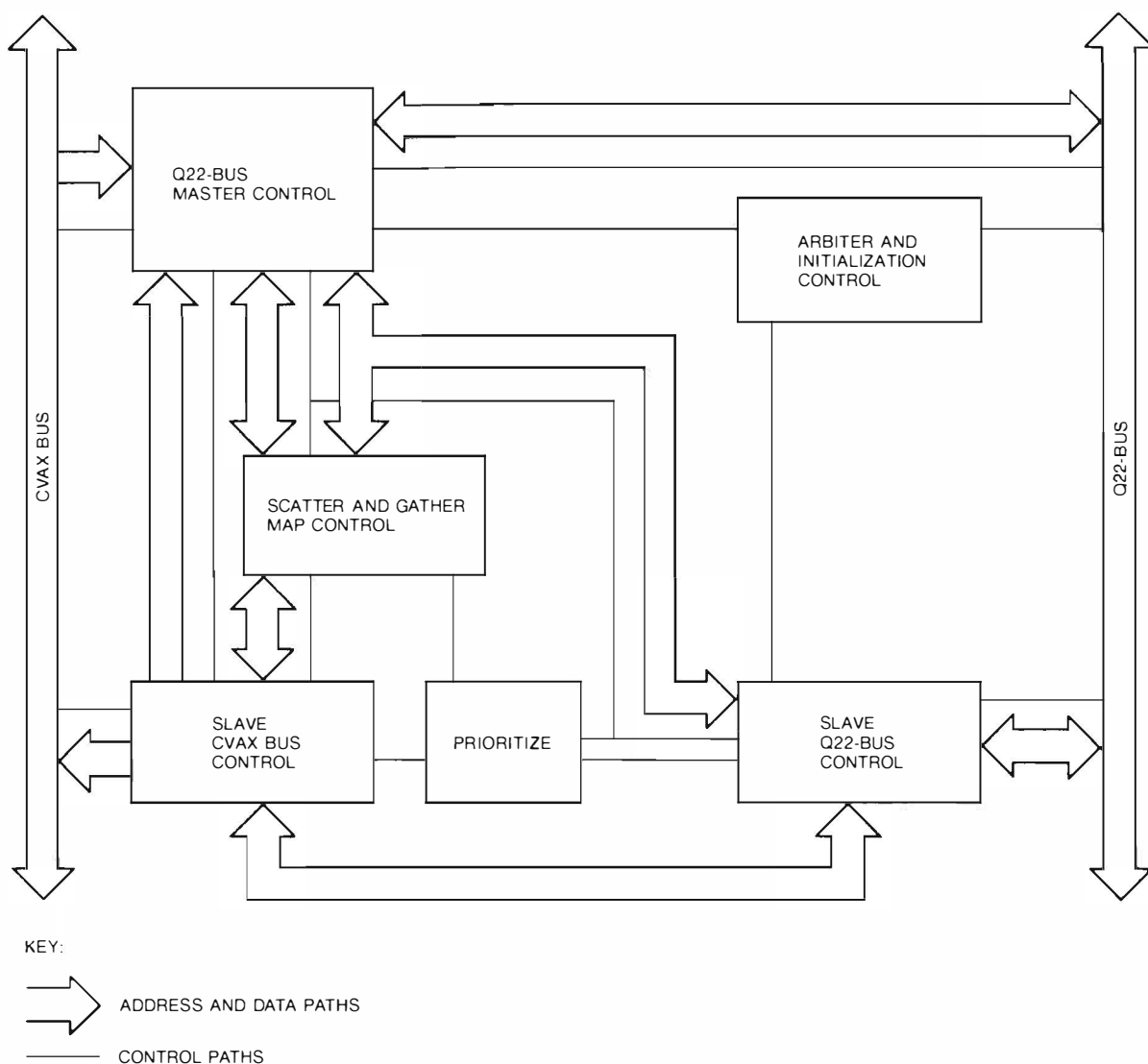


Figure 1 Control Section Block Diagram

When multiple CQBIC chips are connected to the Q22-bus, they take on different functions. The first chip operates as Q22-bus arbiter; the others operate in auxiliary mode. As an auxiliary mode device, a CQBIC chip does not perform Q22-bus arbitration. Instead, the chip behaves as a typical Q22-bus DMA device that is a default Q22-bus slave. Therefore, when the CPU initiates a Q22-bus transaction, its CQBIC requests Q22-bus mastership. The arbiter CQBIC serves as Q22-bus arbiter and grants the bus accordingly to auxiliary mode CQBICs and other DMA devices.

Either as arbiter or as an auxiliary device, the arbiter function performs the system power-up, initialization, power-fail, and power-down sequences.

S/G Map Section

The S/G map consists of 8,192 longwords allocated from system memory. Each map entry consists of a 20-bit page pointer, a 3-bit descriptor which CQBIC ignores, and a valid bit. The low 9 bits of a Q22-bus address pass through as an interpage offset; the upper 13 bits select the con-

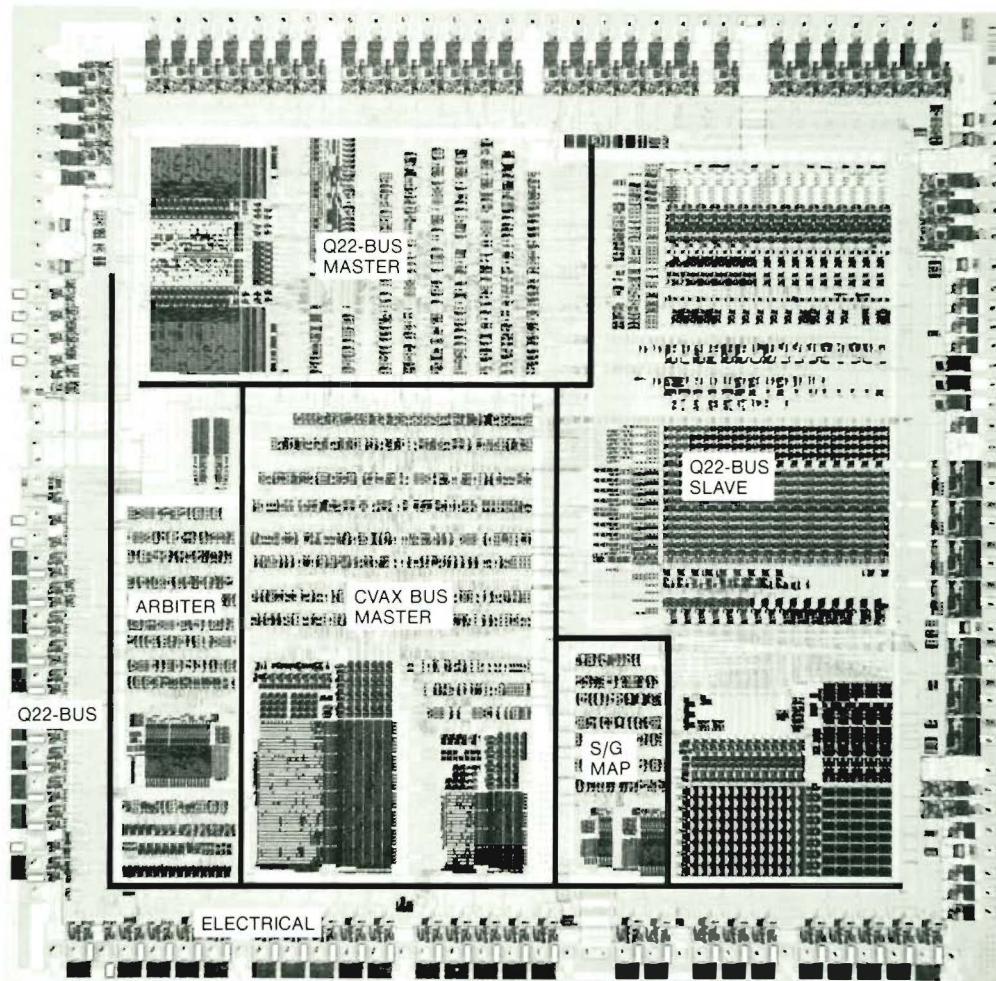


Figure 2 Photomicrograph of CQBIC

tents of one of the 8,192 S/G map locations. The CPU informs the CQBIC of the S/G map location by writing a base address into the CQBIC map base register. This write flushes the valid bits of the cached map entries.

To avoid map cache coherency problems, the CPU accesses the S/G map through a VAX I/O address range decoded by the CQBIC master section. The slave section then performs the S/G map memory transaction. This indirect approach prevents the CPU from directly modifying the S/G map memory independent of the 16 cached pointers. A CPU to S/G map write invalidates the cached map entry as the slave section performs the memory write. CPU to S/G map reads return the cached copy if it was cached or return the S/G pointer from system memory.

As noted in the section Project Research, we selected a map cache size of 16 entries. The research of Q22-bus DMA device transfer sizes and the number of devices active in a dynamic system showed that 16 entries were sufficient to avoid thrashing on entries. The effects of the Q22-bus fair arbitration scheme were used to show that the simple first-in-first-out (FIFO) replacement algorithm selected did not waste performance and was consistent with incrementing DMA device addresses. As a DMA device transfer address incremented to a page boundary, the next map entry would be prefetched, and the previous map entry was not used unless the current I/O request completed and another was requested. We found that the operating system's allocated map entries for I/O requests to Q22-bus DMA

devices from a free pool list maintained in a last-deallocated, first-allocated manner. The overhead of one extra read for a map entry per page was found to be insignificant.

Q22-bus Master Section

The master section contains two configuration registers and three status-and-error reporting registers in addition to all the control circuitry.

The master section's function is to decode all the CVAX bus addresses and cycle status codes. This decoding determines which of two types of actions is required:

- A transaction to an internal register, the S/G map, or the Q22-bus
- Q22-bus mastership prior to completion of the transaction

Each of these actions is described in the text below.

If a decoded address requires no CQBIC response, a signal pin is asserted to external logic for control of buffers and timeout counters.

Transaction to an Internal Register

When the master section detects a CVAX bus address for one of the two control or three address registers in CQBIC, it returns or writes the data. The master section also facilitates a memory lock for the CPU to perform a read-lock and write-unlock operation. First, the master detects a CVAX bus interlocked transaction and then performs a retry until Q22-bus mastership is obtained. Q22-bus mastership is held until an unlock transaction or an exception occurs. As long as other Q22-bus devices follow this protocol, memory that is mapped to the Q22-bus can be shared.

Transaction to S/G Map

As noted in the S/G Map section, S/G map transactions are controlled by the master section. The master requests the slave and map cache sections to complete the memory and cache transactions. To construct the memory address for the slave and map cache, the master uses the significant low 13 bits of S/G map I/O address as an offset from the map base register.

Transaction to Q22-bus

To avoid deadlocks, the master utilizes the CVAX CPU retry transaction. (CVAX CPU relinquishes CVAX bus control to the CQBIC slave section. The

CPU then retries the same transaction when bus control is returned.) S/G map transactions have a higher priority than Q22-bus slave transactions. The slave section therefore performs S/G map transactions in parallel with Q22-bus slave transactions. When the master tries to access the Q22-bus and it is busy, the arbiter attempts to gain mastership. Until mastership is obtained, the slave can perform a retry to satisfy the Q22-bus transactions.

Q22-bus Mastership

When the master acquires Q22-bus mastership, it sequences the transaction. A special case of the sequence occurs when the I/O memory segment address maps back to system memory through the slave and map cache. In this case a retry is used, and the slave gives the data to the master.

The CPU writes to the Q22-bus are accepted by the master in a dump-and-run manner to improve performance.

Q22-bus Slave Section

The slave section design implemented the two quadword read buffers and the two octaword write buffers. This section was the key to realizing the performance goals established for the chip. The slave has to respond to all Q22-bus transactions by checking the address in the S/G map and then sequencing the CVAX bus to put or get data. The slave must coordinate its intentions with all other chip sections to avoid deadlock conditions. This coordination is realized in a prioritization circuit which receives state inputs from all sections of the chip and outputs status codes to the slave and master sections to trigger actions.

The slave watches for master or Q22-bus transaction requests. When the slave receives Q22-bus addresses, it passes these to the map cache for validation. If the S/G entry is not cached, the map cache signals the slave to acquire a new S/G map pointer from system memory. The map cache will cache this new entry if the valid bit is set. If the valid bit is cleared, then an exception is taken. When the address is validated, the slave proceeds to sequence the transaction to or from a buffer and system memory. During slave writes to the system memory, the CVAX is signaled to invalidate its internal cache.

The slave maintains two octaword write buffers to optimize Q22-bus octaword block-mode transactions. By using a CVAX bus multitransfer burst,

the slave can unload one buffer to memory while filling the other octaword buffer.

For each new Q22-bus read request, the slave prefetches four words from memory. This prefetch is done in anticipation of block-mode transactions. These four words are buffered and sent to the Q22-bus master. As the third word is unloaded, the slave prefetches four more words.

As either a Q22-bus block-mode read or write transaction nears a page address boundary, the slave performs an S/G map entry prefetch of the next entry. The slave then passes the prefetched entry to the map cache.

An additional function of the slave section is a Q22-bus addressable interprocessor doorbell register. This register accommodates arbiter and auxiliary mode operation by supplying to the CPU a memory access semaphore, an interrupt request, and a vector address.

Q22-bus Electrical Interface Section

The Q22-bus is a 120-ohm transmission line with near- and far-end parallel termination. The length of the Q22-bus can vary from 25 to 60 centimeters and is subject to reflection and crosstalk noise. CQBIC contains 33 transceivers and 9 receivers which connect directly to the Q22-bus.

The open-drain outputs and filtered inputs were designed to operate reliably in the Q22-bus environment.

The input filter rejects crosstalk and reflection noise by staging a low pass RC filter. The filter is constructed with an n-diffusion resistor and p-type field effect transistor (PFET) capacitor with a differential amplifier receiver which maintains a narrow noise immunity region.

The open-drain output driver controls the edge rates. This control minimizes transmission-line reflections and crosstalk for ac load variation from 30 to 330 picofarads, and dc termination variation of 240 to 60 ohms at 3.6 volts. To satisfy the 100 mA sink current possible on each of 33 outputs without excessive heating, low internal power dissipation was achieved by low steady-state "on" resistance.

A disable control allows the output to power down without affecting the Q22-bus.

Conclusion

A single chip Q22-bus interface was realized and is being shipped in Digital's systems as the result of the successful venture for JRDC, SEG, and MSD.

We learned how to manage efforts from a distance and to coordinate and communicate complex technical information around the globe.

Acknowledgments

The author wishes to acknowledge the technical contributions of S. Kyu, S. Akanuma, A. Abe, H. Hayashi, M. Hasegawa, M. Taiji, S. Iida, M. Kikutani, K. Koga, L. Walker, D. Grondalski, J. Lipcon, and R. McNamara.

References

1. B. Maskas, "Developing the MicroVAX II CPU Board," *Digital Technical Journal* (March 1986): 37-47.
2. P. Rubinfeld, et al., "The CVAX CPU, a CMOS VAX Microprocessor Chip," *ICCD Proceedings* (October 1987): 148-152.
3. G. Lidington, "Overview of the MicroVAX 3500/3600 Processor Module," *Digital Technical Journal* (August 1988, this issue): 79-86.
4. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1987).

The CVAX CMCTL — A CMOS Memory Controller Chip

The CMCTL — part of the CVAX family of chips — is a high-performance ECC memory controller for single-processor systems. Implemented in Digital's CMOS technology, the CMCTL is optimized to satisfy Q-bus-based system requirements. The CMCTL operates as either a synchronous or an asynchronous interface between the CVAX bus at cycles from 60 to 100 nanoseconds and the private memory interconnect. For memory read or write operations, the CMCTL supports the CVAX multiple-transfer protocol. Data parity and memory error checking is implemented for all data transfers. The chip's high performance is achieved in part by a high-speed, page-mode access protocol.

The decision to design a CVAX memory controller (CMCTL) was made in July 1984. The primary goal of the CVAX CMCTL project was to design a high-performance, single-chip, error-correcting code (ECC) memory controller for a single-processor system. This chip would be part of a CVAX family of core peripheral functions.

Several systems being developed at that time utilized the MicroVAX II CPU chip, the predecessor to the CVAX CPU chip. Because company revenue for Q-bus-based systems such as the MicroVAX II is significant and a performance benefit could be gained from a custom chip design, the memory controller design goals were focused to satisfy the requirements of a Q-bus-based system. The initial system requirements for the CMCTL were determined by studying the memory controller specifications and by discussing requirements with key members of the project team for the existing MicroVAX II system. In addition, the Electronic Storage Development (ESD) Group was consulted on the requirements of a memory controller.

Let us now examine the key aspects of the CVAX CPU chip that influenced the system requirements for the CMCTL. First, the CMCTL

had to interface directly to the CVAX bus and handle the memory transactions originating from the CVAX CPU chip. Located in the CVAX CPU chip is an integral primary write-through 1-kilo-byte (KB) cache. The size of this cache can be optionally expanded with a second-level cache function on the CVAX bus. Consequently, the CMCTL-to-CVAX bus interface had to work with or without the optional second-level cache. Furthermore, the primary cache and the optional second-level cache use byte parity for memory error detection. Therefore, the CMCTL bus interface was required both to generate and to check byte parity. For CVAX-based systems operating at 100-nanosecond (ns) and 60-ns CVAX bus cycles and implementing a second-level cache, the performance goals were respectively 2.5 and 4.0 times the performance of the MicroVAX II system. These goals governed the CMCTL bus memory performance, or memory cycle time, requirements described later in this paper. Since memory size requirements are proportional to CPU chip performance, the CMCTL had to support a memory size larger than that of the MicroVAX II. The MicroVAX II CPU memory systems have a byte-parity, memory error-detection scheme. To meet the reliability requirements for larger memory systems, the CMCTL was designed primarily as an ECC memory controller.

Since a direct memory access (DMA) function can also become the bus master on the CVAX bus, the system requirements for the CMCTL were

A shorter version of this paper first appeared in the *Proceedings of the 1987 ICCD: VLSI Computers and Processors*, October 1987 entitled "The CVAX CMCTL. A CMOS Memory Controller Chip" by D. Morgan, K. Chui, J. Clouser, S. Nadkarni, and R. Strouble. Copyright 1987, The Institute of Electrical and Electronic Engineering, Inc.

influenced by these functions also. Because the CVAX CPU chip performs only synchronous transactions and a DMA function could be either synchronous or asynchronous, the CMCTL is designed to run as a synchronous or asynchronous slave on the CVAX bus. Further, the CVAX CPU chip can handle only two of the possible four types of data transfer lengths on the CVAX bus. However, a Q-bus DMA function (CQBIC) needed to generate all four possible data transfer lengths in order to efficiently handle data transfers between the Q-bus and the CVAX bus which have data widths of 16 bits and 32 bits, respectively. The requirement to work with the Q-bus DMA function meant that the CMCTL needed to handle all four data transfer lengths. In addition, since a DMA function could optionally generate and check parity, the CMCTL had to be flexible in this regard as well. Finally, the CVAX CPU chip executes interlocked instructions which must have the effect of "locking" or "unlocking" the memory from DMA read-modify-write transactions. Interlocked memory transactions are not defined in the Q-bus protocol. Therefore, interlocked memory transactions are handled with a bus interlock scheme. In this scheme, the CQBIC stalls, i.e., RETRY, the CVAX CPU chip memory read lock bus transaction on the CVAX bus until it becomes the Q-bus master first — locking out I/O to memory — before the CVAX can perform interlocked instructions. RETRY is a slave response to a bus master on the CVAX bus that tells it to retry the bus cycle because it cannot complete the requested operation. The CQBIC releases the Q-bus after it sees a CVAX CPU chip memory

write transaction on the CVAX bus that signals the termination of the interlock instruction.

Certain base technology constraints influenced the CMCTL specification. First, the high performance requirements for memory in a system that does not implement a second-level cache determined that the CMCTL be implemented in a single custom chip. At the time, it was not possible to implement a memory controller with the required speed in a commercially available gate array that would run synchronous with the CVAX CPU chip. Furthermore, in a Q-bus-based system, memory expansion occurs in the Q-bus backplane. Therefore, a single memory controller that resides on the CPU module and controls the memory by means of signals on the backplane is the simplest and most quickly implemented system solution. Another factor that influenced the single-chip alternative solution was the limited space available on the CPU module that implements a second-level cache. Taken together, these factors ruled out the possibility of designing a slower memory controller using commercially available memory controller components for systems that implement a second-level cache. The availability of CMOS-1 technology in Digital's Hudson, Massachusetts, facility in 1984 drove the design technology choice.

System Overview

The CVAX CMCTL is the core control function of a single CVAX CPU memory system. This chip serves as the interface between devices on the CVAX bus and a CMOS private memory interconnect (PMI). Figure 1 shows the major interfaces

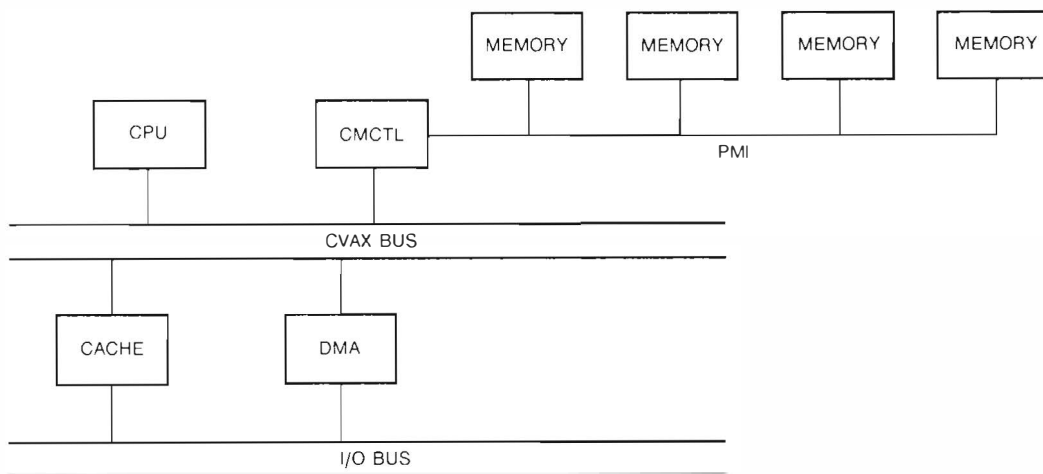


Figure 1 Major Interface Connections of the CMCTL Chip

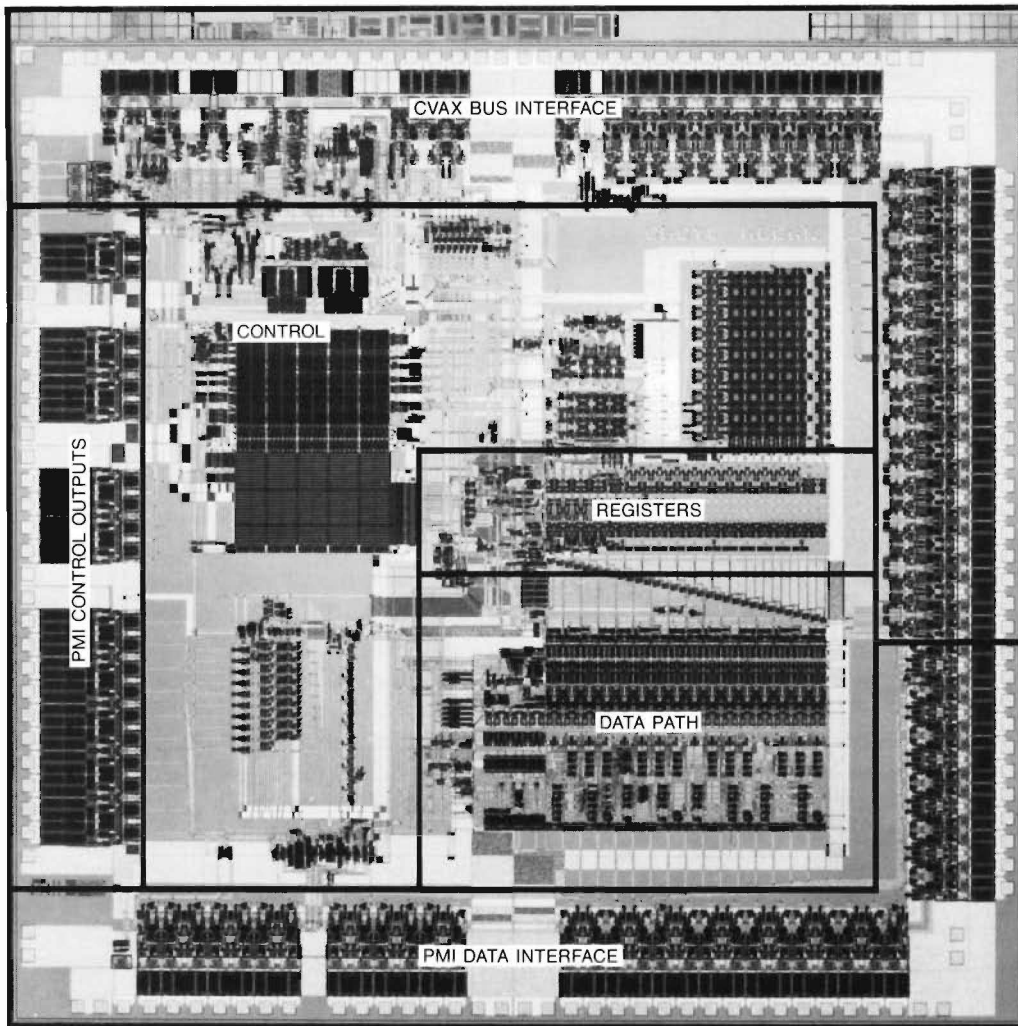


Figure 2 Photomicrograph of CMCTL Showing Major Sections

of the CMCTL in a CVAX system, and Figure 2 shows the major sections of the chip. Table 1 lists the physical characteristics of the chip.

This section presents a brief overview of the CMCTL chip's two major interfaces, data transfer support, and error-checking and notification features.

CMCTL Major Interfaces

As interface to the CVAX bus, the CMCTL responds as either a synchronous or an asynchronous slave device. When the CVAX CPU chip is bus master, the CMCTL responses are synchronous. When a DMA device is bus master, a bus-mode signal determines whether the chip responds as a synchronous or asynchronous device.

The CMCTL connects directly to its other major interface, the PMI. The PMI consists of control, address, and data signals which interconnect the CMCTL and the memory array modules. Through

Table 1 CMCTL Summary Characteristics

Process	2-micron drawn, N-well, dual aluminum CMOS process
Number of transistors	20,000
Die size	7.6 mm × 8.0 mm
Power dissipation	1.5 W worst case
Packaging	132-pin surface-mountable chip carrier with 25-mil lead spacing
Power supply	+5 V

these interconnections, the chip controls up to four memory modules, each containing one, two, or four banks of dynamic random-access memory (DRAM). Each memory module is required to buffer all the PMI signals.

Data Transfer

The CMCTL fully supports the CVAX bus multiple-transfer protocol and can perform one to four data transfers on a memory read or write operation. Each data transfer can have up to four bytes of data. Since ECC is generated across four bytes, write data with less than four valid bytes will cause the CMCTL to do the actual memory write on the PMI as a read-modify-write cycle. Otherwise, the write data goes directly to memory.

Error Checks and Notification

The CMCTL performs two error-checking functions:

- CVAX bus data parity error checks
- Memory error checks

To assist with the error checking of data transfers on the CVAX bus, the CMCTL checks data parity on memory writes. The chip generates parity with the data on memory reads.

For data transfers on the PMI, the CMCTL has two memory error-checking modes: 7-bit ECC, and single-bit parity. In ECC memory error mode, the CMCTL detects double-bit uncorrectable memory errors and detects and corrects single-bit memory errors. In parity memory error mode, the CMCTL can detect single-bit memory errors.

The CMCTL uses four outputs to notify the CVAX bus master of four error conditions. These error-condition notices are as follows:

- The bus transaction was successful and completed with no errors.
- The memory data transfer resulted in an uncorrectable ECC or parity error.
- The memory data transfer resulted in a correctable memory error.
- The CVAX CPU chip-initiated memory write had a parity error.

In addition to these four outputs, the CMCTL provides an output that indicates when the CMCTL is not going to respond to either a memory or an I/O operation. This output reduces the number of external components required to detect addresses not implemented in a system.

CMCTL Performance

The CMCTL achieves its performance in part by using a high-speed, page-mode RAM access protocol on the PMI. DRAMs that run in page mode can perform data transfers in approximately one-half the cycle time of those run in nonpage mode.

The CMCTL responds to CVAX single-transfer memory write or read operations within two or four CVAX bus cycles, respectively. During a memory read operation, the CMCTL starts a memory read access in parallel with an optional cache to increase memory read performance. If the memory read address hits in the external cache, the CMCTL aborts the read operation. The CMCTL performs memory write transactions as dump-and-run.

Table 2 lists the memory operations and the corresponding performance for synchronous data transfers with 4 bytes of data. Two numbers are shown for multiple-transfer memory operations. The first is the time in CVAX CPU bus cycles to complete the first transfer; the second, the time to complete subsequent transfers. In order to tune the memory performance across different CVAX bus speeds, the CMCTL provides a programmable mechanism for varying PMI transaction timing. For CVAX bus cycle times less than 100 ns, the CMCTL can be programmed to add slip cycles to memory read operations in increments of the CVAX bus cycle time. The asynchronous performance of the CMCTL can be estimated by adding one bus cycle to the synchronous data transfer numbers in Table 2.

The CMCTL memory read access time is very important for systems that do not have a second-level cache. For example, a 90-ns CVAX bus cycle with a 5/3 CMCTL memory read access with a second-level cache results in CPU performance 3.0 times that of the MicroVAX II. Without the second-level cache, the CPU performance is

Table 2 CVAX CMCTL Read and Write Performance (in Numbers of Bus Cycles)

Memory Operation (4 Bytes of Data)	CVAX Bus Cycles		
	100 ns	90 ns	60 ns
Single read	4	5	6
Multiple read	4/2	5/3	6/3
CPU single write	2	2	2
DMA single write	3	3	3
Multiple write	3/2	3/2	3/2

reduced by 15 percent, or to 2.5 times the MicroVAX II. If the CMCTL memory read access was fixed at 6/3 without the second-level cache, the CPU performance would be reduced another 10 percent, or to 2.0 times the MicroVAX II, at a 90-ns CVAX bus cycle. Therefore, the ability to program the CMCTL memory read access time as an integral multiple of the CPU bus cycle is a very important feature that helps maximize the CPU performance.

CMCTL Functions

The CMCTL was designed to integrate both the control and data path functions required to control the data flow to and from memory.

Registers

The CMCTL contains two registers:

- A status register
- A control register

How each functions within the CMCTL and the system is described below.

The status register is loaded with important information when the CMCTL detects an error. The system error-handling software uses this information to log the error. The CMCTL has a memory error status register that captures the failed memory address along with the type of memory error (bus parity error or memory error) and error syndrome.

In ECC mode, the error syndrome is a 7-bit encoded number. For correctable errors, this number indicates which data bit was corrected. In parity mode, the error syndrome has no useful meaning.

The chip's control register serves several functions. First, the control register regulates a diagnostic test mode. Second, this register controls the PMI cycle tuning. Third, memory error detection and correction can be turned on or off to facilitate the testing of the CMCTL error-checking functions and memory module RAMs by memory diagnostic software. Finally, a refresh operation can be forced for high-speed refresh testing.

Data Path

In ECC error detection mode, the data path uses a modified Hamming code to detect double-bit errors and to detect and correct single-bit errors. The PMI interface has 39 signals; 32 are used for the memory data, and 7 are the memory check bits. In parity error detection mode, the data path

uses single-bit parity to detect memory errors. The data path transport delay for a memory read or write is one-half the cycle time of the CVAX bus. This performance measure includes module-level interconnect delay.

Memory Control

The PMI interface provides 20 signals. These signals comprise all the control strobes and memory address signals needed to control DRAMs. A fast memory access time is achieved by detecting a valid memory address and starting a memory access within 25 percent of a CVAX bus cycle time.

The CMCTL has an integral refresh counter for refreshing memory.

Summary

The CVAX CMCTL is the core control function of a complete memory subsystem. The chip provides the control for a flexible memory subsystem that functions at CVAX bus cycles from 60 to 100 ns.

Acknowledgments

The author wishes to acknowledge the technical contributions of F. Aires, M. Benoit, K. Chui, J. Clouser, N. Fitzgerald, J. Gerde, B. Griswold, N. Murthy, S. Nadkarni, J. Siegel, R. Strouble, K. Steward, and K. TenHuisen.

digital™

ISSN 0898-901X

Printed in USA EY-6742E-DP Copyright © August 1988 Digital Equipment Corporation

