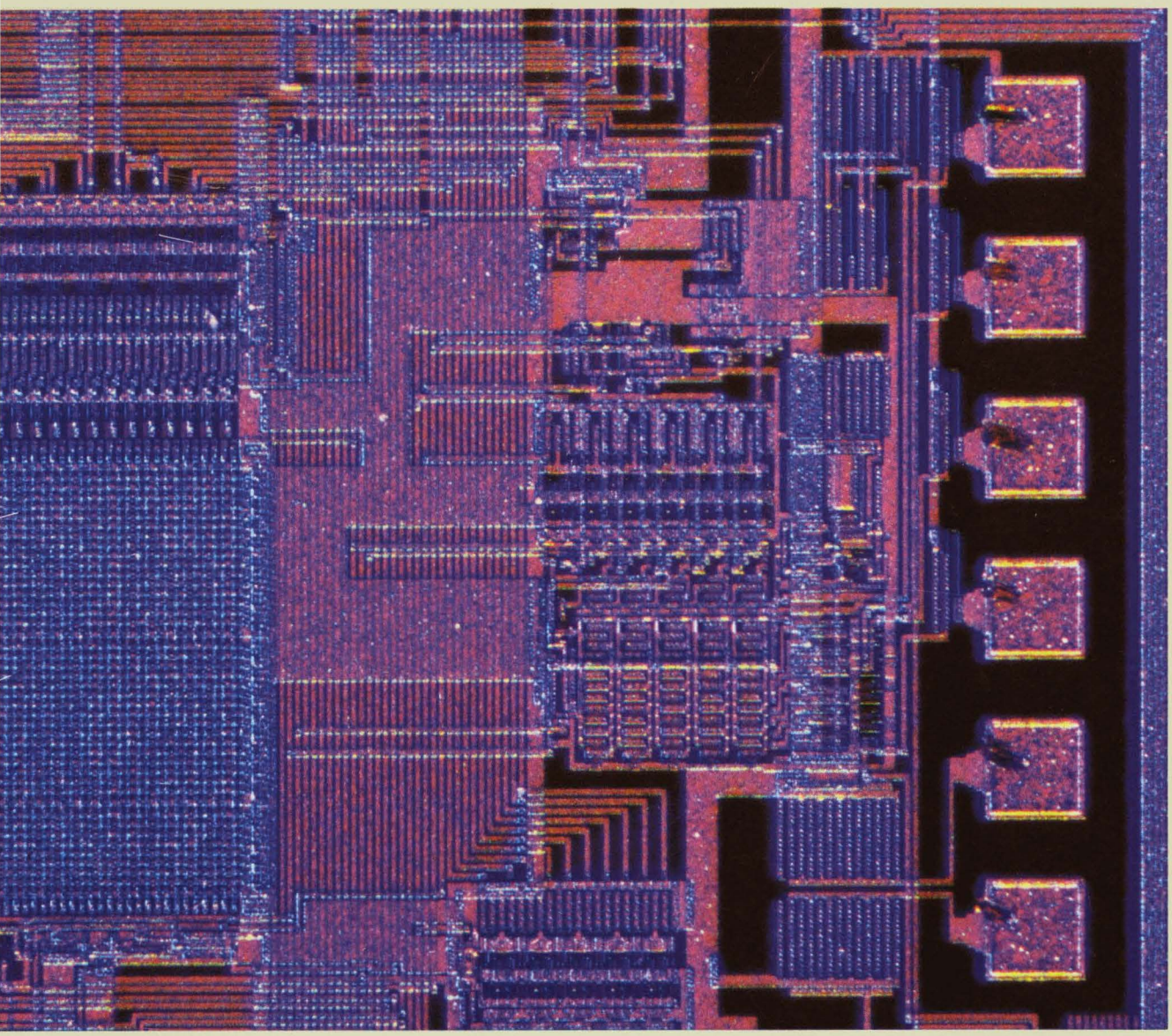
□ *MicroVAX II System*

Digital Technical Journal

of Digital Equipment Corporation



Number 2
March 1986

Editorial Staff
Editor – Richard W. Beane

Production Staff
Production Editor – M. Terri Autieri
Designer – Charlotte Bell
Typesetting Programmer – James K. Scarsdale

Advisory Board
Samuel H. Fuller, Chairman
Robert M. Glorioso
John W. McCredie
John F. Mucci
Mahendra R. Patel
Grant F. Saviers
William D. Strecker
Maurice V. Wilkes

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

Comments on the content of any paper are welcomed. Write to the editor at Mail Stop HL02-3/K11 at the published-by address.

Comments can also be sent on the ENET to RDVAX::BEANE or on the ARPANET to BEANE%RDVAX.DEC@DECWRL.

Copyright © 1986 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 the 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.

ISBN 932376-89-4

Documentation Number EY-3474E-DP

The following are trademarks of Digital Equipment Corporation: CompacTape, DEC, the Digital logo, MicroVAX, MicroVAX I, MicroVAX II, MicroVMS, PDP-7, PDP-11, Q-BUS, RSTS, TK50, ULTRIX, ULTRIX-32, UNIBUS, VAX, VAX-11/730, VAX-11/750, VAX-11/780, VAX 8600, VAX 8200, VAXELN, VAXstation, VMS, VT.

Apple II is a trademark of Apple Computer, Inc.

AT&T is a trademark of American Telephone & Telegraph Company.

IBM is a registered trademark of International Business Machines, Inc.

Mylar is a trademark of E. I. duPont deNemours & Company.

Tek is a registered trademark of Tektronix, Inc.

UNIX and System V are trademarks of AT&T Bell Laboratories.

Xerox is a registered trademark of Xerox Corporation.

68000 is a trademark of Motorola, Inc.

8086 and Intel are trademarks of Intel Corporation.

The manuscript for this book was created using generic coding and, via a translation program, was automatically typeset. Book production was done by Educational Services Media Communications Group in Bedford, MA.

Cover Design

Hardware, software, and peripheral devices for the MicroVAX II system are featured in this issue. Two VLSI devices, the 78032 CPU chip and the 78132 FPU chip, form the core of this system. Our cover shows the input programmable logic array for the FPU chip.

The cover was designed by Deborah Falck of the Graphic Design Department.

Contents

Editor's Introduction

Richard W. Beane
Editor

This issue of the journal is the second published by Digital's engineering organization. Our first issue (August 1985) featured papers about the technologies used in designing the VAX 8600 processor. The journal presents papers written by the technical contributors who design Digital's products. The information is directed at engineering faculty members, Digital's own engineers, and customers.

This issue features the MicroVAX II system, which implements the VAX architecture on a single CPU chip, the 78032. Another chip, the 78132, executes fast floating point operations; a single board holds both those chips, plus one megabyte of memory. New peripherals have been designed, and the VMS and ULTRIX software adapted to the MicroVAX II system. This collection of papers, by authors from different engineering groups, presents a wide spectrum of the MicroVAX II hardware and software.

The first paper, by Dan Dobberpuhl, Bob Supnik, and Rich Witek, is a description of the 78032 CPU chip, which implements a subset of the full VAX instruction set. The decisions about which instructions to microcode are discussed, along with hardware simplifications needed to fit functions on one chip. The chip's various operations are explained, with emphasis on parallel execution.

The CPU chip can use a coprocessor, the 78132 FPU chip, to perform fast floating point operations. The paper by Bill Bidermann, Amnon Fisher, Mike Leary, Bob Simcoe, and Bill Wheeler relates the 78132's architecture and algorithms. The protocol between the two chips is discussed and a description is given of the wiring and signal integrity issues and how they were addressed.

Both chips are mounted on a single board containing one megabyte of memory. Barry Maskas' paper explains how the CPU board had to be designed as a linked sequential machine with dual ports. The development process is interesting because the board and the chips were designed in parallel.

The paper on CAD tools, by Tony Hutchings, relates the large role they played in the chip and board designs. The various levels of CAD support, from behavioral modeling, through logic and circuit simulation, to wirelist generation is described.

The software graphics that turn the MicroVAX II system into a single-user workstation are reported in the paper by Rick Spitz, Peter George, and Steve Zalewski. The control of windowing software and virtual displays is discussed, as are the implementation details.

The RQDX3 disk controller provides fast data transfers between a CPU and disk storage devices. Nick Warchol and Stephen Shirron explain the top-down development process that lead to unique solutions to difficult problems. Their description of the final architecture shows how the original goals were met in the eventual design.

With a subset architecture, those instructions not in the set have to be executed another way. The paper by Kathy Morse and Larry Kenah describes the macrocode emulation of the VMS changes required to do that. The testing techniques are interesting since they were done without MicroVAX hardware.

The paper by Steve Boone and Guenter Schneider describes the TK50, a streaming cartridge tape drive providing fast data transfer. The authors discuss the unique cartridge, tape transport, and controller designs, highlighting the self-threading technique and the serpentine read/write process.

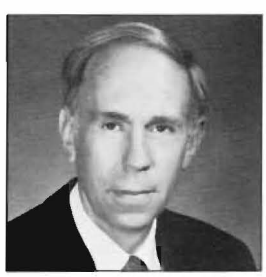
The final paper, by Ray Lanza, describes porting the ULTRIX-32 software to the MicroVAX processor. Ray explains the cross-development environment and the mapping techniques that allowed the heart of the ULTRIX software to fit on a small system.

Dick Beane

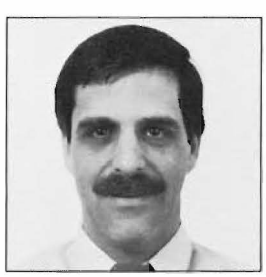
Biographies

William R. Bidermann Bill Bidermann is the engineering manager of the Advanced Development Memory Group. He consulted on the floating point chips for both the VAX 8200 and MicroVAX II processors. Before joining Digital in 1984, he was a consultant for Tenex and Rampower. Previously, he worked as a project manager at Hewlett Packard Laboratories in Palo Alto, California, and as a design engineer at Texas Instruments Central Research Labs. Bill received his S.B. and S.M. degrees in electrical engineering and computer science from M.I.T. in 1978.

Steven E. Boone Steve Boone graduated from Michigan State University (B.S.E.E., 1974) and the University of Michigan (M.S.E.C.E., 1975). He has also done advanced graduate work at Southern Methodist University. Before joining Digital in 1984, Steve worked as a principal hardware engineer for Sequoia Systems, and as a senior design engineer at Prime and Raytheon. For two years, he was an engineering supervisor working on the TK50 controller design. Steve is currently the technical engineering manager for TK Cartridge Tape Subsystem Engineering.



Daniel W. Dobberpuhl Dan Dobberpuhl is a senior consulting engineer and manager of the Processor Advanced Development Group. On the MicroVAX II project, he led the implementation of the 78032 CPU chip. Previously, he consulted on CMOS, ZMOS, and TIPI technology development, and worked on the T11 and F11 projects. Dan joined Digital in 1976 from General Electric Company. He received a B.S.E.E. degree from the University of Illinois in 1967. A member of IEEE, he holds four patents and is the coauthor of *The Design and Analysis of VLSI Circuits*.



Amnon Fisher Educated at Israel Institute of Technology (B.S.E.E., 1973) and City College of New York (M.S.E.E., 1975), Amnon Fisher worked as both a contributor and project leader on the 32016 CPU at National Semiconductor. Joining Digital in 1983, he was a project leader of the V11/SCORPIO floating point chip (VAX 8200 system), and a contributor to the MicroVAX II 78132 chip. Amnon is currently an engineering manager in the Semiconductor Engineering Group, working on the design and development of a four-chip set VAX implementation.

Peter C. George Earning his bachelors and masters degrees in computer science and engineering from M.I.T. in 1980, Peter George joined the VMS Development Group in that year. He first worked on VMS user interfaces, then on the workstation software as a principal engineer on the VAXstation project. Peter is currently a project leader, working on advanced workstation software projects. Peter is a member of ACM, and the national honor societies Tau Beta Pi, and Eta Kappa Nu.

Anthony F. Hutchings Tony Hutchings received his B.S. degree from the University of Newcastle On Tyne in 1965. At ICL in the U.K. for 16 years, he designed operating systems and was one of the VME-system architects on the 2900 series. He later became corporate manager of CAD. Tony joined Digital in 1982 as the project manager for the proprietary DECSIM software and then became manager of the VLSI CAD Group. Tony, a member of IEEE and the British Computer Society, is currently chairman of the CAD section of the ICCD.

Lawrence J. Kenah Larry Kenah, a consulting software engineer in the VMS Development Group, wrote the decimal/string emulator for the MicroVAX project. Since joining engineering in 1980, Larry has worked on the VMS nucleus in the areas of memory management, process scheduling, and image activation. He came to Digital in 1975 as an instructor and course developer in Educational Services. Larry received his B.S. degree (1968) from Boston College and his M.S. (1970) and Ph.D. (1977) degrees in high-energy physics from Northwestern University. He is coauthor of *VAX/VMS Internals and Data Structures.*

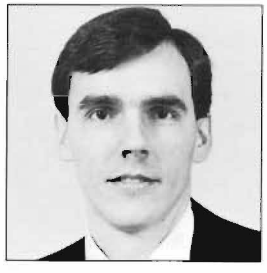
Raymond J. Lanza Ray Lanza is currently the project leader for the ULTRIX-32 system. After joining Digital in 1983, he ported the ULTRIX system to the MicroVAX I processor. As project leader, he ported the system to the MicroVAX II processor in 1984. Ray received his B.S.E.E./C.E. degree from the University of New Hampshire in 1980, then became the lead engineer in a UNIX group at AT&T. Later he was a senior software engineer at Wang Laboratories, Inc., researching windowing systems and UNIX distributed systems.

Burton M. Leary In 1980, Mike Leary joined Digital after receiving his B.S. degree in electrical engineering from the University of Massachusetts. In semiconductor engineering, he worked on chip designs and helped to develop the floating point chip for the MicroVAX II system. Mike did behavioral modeling, wrote microcode, and designed the main sequencer for that chip. He is now a senior engineer in the Advanced Development Memory Group, designing the internal cache for an advanced chip project.

Barry A. Maskas Barry Maskas is a principal engineer currently specifying and designing an integrated circuit, and fiber-optic boards for future systems. As a senior engineer on the MicroVAX II project, he was co-designer of the CPU board and the memory boards. Barry came to Digital in 1979 after receiving his B.S.E.E. degree from Pennsylvania State University. He also holds an associate's degree from the Community College of Allegheny County and did undergraduate work at LSU. Barry is a member of Eta Kappa Nu; he has a patent pending for a self-configurable memory subsystem.

Kathleen D. Morse As a consulting software engineer, Kathy Morse is responsible for VMS support on all low-end CPUs and peripherals. Earlier, she did the VMS support for both MicroVAX systems, the VAX 11/782 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. She has published in the Computer Measurement Group's 1985 Conference Proceedings, and *Datamation*.

Guenter E. Schneider Guenter Schneider joined the Mass Storage Group in 1970, when it had only about 25 people. He has worked on the designs for the RX05, RL01, RX02, TU58, RX50, and RD50/51 storage devices. As a consulting engineer, he helped to design the TK50 cartridge tape drive. Guenter received a Diplom Ingenieur from the Technische Hochschule Aachen in West Germany and his M.S.M.E. degree from M.I.T. in 1969. He holds two patents, with a third pending, and is a member of the engineering society Verein Deutscher Ingenieure.



Stephen F. Shirron Educated at Catholic University of America (B.S., 1980 and M.S., 1981), Stephen Shirron came to Digital after graduating Summa Cum Laude. As a senior software engineer, he developed an interpreter for VAX/Smalltalk-80 and designed the VAXstation 100 firmware. Currently a principal software engineer, Stephen designed and implemented the firmware for the RQDX3 disk controller. He is a member of Phi Beta Kappa and has written a chapter in *Smalltalk-80: Bits of History, Words of Advice*.

Robert J. Simcoe Bob Simcoe is a technical manager currently working on serial interconnect products. He was the technical manager for the floating point chips in both the MicroVAX II and VAX 8200 systems. Before joining Digital in 1982, Bob worked for the Department of Defense and General Electric Company. His duties involved MOS design, process development, and product design using custom ICs. Bob holds seven patents on IC circuitry and systems. He graduated from the University of Illinois (B.S.E.E., 1966).

Rick Spitz Rick Spitz manages VAX/VMS software development for CPUs and peripherals. As a consulting software engineer, he was a primary member of the architectural design team on the MicroVAX workstation project. Rick designed the VMS graphics hardware interface architecture and, for six years, has specialized in VAX/VMS hardware-software interfaces. He joined Digital in 1977 as a senior software specialist and received Digital's Software Excellence Award. Previously, Rick developed microprocessor software for Inco, Inc. He earned a B.S.E.E. degree from Clemson University in 1974 and his M.S.C.E. degree from the University of Lowell in 1983.

Robert M. Supnik Bob Supnik is a corporate consultant and group manager in semiconductor engineering. On the MicroVAX CPU chip project, he was project leader and lead microprogrammer. Bob was the project manager for the J11, a contributor to the F11, and supervised advanced development on the HSC50 and UDA50. Before joining Digital in 1977, he worked at Applied Data Research. Bob received his S.B. degrees (1967) in math and history from M.I.T. and his M.A. degree (1972) in history from Brandeis University. He received *Science Digest's* "100 Top Innovators of 1985" award.

Nicholas A. Warchol In 1977, Nick Warchol joined Digital after receiving his B.S.E.E. degree (cum laude) from the New Jersey Institute of Technology. Later he earned his M.S.E.E. degree from Worcester Polytechnic Institute in 1984. He is a member of Tau Beta Pi and Eta Kappa Nu. Nick has worked on the advanced development of charged-couple device memories, bubble memories, and laser video disks. In his present position as a principal engineer, he worked on the design of the RQDX3 disk controller.

William R. Wheeler After earning his B.S.E.E. degree in 1982 and his M.S.E.E. degree in 1983 from Cornell University, Bill Wheeler came to Digital as a junior engineer. On the MicroVAX II project, he designed the exponent datapath and control for the 78132 floating point chip. Later he designed the exponent section of the floating point chip in the VAX 8200 system. Bill is currently working on the instruction box and bus interface unit for a new microprocessor chip.

Richard T. Witek Rich Witek is a consulting engineer working on the architecture and implementation of new microprocessors. He helped to develop and debug the MicroVAX 78032 CPU chip. Rich also worked on implementing DECnet/E and on the DECnet Architecture Review Group during Phases 2 and 3. He also worked in the VLSI CAD group. Before joining Digital in 1977, Rich was a senior technical associate at AT&T Bell Laboratories and an engineering assistant at Argonne National Laboratory. He received his B.A. degree in computer science from Aurora College, and is a member of ACM and IEEE.

Stephen H. Zalewski Steve Zalewski is a senior software engineer working on the graphics execution routines for the VAXstation II/GPX system. He joined Digital in 1981 after receiving his B.S. degree in computer engineering from Worcester Polytechnic Institute. Steve developed the graphics device driver for the VAXstation I and II systems. His earlier work involved writing RMS file-sharing internals and implementing RMS file sharing and global buffers for VAXcluster software.

Foreword

Jeffrey C. Kalb
Vice President
and Group Manager
Large Scale Integration

The roots of the MicroVAX program go back to the summer of 1981. To understand why this program was initiated and the thinking behind it, one has to look at the events of that time. Many developments were taking place, suggesting that a whole new class of systems capabilities could emerge before long.

The VAX-11/780 system was in its heyday. It was recognized as the standard against which all other computers were compared and benchmarked. And true to fashion, everyone seemed to find some way to benchmark his machine in some particular niche against the 11/780's capabilities. That was particularly true of the upcoming generation of microprocessors and microprocessor-based systems. The universities were busily benchmarking Intel Corporation's latest generations of 8086s, 80186s, and the early 80286s on specific jobs. The same was true of the 68000-based system. Many companies were starting to come to market with engineering workstations and similar products based on these microprocessor chips. In fact if one believed the trade press, the VAX-11/780 system had actually been eclipsed in performance and capabilities by these "upstarts."

Needless to say, these events caused some degree of consternation and soul-searching within Digital Equipment Corporation. Moreover, another factor was becoming painfully obvious: the emergence of the independent software vendors. Hoards of small companies were springing up everywhere to generate software for various personal computers that either had already been introduced to the marketplace, like the Apple II, or shortly would be, like the IBM PC. These small vendors wanted to write software for the systems that had the highest market volume. Their reasoning was clear. To sell as many of their software packages as possible required implementing their ideas on the highest volume hardware. It was also clear that the highest volume hardware was going to be microprocessor based and quite inexpensive.

Meanwhile, within Digital, the Semiconductor Engineering Group (SEG) was busy developing a multichip implementation of the VAX architecture. Built with a midrange, multiuser, high-performance system in mind, this chip set and its attendant system implementations were aimed at the marketplace for systems above $50 thousand. CAD tools were being developed and manufacturing processes developed and refined. The module and system concepts were then in the definition stage.

Discussions began at this time, centered around what was later known as the MicroVAX system. There was a perceived need to counter the rising tide of encroachment on our systems business by microprocessors. We wanted to create systems with volumes high enough to warrant the attention of the independent software vendors. In general, we wanted to establish the VAX architecture as one of the preferred architectures at all potential price levels in the entire industry.

These discussions and strategic thinking converged after receiving an unsolicited proposal from a semiconductor manufacturer. This firm had approached us during that summer, wanting to implement the VAX architecture in one or two high-performance chips. This set of chips could be used in our systems and sold as standalone products. The firm wanted to use the VAX/VMS architecture (and primarily the software associated with it) to get a jump in the marketplace by establishing a high-volume architectural standard at the 32-bit level. We were concerned from the beginning that the capabilities and resources of this smaller firm would not be sufficient to execute such a formidable program. But the notion that building a single-chip VAX implementation and using it to counter-attack the emerging microprocessor-based systems had struck a responsive chord. Until that time, our thinking had been in terms of our traditional price/performance learning curves. Our strategies did not include extraordinarily low-priced VAX systems.

As indicated above, the Semiconductor Engineering Group in Hudson, Massachusetts, was already heavily committed to the multichip VAX system. A number of other major chip projects were in development as well. Therefore, we searched for a larger semiconductor vendor who could bring additional design and manufacturing resources to bear on this concept. Such a vendor could also make available additional distribution channels for sales of high-volume chips to the general marketplace. This line of thinking was pursued with various vendors throughout the fall and winter of 1981, until April 1982.

Interestingly enough, there was less than wholehearted enthusiasm on the part of the various vendors who were approached. Each of them had already decided on an approach to the problem and were unwilling to make the development of the MicroVAX chip a priority item. That commitment was an extremely important issue to us. Experience had shown that complex projects of this nature always exceeded the schedules and the budgets anticipated when they received second-class attention within the merchant semiconductor industry. Thus one criteria for working with a vendor was that he commit to the MicroVAX architecture as a primary market thrust. No one was willing to do that.

At the same time, other issues had to be worked. It was clear that the full VAX architecture as implemented in the multichip set could not easily be put on a single chip. That would have taken over 1 million transistors, a capability that would not be available until the end of the decade. Therefore, early in the project, we recognized that there was a need to subset the architecture to make it implementable on a single chip. By December 1981, the idea of developing a single-chip VAX implementation was beginning to get some positive re-enforcement within Digital. As a result, in that month, Gordon Bell, then vice-president of Engineering, chartered a subcommittee to investigate what should be included in a MicroVAX architecture.

The key people involved were Roy Moffa, who had been leading the strategic thinking about a single-chip VAX system; Bob Supnik, representing semiconductor technology; Dick

Hustvedt and Dave Cutler, representing software technology; and Bill Strecker, representing VAX architecture technology. After a few intensive meetings, they proposed a subset of the VAX architecture in January 1982. Bob Supnik and the semiconductor technologists thought that this subset could be implemented in a single chip. This new architecture would be modified slightly later in the year, but it is essentially the architecture that exists today. The only significant modification was in the memory management capability, and in some sense, this change actually simplified the development of the chip.

In parallel with these other activities, Bob Supnik and other members of SEG had been studying ways to get the chip developed internally. They were hoping to leverage the existing investments in process technology, chip modeling, CAD tools, and the various other elements that were necessary. Furthermore, and highly significant to the whole program, they developed ways of re-using some of the investments being made in the multichip VAX implementation and other programs already in progress. As a result the floating point chip being developed for a PDP-11 microprocessor was used as the building block for the MicroVAX implementation. Not only that but the chip was also retrofitted back into the existing multichip set to minimize the workload. Moreover, the datapath was lifted from the instruction/execution unit of the multichip set to form the backbone of the MicroVAX CPU. Tools and techniques were borrowed whenever it was possible.

In this sense the MicroVAX program was unique. There were almost nine months of strategy discussion and evaluations of various ways of implementing and executing before any real design actually started. While many of the proposed business strategies were never adopted, they at least received a hearing. In any case the die was cast.

The real implementation of the MicroVAX chip did not get started until June 1982, the official start date being July 6, 1982. (Some work had been done prior to that for recruiting and staffing.) It was soon evident that there were some key elements that had to be addressed. The first was CAD tools. There was no question that this device had to be simulated extensively at all levels of implementation. There was no other way to get the quality of design and performance levels being planned. At the time the program started, these tools were mostly experimental. Some techniques had been tested, but the reality was that CAD tools "broke" on numerous occasions during the development of the system. Crisis-oriented SWAT teams had to be put in place to bridge over or break through barriers that threatened to bring the entire program to a halt.

There was another equally important element. The entire program was an extremely complicated one, with many elements on parallel paths. Process technology had to be developed, CAD tools developed and refined, chip designs done, systems implementations executed, and test techniques and equipment developed. Each of those elements was intimately entwined with the others. Therefore the possibility clearly existed that, upon reaching the end of the design, we would be faced with debugging a new process technology, a new manufacturing line, new testers, a new chip design, new packages, and a new system, all simultaneously. A real possibility existed that we couldn't separate the variables in a sufficiently clear and timely manner to allow the chip debugging and system evaluation to take place. This phase could last for months or perhaps even years, something that has happened before on many such programs in the merchant industry.

To avoid that, we segmented the major risks in the program and put plans in place to minimize as many of those as possible in parallel before the new chip arrived. For instance, rather than debugging an entirely new manufacturing line while trying to build this new chip, we combined the existing two wafer fabrication lines into one. The smaller line was then retrofitted to provide a pilot line capability. That gave us a trained staff, a debugged facility, and all the other elements necessary to mini-

mize the interaction of the process and facility. Additionally, a test vehicle was designed so that manufacturing could run wafers, debug process steps, and improve the basic yields of the process well before the new chip arrived. In the test area, test programs were implemented on older, proven testers on which the engineers had experience. That worked even though we knew that, for the eventual production, an entirely new generation of testers would be necessary to precisely test such a complicated device at its full speed.

Similarly, other areas, such as packaging, CAD tool development, and parts of the system evaluation, were examined and improved in parallel long before they had to work together. A major program was put in place to uncouple risks and to hire and train the workforce well in advance of the completion of the MicroVAX chip design. This effort was quite expensive; some people thought that much of the money was being thrown out with the materials that were made experimentally. But the end result was one of the smoothest debugs and introductions into chip manufacturing that I have ever witnessed for a complex device. While there were problems and although things didn't always work right, there were almost always independent ways of separating the variables in the problem. In that way it could be properly analyzed and corrections put in place. This example should serve us well with complex development programs in the future.

One other thing done to enhance the debug and ensure the quality at the system level was to co-locate the CPU module designers with the chip designers. In that way their interaction was enhanced and the rate of problem resolution greatly accelerated. The module team itself was exceptionally small for such a major program, consisting of only three primary engineering people. But this unique program environment featured a high degree of simulation, close proximity of the engineers (the MicroVAX chip team had only 20 people), and heavy reliance on thorough evaluation at every step.

The end result was very, very few bugs in either the chip or the system. In fact there were fewer than 20 bugs that had to be corrected before the integrated chip and system were able to boot the operating system. It should be noted that this quality has continued to manifest itself in the rapid manufacturing ramp-up and the quality of the systems that have been generated. There were more engineering changes to the parts and the system to enhance our margin and ease of manufacture than there were to make the system functional in the first place. That is evidence of a fundamentally different approach to building systems.

As noted above, the MicroVAX program is quite unique, from its initial conception to the continuing efforts to enhance quality and productivity. From the initial conception of the strategy, through the organization of the people and problems, to the ongoing engineering activity around quality and ease of manufacture, this program has provided a new paradigm for program execution and management. Our hope is that, with this knowledge, people can emulate the success of this program while eliminating the errors. In so doing, Digital can greatly enhance its ability to build and manufacture high-quality systems in increasingly shorter periods of time.

Daniel W. Dobberpuhl
Robert M. Supnik
Richard T. Witek

The MicroVAX 78032 Chip, A 32-Bit Microprocessor

The MicroVAX 78032 implements the VAX architecture on one chip. To do that, the instruction set was repartitioned to reduce the number of transistors. The instructions used most frequently are in microcode; others, notably floating point, are emulated in macrocode. Hardware was simplified by having a small address translation cache and no memory cache; however, full VAX memory management is supported. A fast 200-nanosecond microcycle allows instructions to execute in parallel. The CPU chip is made using a 3-micron, double-metal NMOS process. The control store ROM has X-shaped cells, which help to reduce its size.

The MicroVAX 78032 chip is the latest extension of the VAX architecture and the first in the form of a single-chip microprocessor. As the CPU of the MicroVAX II computer system, the 78032 performs nearly as fast as the VAX-11/780 superminicomputer, but in a microcomputer package.

Origins and Goals

Digital began the MicroVAX CPU chip project in late 1981 in anticipation of increasing competitive pressures from industry-standard microprocessors. The original intent of the program was to license a semiconductor vendor to design and manufacture a MicroVAX single-chip microprocessor. However, the leading semiconductor companies were unable to meet the high-performance requirements and tight schedules that the project required. In May 1982, an internal development project was chartered to design the MicroVAX CPU chip.

From a designer's viewpoint, the development of this CPU was a challenging exercise in shrinking the VAX computer architecture without changing its function. There were five major goals that governed the design.

1. The kernel architecture was to be implemented on a single chip. Other chips or hardware could be used to improve performance or to provide additional functionality, but the basic VAX functions had to be incorporated in the base CPU design.
2. The chip had to be compatible with all VAX application programs. It had to execute any application program, whatever its size or complexity, written for any computer in the VAX family. And it had to execute without alterations to the program code. That meant that the chip had to run the MicroVMS and ULTRIX-32m (Digital's enhanced UNIX software) operating systems, and the VAXELN real-time kernel.
3. The chip had to perform at or near the speed of the VAX-11/780 processor. This goal implied that the chip had to have a highly parallel internal implementation, a high-performance external interface, and a fast microcycle. Accordingly, the internal microcycle of the chip was set at the same 200 nanoseconds (ns) as the 11/780's microcycle.
4. The price of the chip had to be competitive with commercial 32-bit microprocessors of comparable complexity. This required a relatively conservative die size and an inexpensive package. It also required the implemen-

tation of an external interface that was compatible with standard VLSI peripheral chips and demanded minimal support from the hardware on the CPU board.

5. The chip had to be designed and built quickly. To meet or beat competitive products, the chip had to be in production less than 2 ½ years after the start of development.

With these goals guiding the chip design team, the major problem was quickly identified: to reduce the number of transistors. That, in turn, required repartitioning the VAX instruction set and simplifying hardware functions wherever possible.

Reducing the Number of Transistors

The principal problem in designing the 78032 was how to implement the complexity of the VAX architecture on a single chip. There are 304 instructions in the full instruction set, with 14 data types and 21 addressing modes. Instructions vary in length from 1 byte to 54 bytes.[1] Demand-paged virtual memory support is required to guarantee compatibility with the operating system software. To accommodate this complexity in a full-scale VLSI VAX implementation requires about 1.25 million transistor sites.[2] However, the semiconductor technologies available at the time of design could support only about one-tenth that number in a single-chip microprocessor.[3]

The architectural functions in all VAX systems are partitioned among hardware, microcode, and the operating system. All previous VAX implementations have similar boundaries between these three. The hardware provides the registers and memory, the microcode provides the instruction set, and the operating system provides the program services. A large control store–a minimum of 400 kilobits (Kb)–is required to contain the instruction microcode. The console function is handled in either microcode or a support processor. Moreover, the control logic needed to support memory management and the variable instruction format is quite complex.[4]

Two different approaches were taken to reduce the transistor count in the microprocessor chip. First, the VAX instruction set was repartitioned to cut the size of the control store to 62Kb. Second, the amount of on-chip hardware was reduced by simplifying some functions, placing others elsewhere, or omitting some altogether.

Repartitioning the Instruction Set

As the first repartitioning step, the design team assumed that all VAX instructions had to be implemented in order to execute all VAX application software. However, there are several classes of instructions that involve a good deal of microcode and yet are infrequently executed. For example, a typical timesharing workload is handled by base instructions, scientifically oriented instructions, and commercially oriented instructions. Analyses of more than 70 million executed instructions showed that the commercially oriented ones represented less than 0.2 percent of the total executed.[5,6] Studies of scientific and engineering workloads showed even lower percentages. Even in commercial applications, the commercially oriented instructions represented less than 4 percent of the total executed, the majority being base instructions. Therefore, emulating the commercially oriented instructions in the operating system rather than using microcode would significantly reduce the size of the control store, but would have little effect on overall performance because these instructions were seldom executed.

On the other hand, floating point instructions require a good deal of microcode and are executed more frequently. Even with microcode, instruction execution is relatively slow unless a separate floating point accelerator (FPA) is used. Therefore, although existing VAX implementations offered both microcoded (warm) and hardware (hot) floating point, the design team decided not to implement these instructions in microcode. Instead, floating point instructions would be executed in an optional floating point chip, or by emulation using macrocode.

In total, 175 of the 304 VAX instructions and 6 of the 14 data types are implemented in on-chip microcode. Those include integer and logical instructions, variable-bit field, control, queue, procedure calls, character string moves, and operating system support. This microcoded subset comprises over 98 percent of the instructions that are used to execute a typical program. However, the required microcode

occupies only one-fifth the control store space of a full VAX implementation. Seventy floating point instructions and three data types (F, D, and G floating) are implemented in the floating point chip, when it is present. If that chip is absent, the instructions are emulated in macrocode. The remaining 59 instructions and 5 data types are always emulated in macrocode. Those are mainly decimal string, character string, and H floating point operations. The CPU chip provides some microcode support for the emulated instructions. Table 1 summarizes the instruction set architecture of the 78032 chip.

The decision to emulate instructions in macrocode has an effect on speed because emulated instructions take three to ten times longer to execute than microcoded instructions. However, the instructions in this group of 59 are normally used so infrequently that the execution speed of a typical program is reduced by no more than four percent. Table 2 illustrates the division of instructions between the CPU chip, the FPU chip, and the macrocode. All in all, the fivefold reduction in the size of the control store halved what would have been the active area of the chip.

Simplifying the Hardware Functions

The principal hardware simplifications in the 78032 are the reduced size of the address translation cache (translation buffer), and the elimination of a memory cache in favor of tightly coupled local memory.

As mentioned earlier, demand-paged virtual memory management was required for compatibility with the VAX architecture. Consequently, the design team decided that the 78032 would

Table 1 Instruction Set Architecture

Implemented in CPU Chip		Implemented in Floating Point Chip		Implemented in Macrocode	
Instructions:					
Integer and Logical	89	F floating	24	H floating	28
Address	8	D floating	23	Octaword	4
Variable Bit Field	7	G floating	23	Character String	9
Control	39			Decimal String	16
Procedure Call	3			Edit	1
Miscellaneous	10			CRC	1
Queue	6				
Operating System Support	11				
Character Move	2				
Total	175		70		59
Data Types:					
Byte Integer		F floating		H floating	
Word Integer		D floating		Octaword	
Longword Integer		G floating		Leading Separate Numeric String	
Quadword Integer				Trailing Numeric String	
Variable Bit Field				Packed Decimal	
Variable Character String					

Table 2 Division of Instructions

	Instructions Implemented in CPU Chip	Instructions Implemented in Floating Point Chip	Instructions Implemented in Macrocode
Percent by Instruction Count	57.6%	23.0%	19.4%
Percent by Microword Count	20.0%	20.0%	60.0%
Percent by Typical Execution Frequency	98.1%	1.7%	0.2%

be the first single-chip CPU with full demand-paged virtual memory support right on the chip. At first the design team proposed to use a simplified version of VAX memory management. During the course of the design, however, the software engineers reported that not providing full memory management was quite expensive in terms of the use of physical memory. Therefore, the design team implemented full VAX double-mapped compatibility in the chip. As the design progressed, it became evident that the incremental cost of providing this capability was much lower than originally anticipated.

All existing VAX processors implement memory management with a large address translation cache (at least 128 entries), with system and process addresses in separate halves. A translation cache must have a high hit rate to be effective. Since most caches are direct mapped, many entries are required to achieve a high cache rate.[7,8] Implementing a comparable number of translation cache entries in the 78032 was out of the question, due to die size constraints. However, the VLSI technology in the 78032 is very amenable to using a fully associative translation cache with least-recently-used (LRU) replacement.

Such a cache needs many fewer entries to achieve the same hit rate as the direct-mapped version. In addition, the tight coupling to local memory, as explained in the next paragraph, made it possible to reduce drastically the amount of time required to process a translation cache miss. Thus the translation cache in the chip has only eight entries, but the cache is fully associative, uses true LRU replacement, and is supported by highly optimized microcode for fast processing of misses. Moreover, simulation studies showed that the best use of the eight entries was with a homogeneous structure. Therefore, the system and process addresses are cached together.

The team also decided to forgo the use of an external memory cache, which required a complex external interface. Use of an internal memory cache had already been ruled out due to die size constraints. Accordingly, the speed of memory access is 400 ns, or two microcycles, which is the speed of local memory. Thus the chip encounters no wait states, and its average time to access memory is approximately the same as the 11/780's. In a typical program, there is little difference between the integer instruction performance of the two CPUs.

Additional simplifications included the elimination of warm (microcoded) floating point in favor of a floating point accelerator, elimination of writable control store capability, and elimination of on-chip console support.

Design Narrative

The starting point for the chip design was the instruction execution chip of a multichip VLSI VAX processor already in design. This chip would provide a general floorplan and a base microarchitecture, and might even provide complete design sections that could be used for the MicroVAX 78032. As the project progressed, the designs of the VLSI VAX processor and the MicroVAX 78032 tended to diverge under the pressure of differing constraints: chip set and system functionality for the former; die size, power, and time to market for the latter. Ultimately, only part of the main datapath was shared between the two; the rest of the MicroVAX 78032 design and its microcode were unique.

The MicroVAX 78032 project took 20 months from start to first-pass mask generation: 6 months for specification and general design, and 14 months for physical implementation. Eighteen people worked on the design team.

Project Design Tools

The design team was aided by a hierarchical CAD tool suite that ran on a VAX system. The use of these tools was one of the primary reasons that the project was completed on schedule. The principal components of this tool suite are as follows:

1. A proprietary chip-database manager and tool interface called the CHAS system
2. A schematic capture program, QUICKDRAW, that uses simple terminals
3. A proprietary hierarchical simulator called the DECSIM system, used for behavioral simulation
4. A switch-level MOS logic simulator, RSIM, used for unit-delay logic simulation
5. A modified version of the standard SPICE circuit simulator that incorporates new analytical, rather than empirical, MOS transistor models
6. Design-rule checking programs, DRC and DRACULA II
7. An interconnect verification program called the IV system, which performs both layout extraction and wiring verification[9]
8. A cross-reference program, XREF, that analyzes coupling, bootstrap ratios, dynamic node stability, and other circuit problems

The chip layout was done on Calma GDS II systems. Three dedicated VAX-11/780 systems and five Calma stations were used throughout the project. The back-end verification of circuits and the layout required as many as eight VAX systems.

Final Chip Design

The final product of this design process is a microprocessor that contains 125,000 transistor sites in a 3-micron, double-metal NMOS chip that measures 8.7 by 8.6 mm. It requires only 5 Vdc and a maximum of 3 watts of power; it is packaged in a 68-pin, surface-mounted leaded chip carrier. The chip operates at 20 MHz and has full 32-bit internal and external datapaths. The 78032 is mounted on a single-board, quad-sized (8.5 by 10.5 in.) CPU module having a Q22 I/O bus and 1 megabyte (MB) of local memory. An optional FPA, the MicroVAX 78132 chip, can also be mounted on the CPU board.

The measured speeds of integer and floating point operations of the 78032 represent a breakthrough in 32-bit microprocessors. System evaluations of MicroVAX 78032 modules indicate that their performance in processing integers is approximately equal to that of the VAX-11/780 system. With the floating point chip, the performance is between those of the VAX-11/750 and VAX-11/780 systems with FPAs.

The remainder of this paper explains the functional organization of the chip and its physical implementation in silicon.

Functional Organization

The diagram in Figure 1 and the photomicrograph in Figure 2 outline the various subsections, or functional boxes, of the MicroVAX 78032 chip. They are organized into three sections. At the left of Figure 2 are the datapaths for decoding and executing instructions and for memory management. At the center is the control logic for internal operations and the protocol signal logic for external operations. At the right is the sequencing logic for both internal and external operations.

The left section in the photomicrograph (Figure 2), comprising the datapaths, consists of the I Box, the E Box, and the M Box.

- The I Box prefetches and decodes instructions. Its main function is to parse the current macroinstruction in the instruction stream and work in conjunction with the microsequencer to generate the microaddress for the next microinstruction. This microaddress is a function of the current macroinstruction. A prefetcher, which works in parallel with other chip operations, accesses and stores instruction data in an eight-byte prefetch queue. The prefetcher acts autonomously by attempting to keep that queue full at all times, using any free I/O-bus cycles to access the instruction

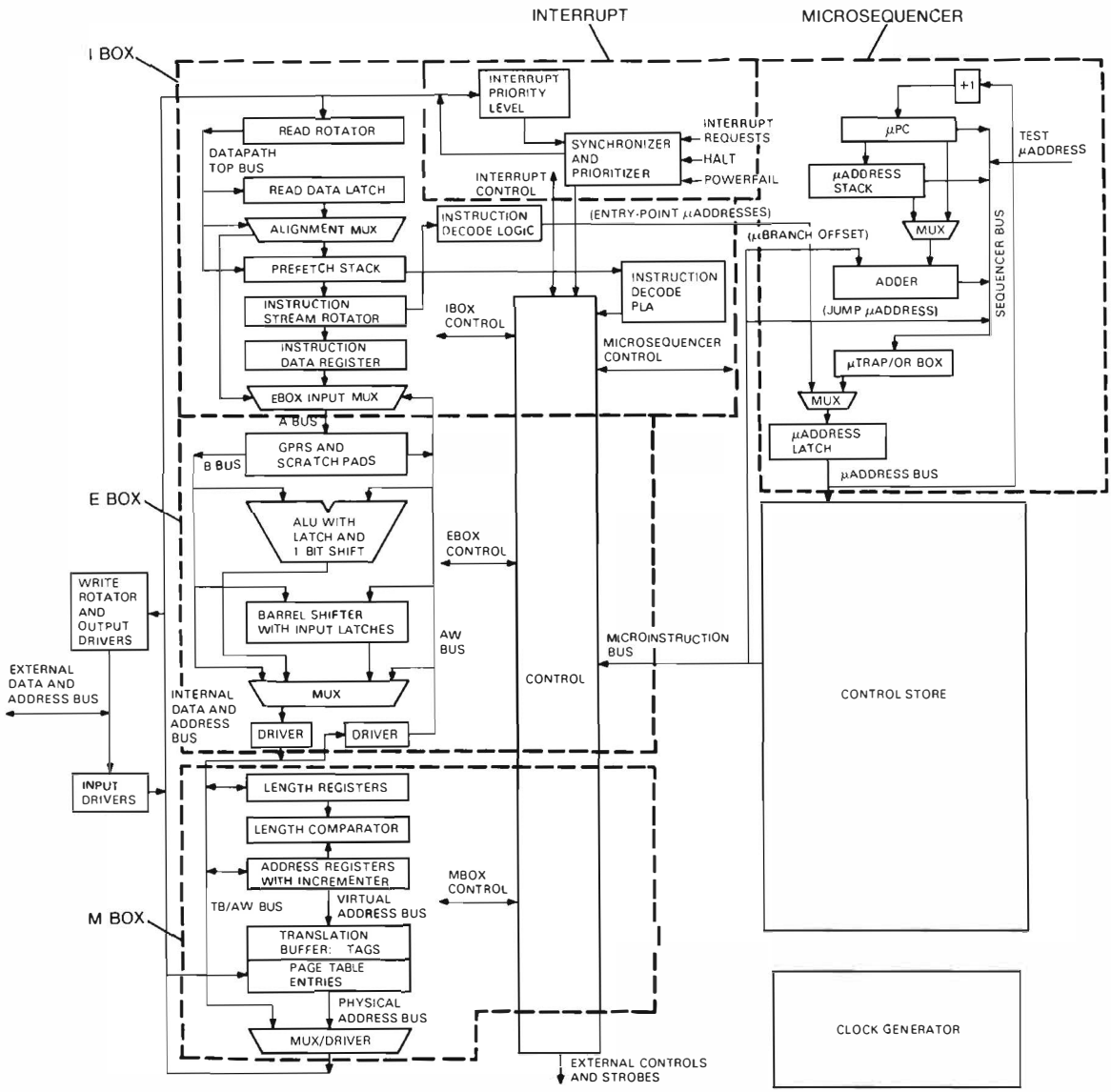


Figure 1 Block Diagram of the CPU Chip

stream. Even if the queue is full, the prefetcher will start to read data if the queue will be at least half-empty after the current microcycle.

The I Box also decodes instructions and variable-length operand specifiers in parallel with other chip operations. That avoids requiring explicit decode cycles to execute successive macroinstructions. Due to the constraints on the size of the control store, most of the address-specific microcode had to be shared among all instructions. The instruction-decode PLA (IPLA) generates 19 bits of opcode-specific data for controlling other chip operations related to a given instruction. That allows many microcode sequences to be table driven and shared.

- The E Box is the instruction execution unit and contains the main datapath of the chip. This box holds 16 VAX-specified general purpose registers (GPRs), 20 microcode registers, a 32-bit arithmetic logic unit (ALU), and a 32-bit barrel shifter. The E Box also maintains condition codes for the process

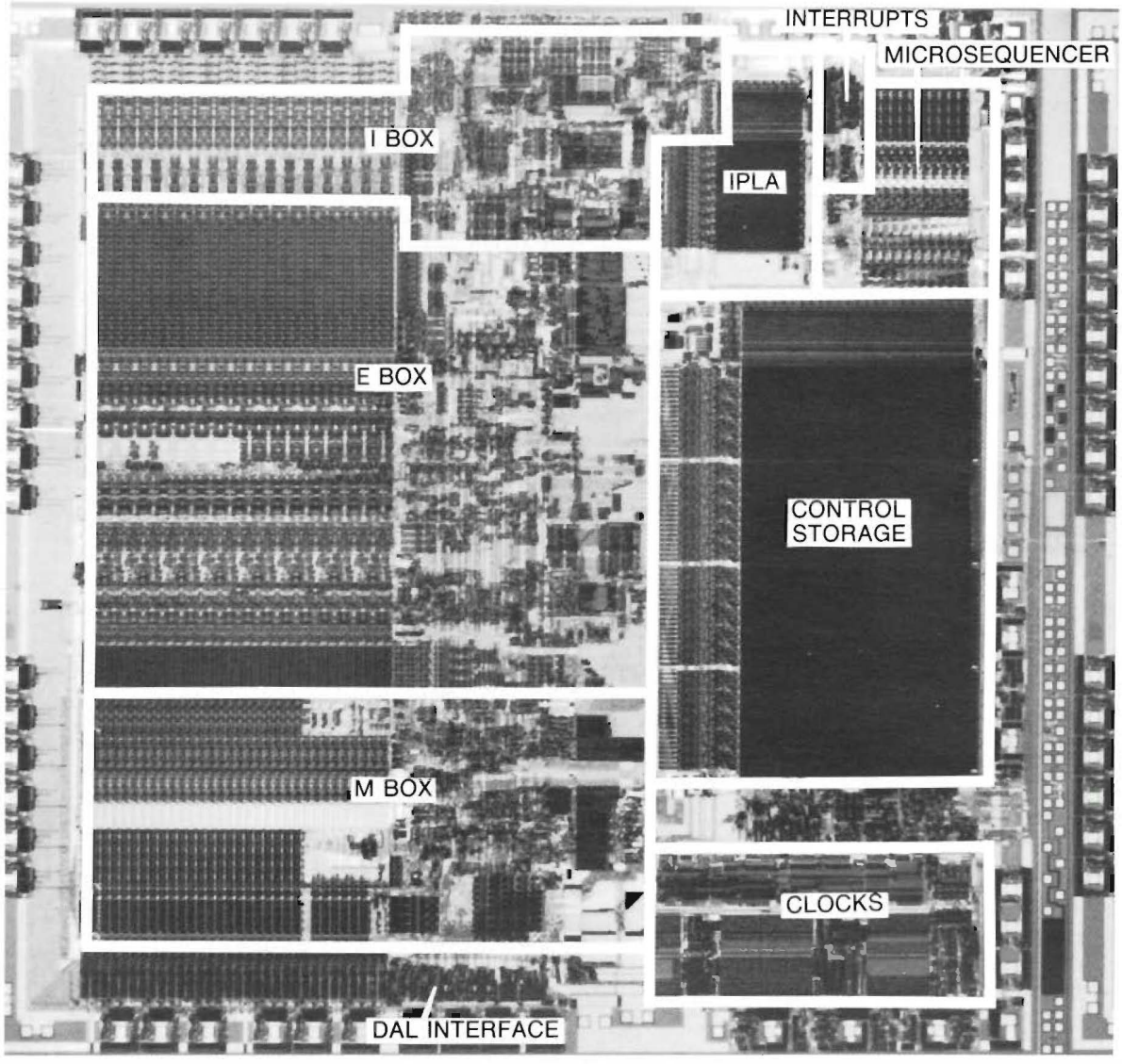


Figure 2 Photomicrograph of the CPU Chip

status longword (PSL) and determines VAX branch conditions at the macrocode level. In a 200-ns cycle, the E Box can read two registers, perform an ALU operation or shift, and write the result into a register. Since reading and writing to registers are performed sequentially, the ALU result bus is multiplexed with an input bus, thus saving vertical interconnect. The ALU employs a 4-bit lookahead carry scheme, with ripple carries across the nibbles. The carry chain uses dual-rail logic for maximum speed. The barrel shifter is a pass-transistor network, which is very compact and fast enough for this task.

- The M Box serves as the memory management unit and translates virtual addresses to physical addresses. The address translation cache, which is fully associative, stores the most recently referenced address translations. The M Box maintains three virtual address registers, one for instruction data and two for program data. This unit also detects cross-page accesses and includes a separate comparator for length checking. A dedicated adder generates the next virtual address for sequential data and instruction addresses. The time to perform an address translation is less than 25 ns when the virtual address is in

the translation cache. This short translation time allows memory management to be transparent to the external chip timing.

The center section of the photomicrograph is composed mostly of random control logic. That logic translates the highly vertical (39-bit) microcode into the many discrete control signals required to operate the datapath.

The right section of the photomicrograph, comprising the sequencing and clocking logic, consists of the interrupt logic, the control store, the DAL interface, and the clock generator.

- The interrupt logic accepts, synchronizes, and prioritizes external interrupt requests, compares them with the current interrupt priority level (IPL), and determines if the request will be serviced. The interrupt requests are checked at the beginning of each microcycle and the interrupt update is forwarded to the I Box. That all happens through the central control logic before the next microcycle begins.

  External interrupt processing has been implemented on-chip in the 78032 to avoid the complexity that results from having the interrupt priorities arbitrated outside the chip. Since these priorities are an integral part of the processor state, an off-chip design would involve broadcasting the interrupt priority level each time it changed. Moreover, off-chip interrupt processing would also require additional hardware on the CPU board.

- The microsequencer accepts inputs from various points on the chip and generates the next microaddress to access the control store. The microsequencer logic performs such operations as microsubroutine calls and returns, microcode traps, n-way (or case) branches, and signed offset conditional branches. Implemented in the microsequencer is an eight-level microprogram stack.

- The control store is a 39-bit ROM with 1600 entries. It receives microaddresses and status signals and generates the next set of microinstructions. The control store transfers those microinstructions to the control section in the center area. That section, in turn, generates control signals for the three principal functions in the main datapath: the I Box, the E Box, and the M Box. The access time of the control store is less than 100 ns.

- The DAL interface handles all control signals and transfers data and addresses between the chip and local memory, peripherals, and other devices outside the chip. The DAL interface transparently processes variable-length operands and aligns data references that cross natural 32-bit memory boundaries. It also causes the microprocessor to stall during I/O references, so that additional microcode is not needed to test for I/O completion. The DAL interface controls transactions involving the CPU chip, the FPU chip, and external devices. It also arbitrates direct memory access (DMA) requests.

- The clock generator receives an external 40-MHz clock reference and produces the eight 25-ns clock phases that time functions on the chip. The control logic of the chip makes extensive use of bootstrapped drivers. For that reason, certain clock phases have to drive very high capacitances, as much as 250 picofarads. To assist in that task, a special driver circuit with current-limiting resistors is used to provide fast edges without using excessive power or silicon area. These resistors control the overlap current drawn during bootstrapping and provide a voltage drop during the overlap.

External Interface

A principal goal in designing the chip's external interface (Figure 3) was to demand as few support functions as possible from the CPU board. The 78032 chip provides seven hardware interrupt inputs. Four of these inputs (IRQ<3:0> L) correspond to standard VAX I/O interrupts and result in vectored interrupt transactions. Three others (INTTIM L, PWRFL L, HALT L) have preassigned interpretations and the corresponding vectors are generated inside the chip. The 78032 takes in a double-frequency clock input from a standard oscillator. The chip produces a normal-frequency clock output, which can be used to drive or synchronize external logic. The functions between the chip and the Q-bus can be implemented in off-the-shelf discrete logic.

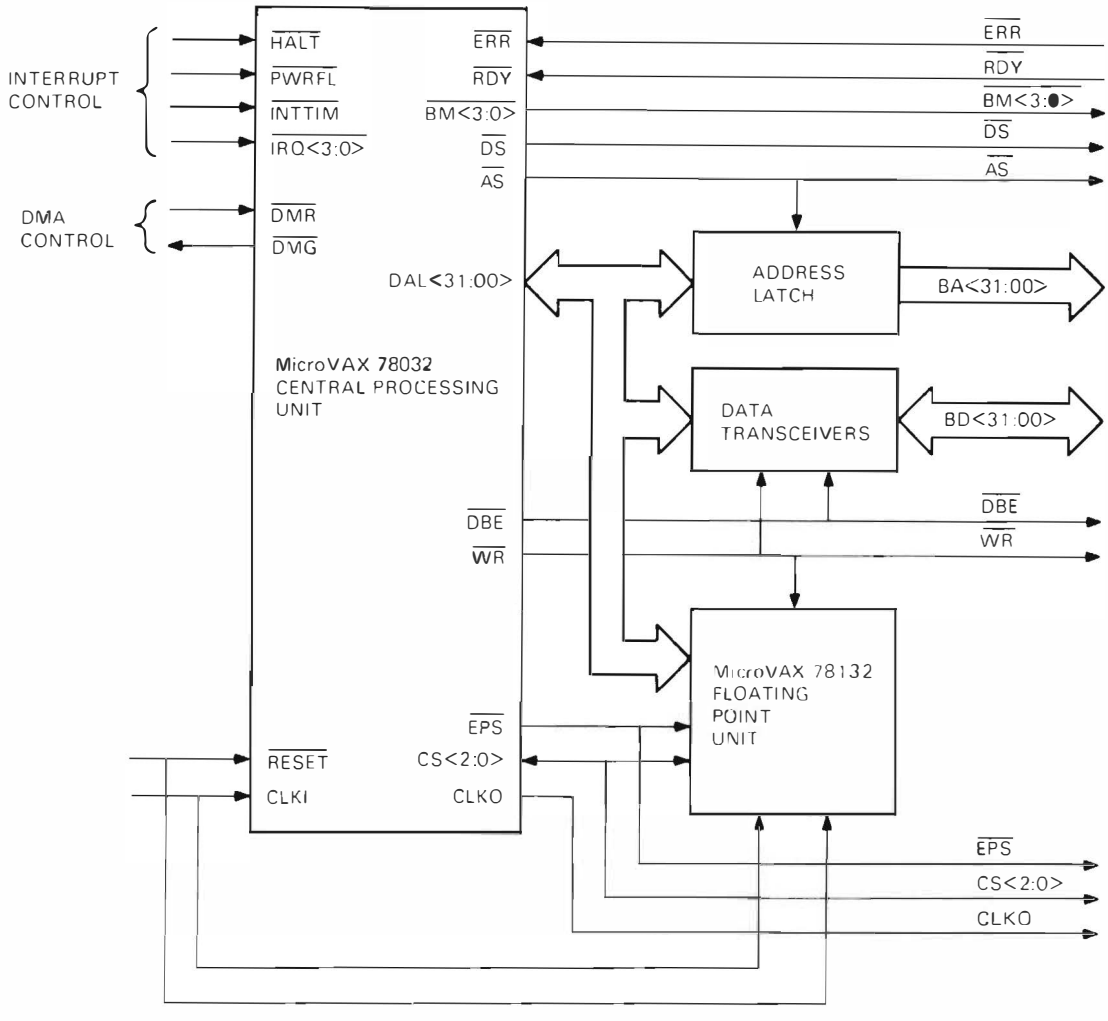


Figure 3 External Interface

Except for the 32-bit DAL bus, the external interface closely resembles those for existing 16-bit microprocessors. Specifically, its timing and signal complement are quite similar to those in current machines. The addresses and data on the DAL are time division multiplexed, with separate timing strobes (AS and DS, respectively, in Figure 3). The data direction and the data buffer signals (WR and DBE in Figure 3) are used to control external transceivers directly. The cycle status signals differentiate among the various types of bus transactions. Four-byte mask signals, one for each group of eight bits on the DAL bus, allow straightforward manipulation of bytes within longwords (four bytes).

The RDY signal allows slower peripheral devices on the I/O bus to stretch the memory access time beyond 400 ns until they are ready to respond.

Parallel Operation

Besides giving the 78032 optimized microcode and a fast microcycle time, the design team enhanced the chip's performance by allowing parallel operations between and within functional subsections. This parallel flow is actually a form of pipelining in which the operations happen independently and concurrently. For example, while the E Box is executing a datapath operation, the control store can access the next microinstruction. At the same time, the

microsequencer can be calculating the address of the microinstruction after that one, and the M Box can be translating a virtual address. Meanwhile, the I Box can be decoding an instruction or operand specifier and prefetching more instruction data. And the DAL interface can be initiating or completing an external bus operation.

For example, assume that the chip is to execute the following two three-microcycle macroinstructions in sequence:

ADDL3 R0, R1, R2

SUBL3 R4, R5, R6

Within the third 200-ns microcycle, some operations associated with these two macroinstructions are performed in parallel by several subsections. The E Box will write the result of ADDL3 into R2 in the register file, set the PSL condition codes, and check for arithmetic exceptions, such as an overflow trap. Meanwhile, the I Box will decode the next macroinstruction, SUBL3, and its first specifier, R4. Concurrently, the prefetcher in the I Box will determine if the decode of the instruction and specifier will clear enough space in the prefetch stack to warrant another longword transfer. If so, the I Box will then initiate the transfer and fetch another macroinstruction, which also involves the DAL interface.

Within each subsection, there are also a number of parallel operations that reduce the overall execution speed significantly. In addition to simultaneous prefetch and decode actions in the I Box (as described above), the microcode access in the control store is pipelined: The next microaddress is accessed while the current microinstruction at the current microaddress is being executed. In the M Box, length checks against referenced addresses take place simultaneously with the translation cache lookups. If a lookup misses, therefore, the length check will have already determined whether or not the referenced page is within range. In the E Box, a separate program counter (PC) adder maintains the PC so that the ALU can be dedicated to its primary task.

Some typical execution times for instructions under normal operating conditions (aligned operands, no memory management exceptions) are as follows:

Instruction	Operands	Typical Execution Time (Nanoseconds)
MOVL	Reg, Reg	400
ADDL2	Reg, Reg	400
MOVL	Mem, Reg	800
ADDL2	Mem, Reg	800
MOVL	Reg, Mem	600
ADDL2	Reg, Mem	1200
Conditional Branch, not taken		200
Conditional Branch, taken		800

Physical Implementation

The MicroVAX 78032 chip is made using a 3-micron, double-metal NMOS process that allows power savings and superior circuit flexibility. Until the MicroVAX 78032 chip design, single metal was a standard for NMOS technology. The use of a second layer on the 78032 chip was a significant departure for NMOS design. There are two main advantages of a double-metal implementation. First, it is easier to place logic circuits in the interconnect layer, where there are more circuits per unit area of silicon. Second, the metal interconnect has lower resistance than polysilicon, thus avoiding wire delays that are difficult to eliminate in design.

The double-metal process provided the chip design team with two layers of aluminum interconnect and four types of devices (N, E, L, and D). The four types allow some savings in power and a substantial increase in circuit flexibility. However, the E device (light enhancement) is typically used only in source-follower circuits, and the L device (light depletion) only in latches and static memories. The second layer of aluminum interconnect manages the complexity associated with 32-bit microprocessors. That permits global communications and allows local control or routing to share the same chip area. However, second metal can only contact first metal, and then only through an offset, or staggered, contact.

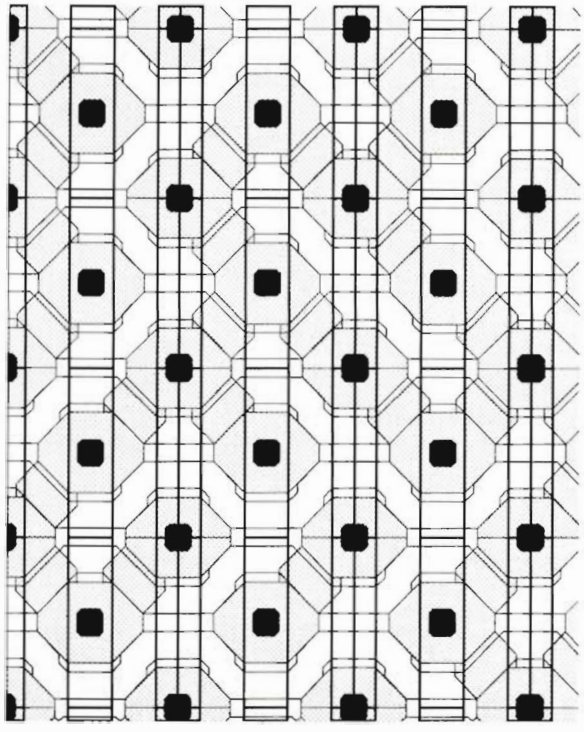


Figure 4 X-shaped Cells

The control store is a 1600-entry by 39-bit ROM. Although its size was decreased mostly through repartitioning and optimized microcode, about ten percent of the reduction was gained through the cell structure chosen. X-shaped cells with a virtual-ground design were used (Figure 4). This ROM has no physical ground, whereas standard ROMs with H-shaped cells have one ground line for every two data lines. The X-shaped cell, which is 95 microns square, is also more dense than the standard cell. Moreover, in the X-shaped cells, second metal is strapped across the top of the array to minimize the row propagation time. The cell access time is 100 ns.

The ROM bit lines are precharged to V_{CC} using depletion pullups. Sensing is done with a cross-coupled stage using local depletion divider voltage references set at $0.6 \times V_{DD}$. Column access occurs in 25 ns.

The control circuits (at the center in Figure 1) are implemented in dynamic logic so that the total power dissipation is kept below three watts. That also allows a low-cost packaging design. The eight clock phases provide refresh timing references to the dynamic logic.

Due to tight silicon constraints, the test features built into the design had to be limited in scope. The principal ones used are as follows:

- Serial shift registers with feedback for observing the control store, IPLA, and microsequencer outputs
- Special test mode for overriding normal sequencing with external microaddresses
- Dedicated microcode for optimizing state observations in the special test mode

Summary

The MicroVAX 78032 represents a major breakthrough both in semiconductor technology and in the VAX family. From a technology perspective, it is the first implementation of a successful 32-bit superminicomputer on a single chip. It is the first chip to provide integral demand-paged virtual memory management. And it is the first chip to provide system performance comparable to the 11/780. From a VAX perspective, the 78032 is the key to the downward extension of the industry-standard VAX family into the realm of small systems and workstations.

Acknowledgements

The authors acknowledge the technical contributions of John Beck, Sandy Carroll, Gerry Cheney, Mary Jo Doherty, John Glynn, Jim Gorr, Bob Grondalski, Dave Grondalski, Pat Hart, Ernie Hohengasser, Taan Lee, Steve Morris, Tony Pasquito, Steve Thierauf, Tim Thrush, Janet Vitello, and Barry Worster.

References

1. *VAX Architecture Handbook* (Maynard: Digital Equipment Corporation, Order No. EB-19580, 1981).
2. W.N. Johnson, "A VLSI Superminicomputer CPU," *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (1984): 174-175.
3. J. Slager et al., "A 16-bit Microprocessor with On-chip Memory Protection," *International Solid-State Circuits Conference Digest of Technical Papers* (1983): 24-25.

4. H.M. Levy and R.H. Eckhouse, *Computer Programming and Architecture: The VAX-11* (Bedford: Digital Press, 1980).

5. D.W. Clark and J.S. Emer, "Measurement and Analysis of Instruction Use in the VAX-11/780," *IEEE Proceedings of the 9th Annual Symposium on Computer Architecture* (1982): 9-17.

6. J.S. Emer and D.W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *IEEE Proceedings of the 11th Annual Symposium on Computer Architecture* (1984): 301-310.

7. W.D. Strecker, "Transient Behavior of Cache Memories," *ACM Transactions on Computer Systems*, vol. 1, no. 4 (November 1983): 281-293.

8. D.W. Clark, "Cache Performance on the VAX-11/780," *ACM Transactions on Computer Systems*, vol. 1, no. 1 (February 1983): 24-37.

9. G.M. Tarolli and W.J. Herman, "Hierarchical Circuit Extraction with Detailed Parasitic Capacitances," *ACM IEEE 20th Design Automation Conference Proceedings* (1983): 337-345.

William R. Bidermann
Amnon Fisher
Burton M. Leary
Robert J. Simcoe
William R. Wheeler

The MicroVAX 78132 Floating Point Chip

A separate chip, the 78132, in the MicroVAX II system performs fast floating point calculations. Three datapaths, each controlled by microcode, work in parallel to yield a 100-nanosecond microcycle. The wide datapaths accommodate a large variety of instructions, using microwords of only 35 bits for control. The 78132 is a 3-micron NMOS chip connecting to the CPU chip of the MicroVAX II system via a general-purpose protocol and a limited set of lines. Crosstalk and resistivity posed particular design problems, as did the routing of signals and power. The 78132's electrical integrity was carefully checked to ensure high reliability.

Scientific and engineering applications require strong floating point support from their computers. All VAX implementations offer both microcoded (warm) and hardware (hot) capabilities to execute the 95 floating point instructions in the full VAX instruction set. The MicroVAX II processor also supports floating point instructions, but in a slightly different fashion. Since the control store in the microprocessor, the CPU chip, has a limited size, these instructions are not executed in microcode; instead they are emulated in macrocode.[1,2] Emulation is relatively slow and does not provide the fast speeds required for intensive mathematical applications. Therefore, a separate floating point accelerator (FPA), the MicroVAX 78132 chip, has been developed as a companion to the CPU chip, the MicroVAX 78032 chip.

The 78132, or FPU chip, is designed to provide fast floating point calculations on a single chip. It executes 61 of the 70 floating point instructions in the MicroVAX instruction set. Nine of the 70 instructions simply move data, and the CPU chip does not need the FPU chip to handle them. The FPU chip also accelerates calculations for 9 integer instructions, which are associated with integer multiplies and divides. The FPU chip executes instructions about 100 times faster than macrocoded emulation.

The FPU chip (Figure 1) contains 32,141 transistors in a 3-micron, double-metal NMOS chip, which requires just under 2 watts of power at 5 Vdc. It measures 8.4 by 6.6 mm and is packaged in a 68-pin leaded chip carrier. The chip has a 100-nanosecond (ns) microcycle, divided into four 25-ns clock phases generated from a 40-MHz input clock. The CPU chip, which also operates on a 40-MHz input clock, has a microcycle of 200 ns. The faster microcycle and wide datapaths enable the FPU chip to perform floating point operations much faster than the CPU chip with its general datapath.

This paper discusses the implementation of floating point in the MicroVAX II's FPU chip and the unique constraints of a single-chip floating point accelerator. These constraints are not limited only to architecture but include interface design, wiring, and signal integrity, all areas where design trade-offs are important.

At the highest level, the FPU chip implements the F, D, and G floating point instructions in the VAX instruction set. The chip is constrained by the requirements of the VAX architecture–data formats, accuracy requirements, and instruction vagaries–and by the characteristics of the technology–limited number of pins, limited die size, and limited inter-

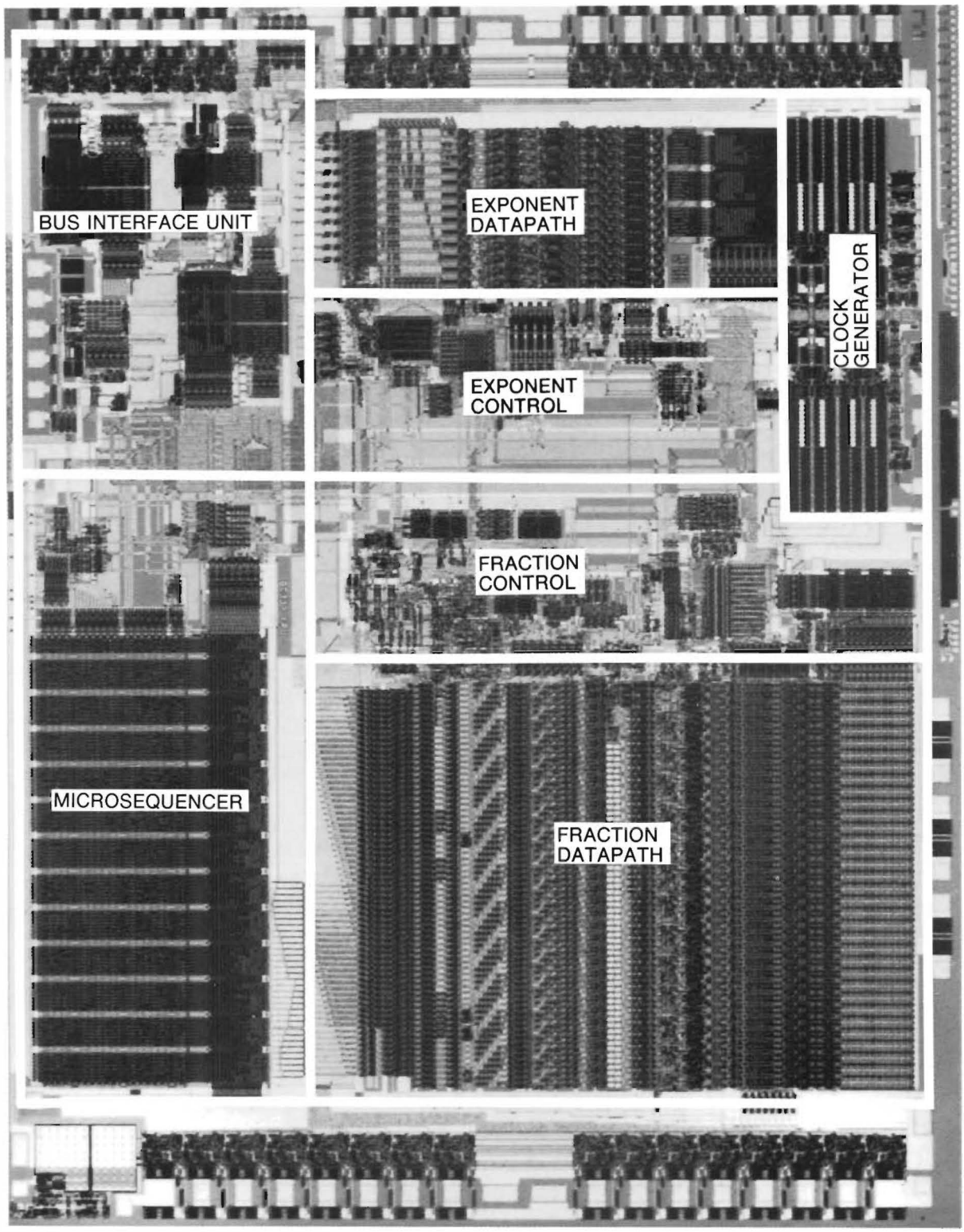


Figure 1 Photomicrograph of the FPU chip

connect. These constraints dictated many of the design considerations in the FPU chip.

FPU Chip Architecture

The main elements of the FPU chip, shown in the block diagram in Figure 2, are similar to those in most floating point devices.[3] Three separate processors–a 67-bit fraction processor, a 13-bit exponent processor, and a single-bit sign processor–operate in parallel. The bus interface unit handles data transfers over the external bus to the CPU chip and data movement into and out of the three datapaths. The microsequencer controls the parallel operations of the processors.

Each element in the FPU chip operates in parallel to speed up instruction processing. The microsequencer steps through the microcode for an instruction and determines which operation is to be performed by each processor for the current cycle. The microsequencer also takes inputs from each of the processors to determine which microword is to be executed next. The datapath of the fraction processor performs all the arithmetic computations on the mantissa of a floating point number. This datapath is designed to be flexible enough to handle the many different operations required in a general-purpose FPA. The datapath is also segmented to handle the F, D, and G data types, and is optimized to provide the maximum possible performance from the N-channel MOS technology.

The datapath of the exponent processor handles only the exponent portion of a floating point number. The exponent datapath is also used as a counter during certain operations such as multiply and divide. This datapath does all the exception and bounds checking for operations like addition and subtraction. The sign processor is incorporated into the exponent datapath and handles all operations pertaining to the sign bit. During an addition or subtraction, the sign bit determines which case is performed by checking the signs of the two operands and the opcode of the instruction.

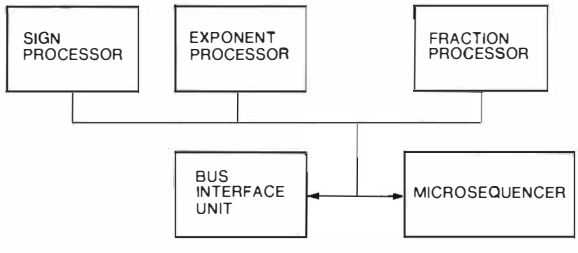


Figure 2 Block Diagram of the FPU chip

The bus interface unit (BIU) is responsible for handling all the FPU portions of the bus traffic between the FPU and CPU chips. The BIU decodes the opcode sent to the FPU chip and tells the microsequencer which instruction to execute. That allows the FPU and CPU chips to coordinate their actions without a lot of protocol or pins. Since many different data types are processed, the BIU is responsible for unpacking the operands and steering them to the appropriate datapath. Once the instruction is completed, the BIU takes the unpacked result from each datapath and formats the result into the specified data type. Figure 3 contains a more detailed block diagram for the entire floating point unit.

Algorithms

To keep the FPU chip at a size that could be produced, we decided not to use special-purpose hardware to implement instructions like addition or multiplication. Instead, the datapaths are designed to be general-purpose ones to accommodate the needs of a wide variety of instructions.

Addition and Subtraction

The datapaths are under microcode control and work in parallel. Within each, the steps required for either addition or subtraction are done serially. First, the exponents of the two operands are compared to see if they are of equal magnitude. If not, the larger exponent is stored in a register, and the exponent difference is used to control the alignment. The shifter on the output of the fraction arithmetic logic unit (ALU shifter) allows the fraction with the smaller exponent to be aligned five bits at a time. During each alignment step, the exponent difference is reduced by up to a magnitude of five until the exponents are equal. Once equal, the fractions are added. (In subtraction, the fraction to be aligned is complemented before alignment.)

The resulting fraction is then normalized. The normalize shift is accomplished by a single left shift in the fraction ALU and two left shifts in the ALU shifter. If the addition of the

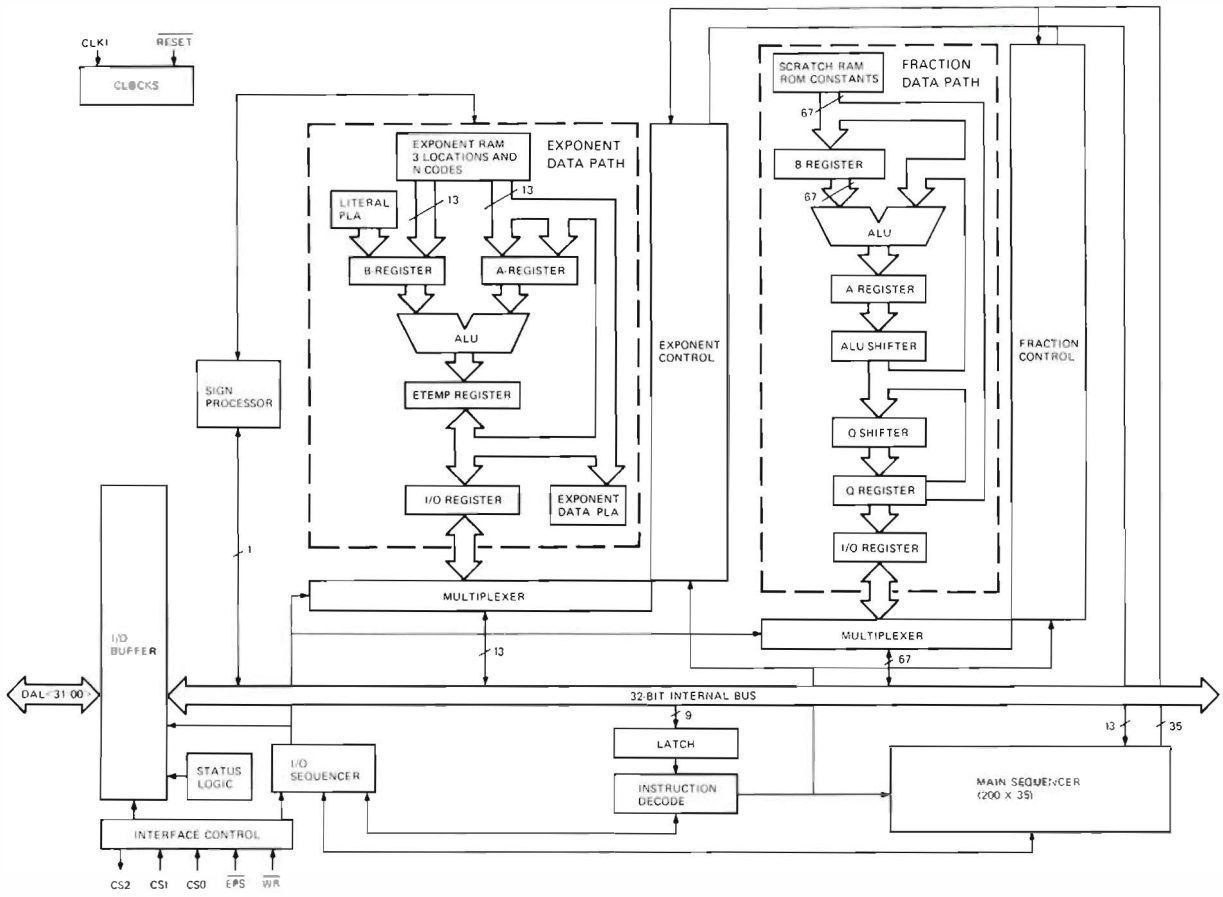


Figure 3 Block Diagram of the FPU Processor

fractions results in an overflow into the top guard bit, a single right shift in the ALU shifter is required to normalize the result. During normalization, a 3-bit code is sent to the exponent datapath, which determines the amount the exponent must be adjusted.

After normalization, the fraction is rounded using a rounding constant appropriate for the data type of the floating point operation being performed. If the round results in an overflow in the fraction datapath, the exponent is incremented by one and the fraction is normalized. The exponent datapath then checks the resulting exponent for any error conditions. If no errors are found, the final fraction and exponent values are loaded into the output register and the sequencer signals the BIU that the operation is complete.

Multiply

The multiply operation in the FPU chip is based on a 3-bit retirement algorithm. The 3-bit retirement, or octal multiply, must generate the required multiple, 0–7, of the multiplicand to be added into the partial product for each step. The multiples must be generated by simply shifting the multiplicand and adding or subtracting them from the partial product. The multiples 0, 2, 4, and 8 are easy to generate in this way. The multiple 6 can be formed by taking three-quarters of the multiplicand and storing that in a register at the beginning of the multiply ($¾ \times 8 = 6$). As shown in Table 1, all the even multiples can be generated. To generate all the odd multiples, a -1 multiple is added to achieve the final exact multiple for each retired group of three bits.

Table 1 Multiply Operation – Booth Encodings

Multiplier Group	Required Multiple	Data Used	Multiple Shift	Multiple Added	Multiple Owed
000	0	0	0	0	0
001	1	mult	1	2	−1
010	2	mult	1	2	0
011	3	mult	2	4	−1
100	4	mult	2	4	0
101	5	¾ mult	3	6	−1
110	6	¾ mult	3	6	0
111	7	mult	3	8	−1

The key to making this scheme work is that this −1 multiple must be generated from the previous group of three bits. To that group, the −1 multiple for the next group is equivalent to a −8 multiple. To know whether or not the next group will need the −1 multiple, it is sufficient to examine the least significant bit (lsb) of the next group of bits. If the lsb is a 1, then the group will be odd and will need the −1 multiple. This process is started by examining the lsb of the multiplier and initializing the partial product register to either zero or minus the multiplicand. If the lsb is a 0, the −1 multiple will not be needed. The operation always terminates in the case not requiring compensation because the numbers are all normalized. Table 1 shows the Booth encodings for each multiplier group.

These Booth encodings translate into the fraction datapath controls depicted in Table 2.

A multiplication in the FPU chip is begun by loading the multiplier into the Q Register (quotient register) and loading the multiplicand into register 0 in the scratch RAM. Three-quarters of the multiplicand is then calculated during two ALU cycles and is stored in register 1 of the scratch RAM. Subsequently, the A Register is initialized to store the partial products.

During each cycle of the multiply loop, the four least significant bits of the Q Register are latched to control each multiply step. Based on these four bits, the multiply control loads either the multiplicand or three-quarters of the multiplicand from the scratch RAM into the B Register. The control then adds or subtracts the B Register from the A Register. The resulting new partial product is shifted right by the ALU shifter and relatched in the A Register. The Q Register is then shifted three bits to the right to retire the current set of multiplier bits and to set up for the next iteration.

The exponent datapath is used to control the number of iterations that should occur for each multiply operation and to calculate the resulting exponent. The number of iterations that take place for a multiply depends on the length of the mantissa. For example, an F format number with a 23-bit mantissa requires eight iterations.

Division

The floating point unit performs a 1.5-bit, non-restoring division. This algorithm is similar to a 1-bit, non-restoring division, but takes advantage of the fact that long strings of zeros or ones in the partial remainder can be skipped over without doing an addition or subtraction. The FPU chip handles double precision through its normal datapath.

Within the FPU chip, the partial remainders will always be $< +½$ and $> -½$ because both floating point numbers are normalized. If the partial remainder is small relative to the normalized divisor, a 1 will not be shifted into the quotient over the next few cycles. (The opposite is true if an addition is performed.) Knowing this fact and whether the previous operation was an addition, subtraction, or a shift will determine how the quotient bits are developed. If the previous operation was a shift, the process is in the middle of a long string of zeros or ones and no addition or subtraction has to be performed. If the partial remainder is not small relative to the normalized divisor, the quotient bits are developed as they would be in a 1-bit

Table 2 Multiply Operation – Fraction Datapath Controls

Next Group Look Ahead	Actual Multiple	Present Group	Group Multiple	Group Multiple	Multiple Generated	ALU Operation
0	0	000	0	0	0	A ← A
0	1	001	2	0	2	A ← A+B; B=R0
0	2	010	2	0	2	A ← A+B; B=R0
0	3	011	4	0	4	A ← A+B; B=R0
0	4	100	4	0	4	A ← A+B; B=R0
0	5	101	6	0	6	A ← A+B; B=R1
0	6	110	6	0	6	A ← A+B; B=R1
0	7	111	8	0	8	A ← A+B; B=R0
1	0	000	0	−8	−8	A ← A−B; B=R0
1	1	001	2	−8	−6	A ← A−B; B=R1
1	2	010	2	−8	−6	A ← A−B; B=R1
1	3	011	4	−8	−4	A ← A−B; B=R0
1	4	100	4	−8	−4	A ← A−B; B=R0
1	5	101	6	−8	−2	A ← A−B; B=R0
1	6	110	6	−8	−2	A ← A−B; B=R0
1	7	111	8	−8	0	A ← A

where: R0 contains the multiplicand
R1 contains ¾ multiplicand

division algorithm. Table 3 summarizes the 1.5-bit, non-restoring division.

The implementation of this algorithm in the FPU chip is straightforward. To start, the divisor is loaded into the B Register and the dividend into the A Register. The Q Register is initialized to 0 and will become the location where the quotient is developed.

During each step of the division, quotient bits are inserted at the least significant end of the Q Register. The register contents are then shifted left either 1 or 2 as required to develop the new quotient for that step. If necessary, the divisor is added to or subtracted from the partial remainder. The result is then shifted left by the appropriate number of places.

When bit 65 in the Q Register becomes a 1, the division is stopped. Since these numbers are normalized, the result will fall in the range of greater than ½ but less than 2. The contents of the Q Register, already normalized, are then read back into the A Register. However, if the initial subtraction resulted in a positive partial remainder, then one must be added to the exponent to account for the fact that the result has a whole part (i.e., $\geqq 1$).

Integer Division

The FPU chip also performs a 1-bit, non-restoring divide algorithm, which is used to accelerate the execution of the DIVL and EDIV instructions. In all cases, the integer divide is accomplished with a 32-bit divisor and a 64-bit dividend.

Polynomial Calculations

The polynomial evaluation algorithm, POLY, uses Horner's Method to calculate all trigonometric functions. Because execution time can be so long, POLY is the only VAX floating point instruction that can be interrupted by the CPU chip. The algorithm performs a series of $ax+b$ operations once during each cycle. In each operation, x is treated as a constant, the value of b is provided by the CPU chip, and the value of $ax+b$ in the current cycle becomes a in the next cycle.

The FPU chip first multiplies a by x with the MUL algorithm and then adds b with the ADD algorithm. The main sequencer tells the I/O controller that the first POLY cycle has been completed and that the result is ready in the

Table 3 1.5-Bit Division Operation

Most Significant Bits of Partial Remainder								
66	65	64	63	Value of bits 66–63	Shift Left	ALU Operation	Add/Sub Quotient	Shift Quotient
0	0	0	0	0	2	none	10	00
0	0	0	1	1/8	2	subt	10	00
0	0	1	0	1/4	1	subt	1	0
0	0	1	1	3/8	1	subt	1	0
1	1	0	0	>−1/2	1	add	0	1
1	1	0	1	−3/8	1	add	0	1
1	1	1	0	−1/4	2	add	01	11
1	1	1	1	−1/8	2	none	01	11

Add/Sub Quotient: Bits shifted into the quotient if previous operation was an addition or subtraction.
Shift Quotient: Bits shifted into the quotient if the previous operation was a pure shift (no ALU operation).

I/O registers for transfer to the CPU chip. The sequencer executes the second MUL, $(ax + b)x$, during the time that the CPU chip is reading the first result, storing it in a register, and transferring the next value of b to the FPU chip. The second ADD operation, $(ax + b)x + b$, then takes place to complete the second cycle, and the process continues. The CPU chip's register is updated with the new result at the end of each cycle. This pipelining allows fast generation of trigonometric and transcendental functions. Both the CPU and FPU chips are working to implement the instruction, and the actual multiply time is overlapped by the operand fetch time.

The Microsequencer

The microcode for the FPU chip is contained in a large programmable logic array (PLA), which is the heart of the microsequencer. Inputs to the PLA are received from all major sections of the FPU chip. A microword of 35 bits is all that is needed to control the two main datapaths (the sign processor is part of the exponent datapath) and to communicate with the bus interface unit. Each field in the microword is encoded to reduce the number of wires routed to the other sections. Two hundred microwords are required to implement the sixty-one floating point and nine accelerated integer instructions executed by the FPU chip. The block diagram for the microsequencer is shown in Figure 4.

Inputs to the PLA are comprised of five next-address bits, three dedicated inputs, and forty signals from the three major processors on the chip. Three bits from the next-address field are used to select five of the forty signals for the next FPU cycle. These five multiplexed inputs, in conjunction with the eight direct inputs, are used to address the next microword. The thirty-five outputs, or signals, from the PLA are used to communicate with the rest of the floating point unit. These signals determine which operation is to be performed by each of the three datapaths (exponent, fraction and sign processor).

Interface Between Chips

Interface Lines

The communication between the CPU and FPU chips is done through a very limited set of lines: a write (W) strobe, three cycle status (CS) lines, an external processor strobe (EPS), and the 32-bit data and address lines (DAL). (This approach was used to reduce the pincount on both chips.)

In the MicroVAX II processor, the chip protocol is designed as a general-purpose one so that other coprocessors could take the place of the FPU chip. Each interface line has a specific purpose, as explained below.

- The W strobe sends a signal from the CPU chip to indicate the direction of data flow over the DAL. For the FPU chip, the write

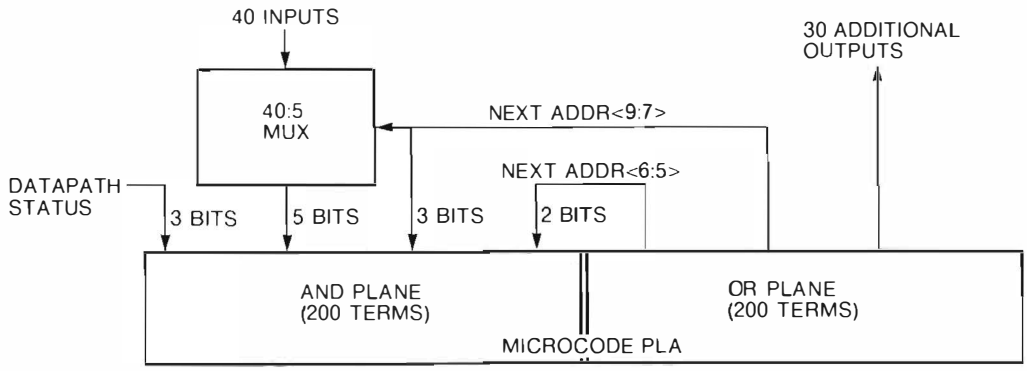


Figure 4 Block Diagram of the Microsequencer

signal indicates that data is being transferred from the CPU chip.

- The EPS is used by the CPU chip to qualify all communication between itself and the FPU chip or other non-memory device.
- The three CS lines provide status about the current bus cycle. Two of the lines indicate the type of information being transferred; they are "valid" when the external processor strobe is asserted. The third line is an open-drain output (functionally similar to an open collector in TTL), which will be active when the bus cycle is a response enable and the FPU chip has completed the current commanded operation.
- The DAL is a 32-bit, bidirectional bus that exchanges data between the CPU and FPU chips. The CPU chip is always the bus master and controls the transfer of operands to the FPU chip and results back to itself.

The information exchanged between the CPU and FPU chips could be of different types: write external processor command, read or write external processor data, command to other external processors (not the FPU chip), and external processor response enable. The external processor strobe (EPS) is used by the CPU chip to qualify all communication between itself and the FPU chip.

Figure 5 illustrates all the interface lines between the two chips.

Communications Protocol

The communications protocol permits the FPU and CPU chips to communicate efficiently. Every interchip operation will be associated with the following sequence of bus activities:

1. The CPU chip initiates an interaction by placing a command onto the DAL bus, a status code on two CS lines, a write signal of "low," and an EPS of "low." The FPU chip recognizes this sequence as a command-write cycle and aborts any instruction being executed. The FPU chip then decomposes the command to determine the required operation and the number and size of the operands.
2. The CPU chip fetches the required operands and executes one or more data-write cycles to transfer them to the FPU chip.
3. After transferring the last operand, the CPU chip asserts a response-enable signal on the CS lines and pulses the EPS "low." The chip does that once for each microcycle that it has control of the bus in order to determine if the FPU chip has finished processing the data.
4. To signal the completion of operations, the FPU chip asserts the CS<2> line "low" when the response-enable signal is on the two CS lines and the EPS is "low." At the same time, the FPU chip asserts the status of the just-completed operation.
5. The CPU chip recognizes the "low" signal from the FPU chip and reads the status information. The CPU chip will repeat this transaction to compensate for

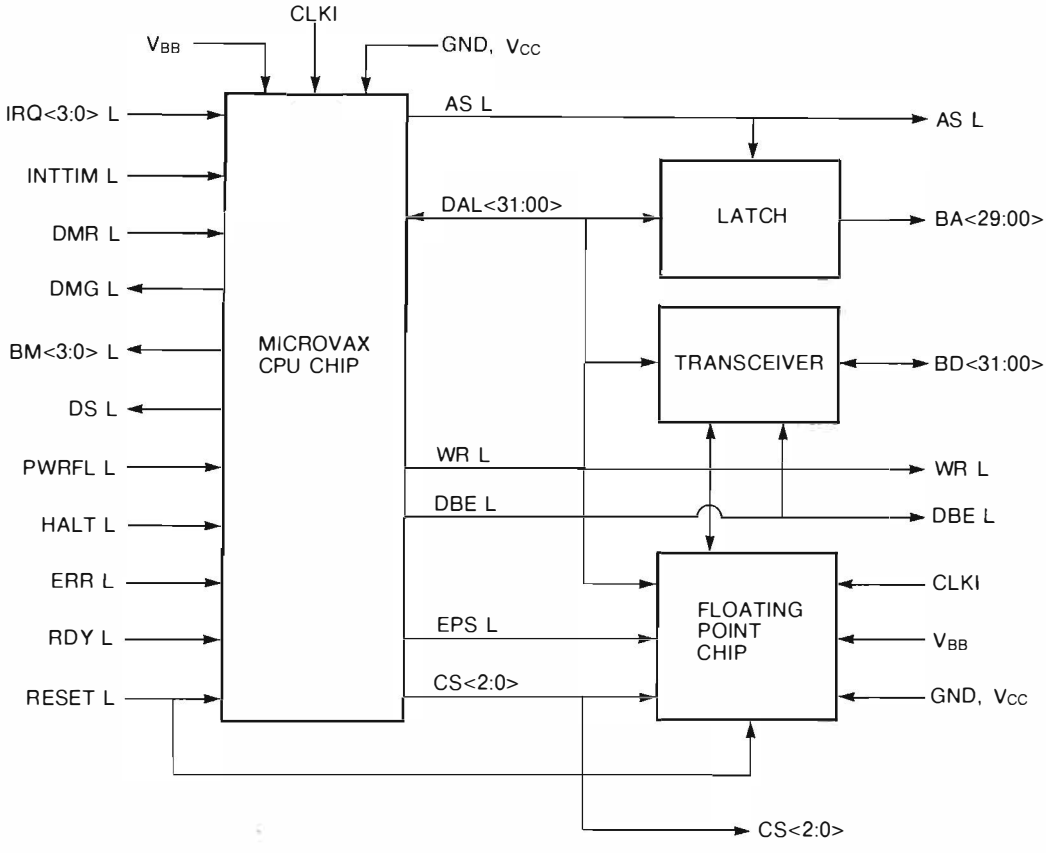


Figure 5 Interfaces Between the CPU and FPU Chips

its microcoded pipeline, capturing the status information the second time.

6. The CPU chip executes zero or more data-read cycles to read the results, if there are any, from the FPU chip. Both chips are now free to perform the next transaction in the instruction stream.

(The FPU chip will respond unpredictably to other nonstandard protocols and relies on the sequence of interactions described above for proper operation.)

Performance Analysis

The performance of the FPU chip is very sensitive to the I/O bandwidth. Every floating point operation is associated with a specified sequence of events that must occur between the chips before the execution can start. There is another sequence of events that must take place when the computation is completed. These sequences happen without any parallelism or pipelining.

The protocol affects the performance of the FPU chip because cycles must be expended for sending and reading status signals, and transferring data. Table 4 illustrates the individual steps that occur for three types of operations: ADDF, MULF, and MULD. For these examples, assume that no time is spent on instruction fetch and decode, and that the memory subsystem has an unlimited bandwidth and buffering capability for reads and outstanding writes. The performance is measured from the completion of the initial instruction decode to the final result store in the memory (or a register).

The total execution time for other instructions can be derived in the same manner using the following internal execution times:

Add in D format – 700 ns

Division in F format – 2200 ns

Division in D format – 4400 ns

Table 4 Steps for Add and Multiply Operations

Instruction: ADDF	Register Mode		Byte Displacement	
	Protocol Time (nanoseconds)	Execute Time (nanoseconds)	Protocol Time (nanoseconds)	Execute Time (nanoseconds)
Specifier decode and data transfer for first operand	300		500	
Specifier decode and data transfer for second operand	200		500	
Internal transfer (first operand)		100		100
Execution		600		600
Status read	200		200	
Status read	200		200	
Result transfer on DAL bus	200		400	
Total	**1100**	**700**	**1800**	**700**
Total Execution Time:	**1.8 microseconds**		**2.5 microseconds**	

Instruction: MULF	Register Mode		Byte Displacement	
	Protocol Time (nanoseconds)	Execute Time (nanoseconds)	Protocol Time (nanoseconds)	Execute Time (nanoseconds)
Specifier decode and data transfer for first operand	300		500	
Specifier decode and data transfer for second operand	200		500	
Internal transfer (first operand)		100		100
Execution		1900		1900
Status read	200		200	
Status read	200		200	
Result transfer on DAL bus	200		400	
Total	**1100**	**2000**	**1800**	**2000**
Total Execution Time:	**3.1 microseconds**		**3.8 microseconds**	

Instruction: MULD	Register Mode		Byte Displacement	
	Protocol Time (nanoseconds)	Execute Time (nanoseconds)	Protocol Time (nanoseconds)	Execute Time (nanoseconds)
Specifier decode and data transfer for first operand	400		600	
Specifier decode and data transfer for second operand	300		600	
Internal transfer (first operand)		100		100
Execution		2700		2700
Status read	200		200	
Status read	200		200	
Result transfer on DAL bus	400		800	
Total	**1500**	**2800**	**2400**	**2800**
Total Execution Time:	**4.3 microseconds**		**5.2 microseconds**	

Wiring and Signal Integrity in the FPU

Signal integrity in a large VLSI chip such as the 78132 is fundamental to ensure correct functionality and good yield, given the variations in manufacturing. The one- to two-micron proximity of signal lines on an integrated circuit (IC) can cause significant coupling problems. Moreover, there are problems in terms of clock distribution and power-supply noise. The design of the logic must allow sufficient noise margin to permit correct operation in spite of the noise present in the system. The use of charge as the signal (used in many circuits in an NMOS design), rather than voltage or current, created some special design problems for the FPU chip team.

IC Wiring Characteristics

The FPU chip has four layers–two of metal, one of polysilicon, and one of diffusion–that are used to interconnect and form devices. The wiring in an IC is conceptually similar to the wiring on a printed circuit board. Although the total wiring length on the FPU chip is only about four meters, the interconnected nodes and elements number in the tens of thousands. Placing and routing the logic functions inevitably affects the estimates of loading and system performance. Thus an iterative process of first routing a design, then simulating the subsequent performance is needed to identify a workable routing plan. Once this workable routing-performance trade-off is identified, the final routing and loadings can be made.

The wiring considerations for a VLSI design are different from those for conventional systems in several ways. First, the dimensions are smaller. In the NMOS process the horizontal metal separation is about three microns and the vertical separation is from one to two microns. Even with the smaller size of the wiring in the MicroVAX II chips, crosstalk can become a serious problem. On a MOS chip, crosstalk between poorly designed nodes can approach fifty percent. The capacitance on many of the critical nodes in the FPU chip is only about 100 femtofarads (0.1 picofarad). Any coupling at all on these nodes becomes quite significant. The largest capacitance on the chip is the clock lines at around 110 picofarads. On dynamic nodes, which rely on a charge stored on a capacitor to represent a logic level, this coupling is particularly troublesome.

To eliminate this problem on the FPU chip, the design team checked each of the over 12,500 nodes for crosstalk from all other nodes in the chip. This data was then used to change the layout, where appropriate, to minimize or in some critical cases, eliminate intolerable levels of crosstalk. These checks took about three man-months to complete.

Another difference in the wiring of a VLSI chip is the resistivity of the wiring. The metal layers in the FPU chip have resistivities on the order of 100 milliohms per square. However, the resistivities of the polysilicon and diffusion interconnect layers are about 40 ohms per square, or 400 times that of the metal layers. The interaction of this parasitic resistance with the on-chip capacitive loads can cause serious performance limitations if not carefully monitored.

In fact, these two layers are so resistive that they were unusable for unconditional routing of either signals or power; they could be used only for very local routing. As a special precaution, a hand-check of those layers was made at pattern generation time to verify that no long, speed-critical paths utilized these layers as part of the routing network.

Power and Signal Routing

A minimum-width wire routed the length of the FPU chip has a resistance of about 200 ohms. The use of metal layers with noticeable resistance therefore begins to set system performance limits through RC delays as well as IR drops, which happens in larger systems. The clock distribution introduces a delay of about one nanosecond across the FPU chip, due solely to the resistance of the metal interconnect and the distributed load capacitance. This delay amounts to about four percent of the length of a single phase in the chip. A well-monitored clock distribution system is a requirement in any semiconductor chip. The problem is that the performance of the underlying semiconductor device is beginning to outstrip the capability of the chip wiring to distribute the clock. RC delays become the limiting speed factor of the wiring in an IC, while the speed of light across transmission lines is the limiting factor in a larger system. These resistances can also

seriously affect the power and ground supply as it is distributed throughout the FPU chip.

We used several techniques to keep the supply noise under 200 mV as power is distributed throughout the chip. First, the total dc current was calculated by summing the current used in each power and ground line as it joined other branches on the route to the actual supply pad. At this point in the net, two factors had to be analyzed so that the width of the power bus could be sized correctly. That sizing kept the equivalent resistance low enough so that the overall drop from a pad to the most remote logic could be kept under 200 mV. Unfortunately, that sometimes required large (on an IC scale) power buses in which a significant fraction of an ampere must be provided by one supply line.

The second problem, and the more difficult one, associated with the power and ground wiring is the large ac voltage transients that can occur when large portions of the system switch at the same time. That problem is especially significant with the V_{SS} lines. And it is particularly difficult when driving wide buses or large datapaths as wide as the 81 bits in the FPU chip. In these cases, large transients (one ampere or more) flow in ground and power lines for a few nanoseconds. In a large system environment, decoupling capacitors can be used to supply these currents locally. Unfortunately, that is not possible in an IC environment where such large capacitors are not practical. As a result certain ground lines in the FPU chip are allowed to have significant noise on them. In some cases this noise spike can be as much as two volts. This noise is handled by running these "dirty" grounds in a separate metal line all the way back to the pad on the chip.

However, even when the line is taken back to the pad to prevent local IR drops from upsetting the logic, parasitic inductance in the packaging can still cause problems. The most striking example is that of off-chip bus drivers. Here a typical 32-bit bus is driven over 4- or 5-volt swings in as little as four or five nanoseconds. With each bus load being on the order of 100 pf, the large dI/dt that the chip imposes on the power pins causes inductive ringing. Solving this problem by placing a decoupling capacitor on the external pins is of little value since the package inductance effectively isolates the capacitor from the actual nodes it must decouple inside the chip. Therefore, the FPU chip, like most chips that drive wide buses, has separate power pins going only to the output transistors. The subsequent ringing is tolerated since it does not affect any internal logic. (The ringing can become even more of a problem on chips with several buses with different timings, since separate supplies must be used for each bus. That drastically increases the number of supply pins required on the chip.) The FPU chip devotes 19 of its 68 pins to V_{SS} and V_{DD} distribution.

Electromigration

A final wiring consideration in designing the FPU chip was electromigration. Electromigration is a reliability issue in IC wiring because high current density in the metal interconnect can cause the metal to migrate, thinning sections of wiring until they finally fail. Current densities much higher than 10^5 amperes per square centimeter can cause increases in wiring resistance and eventually, open circuits or increased interlevel leakage, and short circuits. Clock lines, power and ground buses, as well as some global wiring, are susceptible to this failure mechanism. As a result, all lines on the FPU chip have an additional current constraint imposed by electromigration. When the chip was designed, these lines all had to be checked to eliminate the problem.

Wiring Integrity

Considerable time was spent checking the electrical integrity of the wiring in the FPU chip. The following list contains the most important wiring integrity checks made of the interconnect on the chip:

1. Transistor Source/Drain Integrity – This check assured that the silicon interconnect resistance caused less than five percent degradation.
2. RC Delays – All RC delays greater than one nanosecond were analyzed.
3. Coupling – All internodal coupling capacitors were checked to verify that there would be less than 200 mV of noise injected into the node.
4. V_{DD} and V_{SS} Nets – Three checks were performed. First, all IR drops were measured to ensure that ac and dc voltage

sources were kept under 200 mV. Second, all buses were sized to verify their reliability for electromigration resistance. This check included contact electromigration. Third, a check ensured that sufficient isolated power pins existed to guarantee that clean and dirty grounds were isolated.

5. Clock – An analysis identical to that for V_{DD} and V_{SS} nets was done on all eight clock lines.

Although there were significant CAD tools to perform most of the checking, this task alone required approximately ten percent of the total engineering time for the entire project.

Summary

The VLSI chips we are now designing are as complex as several boards of TTL used in past implementations of the VAX architecture. The FPU chip performs the same functions at about the same speed as five boards containing ICs in the VAX-11/780 system. The designs of these complex systems on chips present a set of constraints and considerations similar to and yet different from those encountered by board-level system designers. We hope that this paper captures the complexity and uniqueness involved in the MicroVAX FPU chip.

Acknowledgements

The FPU chip team completed the design of two VAX floating point chips, the MicroVAX FPU and the 8200 chip, in eighteen months. That was possible only because another design team working on the J-11 FPA had established the basic architecture and took the time to help our team to understand that work. This close working relationship allowed us to complete the MicroVAX FPU design in step with the CPU chip team, which was our major challenge.

References

1. D.W. Dobberpuhl et al, "The MicroVAX 78032 Chip, A 32-bit Microprocessor," *Digital Technical Journal* (March 1986, this issue): 12-23.

2. R.J. Simcoe et al, "A Floating Point Unit for a 32-bit Microprocessor System," *Proceedings of the 1984 IEEE Custom Integrated Circuit Conference* (May 1984): 478-481.

3. 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.

Barry A. Maskas

Developing the MicroVAX II CPU Board

Within the MicroVAX II system, the CPU board provides an environment to optimize the performance of the CPU and floating point processor chips. The board is designed as a linked sequential machine to accommodate the sequential control of the CPU chip. A Q-bus handles I/O for the system. The memory access path is dual ported, allowing the memory and the CPU chip to run synchronously without wait states. A scatter-gather map provides Q-bus address translations. To minimize product delivery time, the CPU board was developed in parallel with the chips. Using CAD tools helped to go from first-pass chips to running the MicroVMS system in only two weeks.

The CPU board in the MicroVAX II system (Figure 1) holds two chips: a microprocessor, called the CPU chip, and a floating point coprocessor, called the FPU chip. The board also integrates a synchronous memory subsystem, a synchronous I/O-bus controller, and a synchronous on-board I/O subsystem. The project to develop the CPU board was governed primarily by time-to-market considerations.

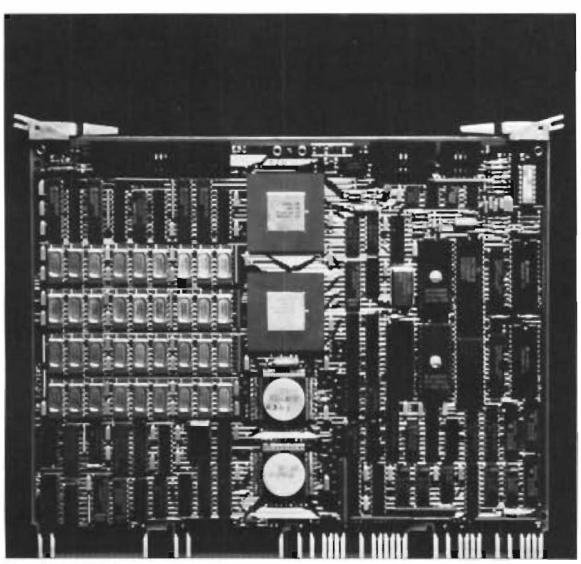


Figure 1 The MicroVAX II CPU Board

Other factors, such as VMS and ULTRIX compatibility, performance, reliability, cost, and ease of high-volume production were also important criteria. The end result is a successful balance between all these factors.

Development Goals

The importance of the primary goal governed how the project team organized itself to make decisions and to execute tasks. Rapid decision-making, and parallel and overlapping activities were the norms for this development effort. Unfortunately, parallel activities can cause communication problems, thus increasing the risks of product failure. However, these problems were anticipated and mechanisms put in place to reduce the risks to an acceptable level.

The CPU board was designed around the specifications of the CPU and FPU chips, which were being developed at the same time. Therefore, one development goal was to minimize the dependency of the board design and layout on the first-pass designs for these chips. The team aimed at providing a fully functional system environment into which the first-pass chips could drop. This aggressive approach lead the team to leap-frog over events rather than to take a conventional stepping-stone progression. The overall project manager encouraged the taking

of prudent risks because he was responsible for meeting the development schedule. The acceptance of these risks eventually paid off in an on-time delivery of the CPU-board design.

Single-board Design

Developing the CPU board around the two chips required us to provide a specific system environment. That environment had to balance the memory bandwidth of the CPU chip against its I/O bandwidth requirements. The realization of that balance is the key to the board's success. Having either a slower memory or a slower I/O subsystem would degrade system performance by at least twenty-five percent. The environment also had to support the MicroVMS, ULTRIX, and VAXELN operating systems.

Our goal was to provide the hardware specified by the three operating systems on one Digital-standard quad-sized board (8-½ by 10-½ inch). The single-board goal was a consequence of technology improvements balanced by the costs of replacing the unit in the field. In this case, needing fewer pieces to build the system would reduce manufacturing costs, improve reliability, and ease maintainability costs. The objective of operating at the full bandwidths of the chip and the I/O bus was especially challenging when so little board space was available for the necessary functions.

Most new chips do not run at their full speed immediately; they take some time to debug. Our design objective was to run the CPU chip at an operating frequency lower than its maximum during the first-pass debug. Of course, running at a slower clock rate was never an acceptable compromise for the final product. (Two versions of the CPU board were developed with minimal component differences, one running at the full 200-nanosecond (ns) microcycle speed and the other at a slower 242-ns microcycle speed.) However, if the first-pass chip had missed its performance target, the development of the CPU board could still have continued. It is a tribute to the chip designers that the first-pass chips did run at full speed, which was quite unusual in so complicated a product.

The bus chosen to meet the I/O needs of the system was the Q22-bus. This 22-bit bus has sufficient bandwidth to handle traffic from the system disk, the Ethernet LAN, and other I/O sources, such as other processors. The risk of using this bus was low due to its proven design, and the development cost for this application was reasonable. The Q22-bus is also supported by many disk, tape, and other I/O products from both Digital and third-party add-on manufacturers.

CPU Board Functions

We ruled out using the Q22-bus for accessing memory directly, since the bus could not meet the memory cycle time of 400 nanoseconds for the CPU chip.[1] Therefore, a new memory architecture had to be developed. We investigated two alternative schemes, the first being the widely used direct memory access (DMA) with a single port. Unfortunately, DMA forces addresses and data to cross the microprocessor bus on their way to memory. The usual procedure is to halt the microprocessor with a DMA request or grant while the DMA device uses the microprocessor's data and address paths. In this case the CPU chip, having no cache, would waste time by exercising the memory request and memory grant signals. Therefore, we chose the second scheme, a dual-ported memory controller. Figure 2 depicts the single- and dual-ported memory controllers that were considered.

This dual-ported controller requires that the CPU chip have different address and datapaths for the Q22-bus and the memory controller. While a DMA access is taking place, the CPU chip can continue operating on its 32-bit external datapath, primarily communicating with memory and the FPU chip. In this context, memory cycles can be pictured as strings of 400-ns time slots controlled by a central arbiter. This memory controller minimizes the impact on the CPU chip's performance by DMA accesses to memory on the Q22-bus. This organization is not locked up by asynchronous Q22-bus cycles, whose transactions are three to four times slower than the CPU chip's memory cycles. It also allows the Q22-bus protocol to operate autonomously with the CPU chip and memory, except when the buffered bus protocol and the memory system exchange buffered data.

The memory controller also serves as an alternative to one based on a cache. The CPU chip does not implement an internal cache due to power and chip-size constraints.[1] Cycles for

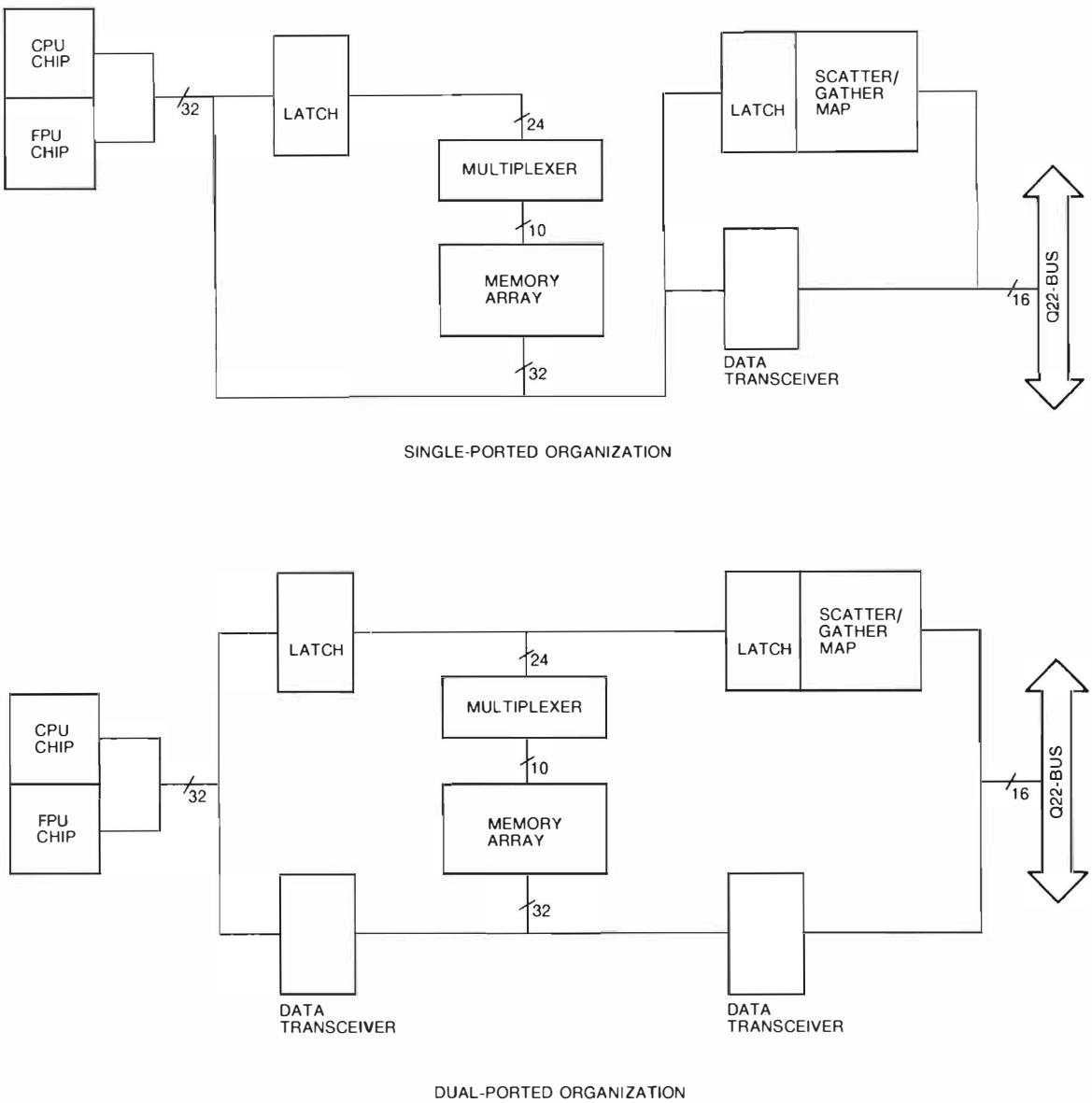


Figure 2 Block Diagrams of the Proposed Controllers

DMA, refreshing memory, and CPU-chip access are interleaved in time.

The MicroVAX II system is designed to be used in a multicomputing environment. Therefore, the bus interface logic has to accommodate the role of either bus arbiter or auxiliary processor. To that end, a doorbell register facilitates an interprocessor interrupt mechanism. The datapath of the Q22-bus interface has to provide the address translations from the virtual memory space of the bus to the address space of memory.

We defined several other elements as being essential for supporting an operating system on a single board. Those are the time-of-year (TOY) clock, the console serial line, the VAX console command program, and the console-interface-boot and self-test ROM. These elements, along with some status and error registers, comprise the on-board I/O subsystem.

The functional organization of the CPU board is depicted in Figure 3.

Linked Sequential Machines

Optimizing the overall computer performance means that data transfers between the CPU chip and memory have to be as fast as the chip can operate. Without a cache memory, the CPU chip has a relatively long memory cycle time of 400 ns (two microcycles). Thus CPU chip-to-memory data transfers can take place without wait states.

The 400-ns I/O cycle is nevertheless fast enough that the CPU board had to be designed as a linked sequential machine rather than as flow-through logic. The control function in the MicroVAX II system receives signals, interprets them, and generates control outputs, all in a defined sequence. This mode of control cannot be satisfied using a combinational logic system.

In addition to permitting 400-ns memory cycles without wait states, sequential machine design requires less random logic and board space than a flow-through design. The design process is simplified because the machines are implemented in easily changeable FPLS (fuse programmable logic sequencer) logic. Moreover, design changes can be readily documented and less time is needed for debugging and tracing events. Sequential circuitry is more easily simulated than random logic, in which all events must be sampled. And, since the CPU board's logic components run on the same clock, it is possible to debug them at faster or slower operating speeds.

When the CPU-board project started, this sequential machine approach had not been widely used in microcomputer design. Off-the-shelf hardware and adequate CAD tools were not available. This project shows that designing with commercial PALs and FPLS logic can reduce the chip count, as well as cost and development time.

The overall control logic of this linked sequential machine is divided into partitions. The events inside individual partitions are gov-

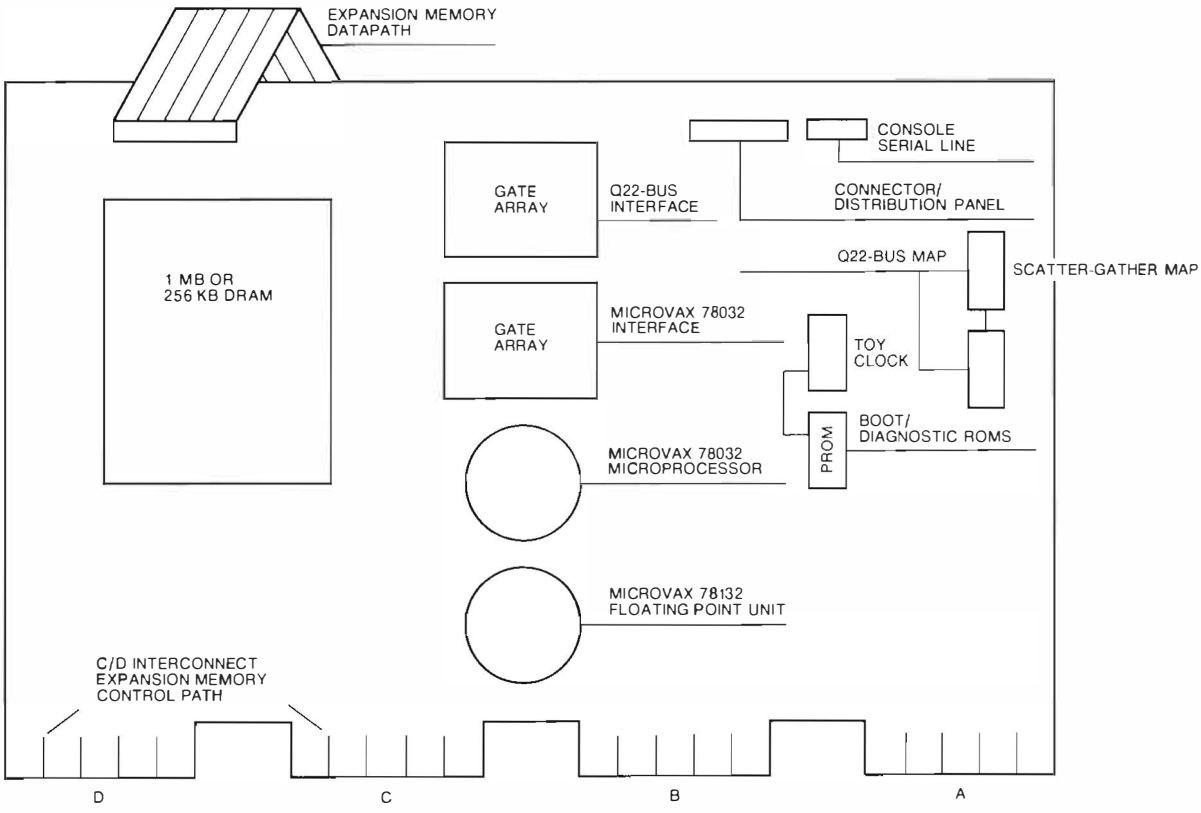


Figure 3 Functional Partitions of the CPU Board

erned by independent sequential machines, called controllers. The logic within a partition goes through a fixed, repetitive sequence of operations, or states, during the four quarters, or phases, of a microcycle. The operations of the various partitions are coordinated in two ways. First, all sequential machines run from the same clock so that their timing is based on the same stream of clock edges. Second, the sequential machines are constantly exchanging signals, providing each other with the protocol information needed for coordinating their flow sequences.

The sequential machines can be classified as modified Mealy machines.[2] The outputs are determined by the present input conditions and the present state of the machine. However, the state register is separated from the output register, with the AND programmable logic array fed by both the state register and the inputs to generate OR plane terms for the clocked SR latches. The advantage of clocked SR latches is that the past state need not be regenerated by every clock edge; only changes need activate an OR term. Using D-type latches would require that regeneration.

The block diagram in Figure 4 depicts a sequential machine representation of the CPU board's functional configuration in Figure 3. Under the on-board control partition at the left, the control function for the memory subsystem is distributed among three sequential devices: the memory sequencer, the memory arbiter, and the auxiliary device controller. Under Q22-bus control, there are also three sequential devices: the slave, arbitration, and master machines. These machines exchange request, acknowledge, and status signals to control operations.

Memory Subsystem

Our market research data suggested that the on-board memory should be either 256 kilobytes (KB) or 1 megabyte (MB). The amount depends on whether 64K DRAMs or 256K DRAMs are used. At the time the design was started, 256K parts were in short supply. Therefore, using 64K DRAMs was a strategy to counter that shortage.

The function of the memory controller is to carry out 400-ns read and write operations and to refresh its RAM chips. This controller con-

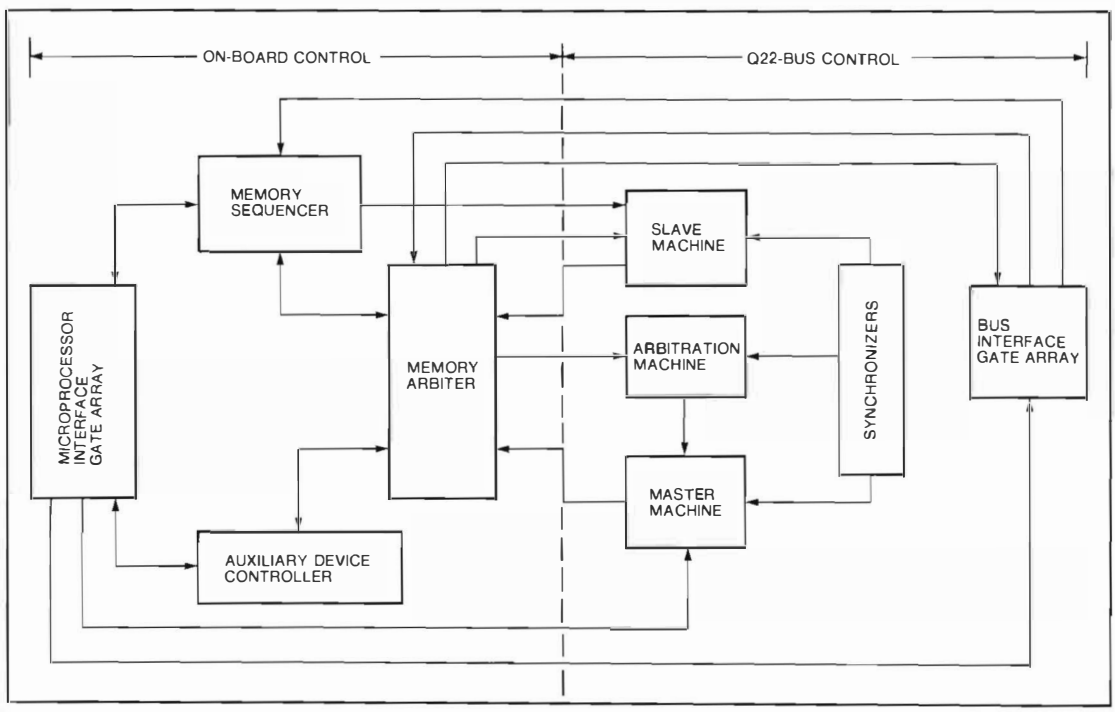


Figure 4 Block Diagram of the Control Architecture

tains a Q22-bus scatter-gather map that handles transfers between the Q22-bus virtual memory and on-board physical memory.

Memory access is controlled by the memory arbiter. This arbiter checks for outstanding memory access requests in a fixed-priority sequence at the ends of 200-ns idle cycles and 400-ns memory cycles. It also checks for requests from the Q22-bus slave machine, the memory-refresh counter, and the CPU chip, in that order. The fixed-priority sequence resolves collision requests for memory usage. If the arbiter requires exclusive control of the memory subsystem, a locking mechanism built into the subsystem prevents contention.

When the CPU chip requires a memory-read lock, the memory arbiter will stall the chip and direct the Q22-bus arbitration machine to suspend other bus activity. Those actions will happen only after any pending memory cycles of the slave machine have been completed. The arbitration machine will retain Q22-bus mastership until the write/unlock cycle of the CPU chip frees the bus. Until the arbitration machine becomes Q22-bus master and while the CPU chip is stalled, the memory arbiter will perform the demand-driven refresh cycles and resolve slave-deadlock cycles from the Q22-bus. As each memory cycle is completed, the memory arbiter checks these requests again, and either the Q22-bus or the refresh-memory cycle can begin at the next clock edge. If no Q22-bus or refresh requests are pending, the arbiter anticipates that a CPU-chip cycle will be next.

That anticipation and the fixed-priority sequence save a lot of program execution time. The CPU chip makes about seventy percent of all memory references. Slave machine accesses by the I/O bus devices occur twenty percent of the time (a maximum burst rate, not the average rate), and those by the refresh counter, two percent. (The remainder are idle cycles.) Therefore the controller, by anticipating that the CPU chip–rather than the I/O bus or the memory-refresh counter–will make the next memory access, allows a memory cycle of 400 ns, instead of 600 ns. (The 600-ns cycle would be necessary because the address strobe of the CPU chip would have to assert before the memory cycle could start, thus wasting one microcycle.)

When timing microcycles, the memory arbiter enables the memory sequencer at phases coincident with the CPU chip's entry to a new microcycle. This enabling happens even though the sequencer does not yet know whether or not there will actually be a memory access by the CPU chip. Not until three phases later can the sequencer determine whether or not the address strobe has been asserted for a memory reference. If so, the sequencer enables the continuation of the anticipated memory access. After that cycle completes, the next memory access will be enabled, and the procedure repeated. If not, the sequencer "kills" the cycle and runs another poll loop after checking for Q22-bus slave or refresh requests. Not anticipating a memory access would reduce performance by approximately thirty-three percent.

The memory sequencer generates the row and column address strobes, sets up reads and writes on each byte, and handles parity generation and detection. The auxiliary device controller can "stretch" the memory cycle of the CPU chip to synchronize its timing with slower devices, such as the TOY clock and the boot ROM.

The scatter-gather map converts between the 22-bit virtual addresses of the Q22-bus (4MB addressable) and the 24-bit physical addresses of the memory (up to 16MB addressable). As defined by VAX memory management, the 4MB is divided into 8192 pages of 512 bytes each. The 22-bit virtual address consists of a 13-bit page number and a 9-bit offset to the addressed byte in that page. The 24-bit physical address consists of a 15-bit page number and a 9-bit offset. An entry in the map for each 512-byte page and offset points to a location in physical memory. Each physical address has four byte masks that select which bytes are inactive on any memory reference.

There are, of course, other ways to map addresses between the I/O bus and memory. One way is one-to-one address translation, which in this case would have restricted physical memory to 4MB. Another way is first to map one-to-one into the lowest 4MB of memory. Then, the CPU chip can perform the translations and data transfers to the proper pages in the address space of the remaining memory. Unfortunately, this approach is unacceptable due to its effect on performance. A third way is to have fewer than 8192 mapped pages. In this case, programmers might have to provide their own mapping software for many real-time I/O

applications. That typically involves DMA access to large numbers of RAM locations. None of these methods proved as satisfactory as the use of the scatter-gather map.

Interface Control Signals

The interface control signals to the CPU chip include the following:

- Clock-in (40 MHz), clock-out (20 MHz; used to time the sequential machines), and reset signals
- Address, data, external-processor, and timing-strobes-out signals
- Three chip-status, four byte-mask, and the read/write signals
- DMA-request and DMA-grant signals
- Four interrupt-line signals and one HALT signal
- Ready and error signals

The pulse of the design is a four-state grey-code binary counter, which is clocked from the synchronous clock-out signal of the CPU chip. The first edge assertion of the clock-out signal after power-up puts the CPU chip in the first 50-ns phase of the four-phase microcycle. The grey code allows the memory arbiter and auxiliary device controller to track the state of the microcycles. The 28-bit address of the CPU chip is decoded to select the accessed device and then encoded into a series of 3-bit cycle codes. The auxiliary device controller, the memory arbiter, and the master machine decode those cycle codes to identify what type of timing cycles to sequence through. The two key signals, apart from the cycle codes, are those for the address strobe and the read/write. They direct the auxiliary device controller, the memory sequencer, and the master machine to perform the read or write operations with the device specified in the cycle codes.

Those three elements control the CPU chip's cycles and any system exceptions via the ready and error signals. The DMA request signal is used only during a reset operation to delay the CPU chip until the system has finished resetting. The byte-mask signals simply direct the control logic to perform certain operations. Those include masked (byte or word) or unmasked (longword) memory cycles and data funneling operations on the Q22-bus. (Data funneling converts 32-bit longwords to 16-bit words and vice-versa.) The unmasked cycles are required since the Q22-bus is 16 bits wide, whereas the memory and CPU-chip buses are 32 bits wide. The on-board I/O time can be extended to accommodate slower external devices. The memory controller allows the memory cycle to end only when a device has asserted a ready (RDY) signal, indicating the completion of its task.

Add-on Memory

The system's memory can be expanded with one or two memory boards, each containing either 1, 2, 4 or 8 megabytes. Thus total memory can be as large as 16MB and still offer a fixed 400-ns access time with no wait-states. Each board is linked to the CPU board by means of a local interconnect. This interconnect consists of special control signals on the C and D rows of the Q22 backplane and a 50-pin module-header and ribbon cable for data. Each interconnect links a board directly to the one just below it in the board cage of the system enclosure. Thus control signals and addresses can pass directly between the chips and memory without using the Q22-bus. The diagram in Figure 5 shows the functional organization of the memory boards.

For ease of installation and maintainability, the add-on memory boards are self-configurable; there are no user-settable switches or jumpers on the CPU board or memory boards. This design requires a logic function that combines active addresses with static configuration data to generate the proper control strobes according to the configuration. Therefore, although the add-on memory boards are position independent, they "recognize" which expansion slots they occupy. (To get the full 16MB configuration, the memory controller design supports 1MB-by-1 DRAM chips.)

On-board I/O Subsystem

The serial line interface in the on-board I/O subsystem provides the CPU board with a full-duplex, RS-423 EIA console terminal interface. The console interface program is implemented in macrocode in the boot ROM. The console-mode functions include general booting, user-computer interface, self-test and HALT. The boot ROM also includes special support func-

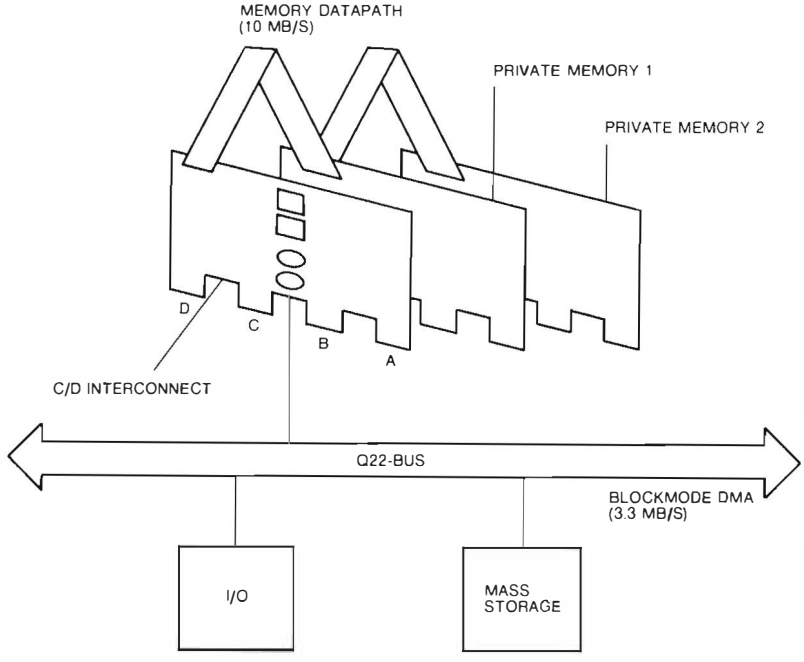


Figure 5 Functional Partitions of Memory Modules

tions for the software in the MicroVMS, ULTRIX and VAXELN systems.

As the boot ROM goes through a self-test sequence, programmable LEDs display the test status, identifying any board subsystem that contains a failure. By analyzing this sequence for effectiveness, we found that it provided a confidence level of eighty-six percent in the functional integrity of the CPU board and add-on memory boards. Although some Q22-bus logic functions could not be tested with this method, it helped to reduce significantly the times to do manufacturing and field service tests.

To emulate a CPU-halted condition, the CPU chip can be directed by either software or hardware switches to transfer program control to a firmware routine at a fixed PROM address. The HALT function retains the board state. The CPU chip traps to the boot ROM when there is a HALT, masking it until there is an instruction fetch outside the ROM. While in this emulated HALT, the firmware will perform the specified operations only after receiving either console commands or a signal from the AUTO-REBOOT switch.

The CPU chip does not have a RESET instruction; the chip simply sets a RESET request flag. The UNJAM command in the console mode initializes the bus by forcing the CPU chip to the DMA grant state. UNJAM then transfers control to RESET in the interface gate array of the CPU chip. After that, the logic resets the board's functions and the arbitration machine resets the Q22-bus. Any auxiliary processors are reset from the Q22-bus reset signal.

Exceptions, which may originate in the console, the on-board I/O, the Q22-bus, or the memory subsystem, are reported to the CPU chip for a machine check. This process involves setting an error-register flag in the interface gate array of the CPU chip. The chip then treats the exception as either fatal (HALT or AUTO-REBOOT) or non-fatal (abort the process).

Board Components

Logic hardware for the CPU board was selected by balancing the need for minimum power and board space against the use of low-cost, off-the-shelf components. The gate arrays for the CPU board and the bus interface, for instance, are more expensive than discrete logic; however, they are necessary to fit all support functions on one quad-sized board. Due to a conductivity connectivity limitation through the board's edge fingers, the maximum allowable power consumption is 45 watts for a 1MB on-board memory configuration. We were also con-

strained by the watts per square inch that had to be conducted from the board surface to the environment. That was important given that the enclosure is cooled by the flow of forced air.

The gate array for the CPU-chip interface decodes addresses and latches boot-ROM words. This gate array also contains registers for booting, diagnostics, and memory subsystem errors; the on-board I/O datapath; and the interrupt-acknowledge decode and control.

The gate array for the bus interface includes such components as the doorbell register, the memory-refresh counter, the holding latches for byte and word packing and unpacking, and timeout counters. This gate array also generates the bus addresses.

The memory subsystem includes a number of discrete components. The memory arbiter and auxiliary device controller are both commercial programmable sequencers. The memory sequencer consists of 12 discrete logic chips. However, we had to design our own memory controllers. The available commercial ones could not handle both the speed and the higher-level arbitration function required to anticipate memory accesses.

Previous board designs used an eight-layer construction technology (two power, four signal, two covers, and top and bottom solder masks). However, to reduce the board's cost, a six-layer technology had to be developed (two power, four signal, and top and bottom dry-film solder masks). Six-layer construction costs less than eight-layer due to alignment and drilling problems with the stacked layers of the latter. We used a CAD system to evaluate the chip interconnects on the board layout. The system showed that the signals could not be routed on two signal layers, but could on four. The two additional layers provide the 5V power and ground planes. Digital's Computer-Aided Design (CAD) Group in Maynard, Massachusetts, designed a custom software tool to help in developing the board layout. With this tool, it was possible to fit all functions on the board with 8-mil lines and spaces, and 60-mil pads. Having the lines and pads as wide as possible offers satisfactory yield in production and good signal quality due to strip-line characteristics.

Enclosures

Two enclosures were considered to house the boards, the BA23 and the BA123 boxes. At the time, the BA23 box was an active product; only minor modifications were needed to accommodate it to the MicroVAX II system, a nice, low-risk plan. In contrast, the BA123 box was still being developed. Using it represented a greater risk; however, it could support more mass storage. The backplane cages of either box could accept add-on memory and peripheral device interfaces on either quad-sized or dual-sized (5-¼ by 8-½ inch) boards. However, the BA123 box accepted more quad-sized and dual-sized boards. That was a distinct advantage because there would be different numbers of board slots in the board cages in different packages of the MicroVAX II system. Moreover, each enclosure had a different thermal environment that had to be considered in the layout of the CPU and memory boards. Based on these considerations, we chose to use both the BA23 and BA123 boxes as the enclosures for the boards.

CAD Tools

The tight schedule dictated that separate design teams had to develop each of the chips and the CPU board as parallel projects. These separate efforts were made possible by the extensive use of CAD tools and computer simulation. Simulation was used extensively to design the CPU and FPU chips, the on-board memory and I/O subsystems, the gate arrays, the sequential machine controllers, and the Q22-bus. A board-development tool set was selected from CAD packages available in the industry. Since these packages were generally incompatible, we developed a process that transported wire lists between these various CAD tools. The process linked inputs and outputs between the schematic-capture work stations, the PC-board layout system, the simulator, the gate-array vendor, and the documentation control group. One key to the rapid development of schematics was to let the designers retain control by performing their own drawings and edits.

We planned to use gate arrays right from the start of the project. Therefore, a hierarchical schematic-capture system was needed to facilitate the representation of devices at a number of levels. To verify the schematics, we selected a mixed-mode logic simulator that had library support for most of the off-the-shelf devices used in PC-board design. That minimized the development time to construct the simulation libraries. A complete simulation model of the CPU board was also constructed to expedite the

design verification process. This model provided a "soft" test bed for design changes before they were committed to hardware. Behavioral models were used to simulate the signals from the CPU chip, as well as any device attached to the Q22-bus. No attempt was made to emulate the VAX instruction set. Instead, the goal was to verify the sequences for reads, writes, interrupt acknowledgements, and the cycle flows for the block and non-block modes of the Q22-bus.

Several CAD packages developed by Digital were also employed to expedite the board design process. Figure 6 shows the CAD flow process that was assembled. (For more details on the CAD tool suite, see reference 3.)

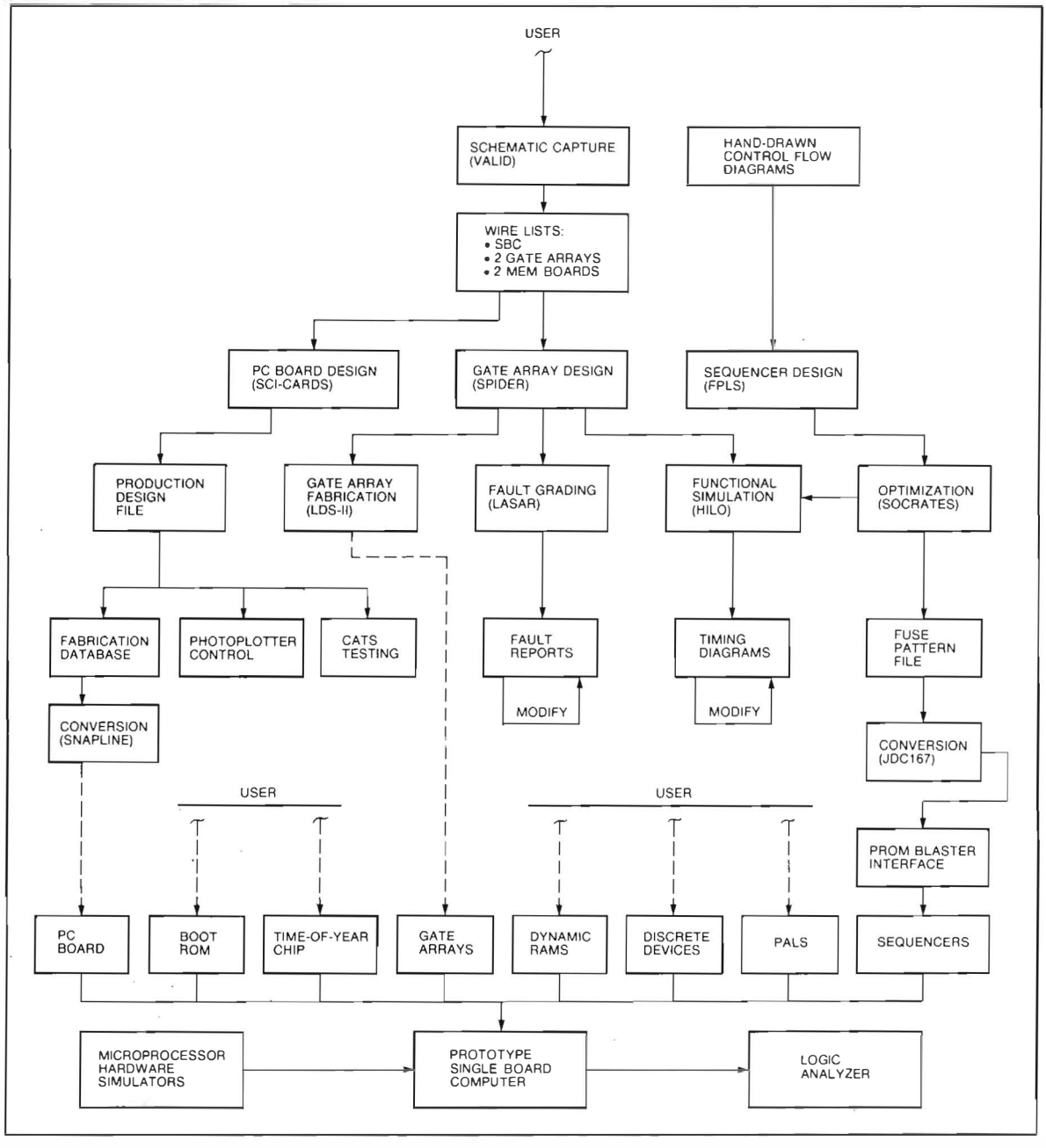


Figure 6 CAD Tools Used in the CPU Board Development Process

Two CAD tools were used to help in the decision process for selecting reliable components. The CPU board was modeled with the reliability prediction program PREDIC, which is based on MIL STD 217. PREDIC utilizes component thermal data from the second tool, the THUDS analysis program. Using these tools helped us to avoid the creation of hot spots on the board layout and the use of low-reliability components.

These CAD tools were so successful that the CPU board was ready by the time the first-pass CPU and FPU chips were ready. It then took only two weeks of debug to go from the functional chips to running the MicroVMS operating system. In all, the development of the CPU board took less than one year from initial specification to operational prototypes.

Summary

The CPU board was designed as part of a larger project with formidable time constraints. Such an environment demanded that the design of any one component rely on the proposed specifications for other, interlocking components, rather than on actual pieces of developed hardware. That environment required a cooperative team spirit that was goal oriented and fostered the assumption of rational risks. Both intergroup and intra-group communication became extremely important. The achievement of these factors was largely responsible for the success of the MicroVAX II project.

Especially important was the fact that communication was aided by the CAD tool suite used to support the overall project. In the case of the MicroVAX II system, we started from a well organized datapath and employed sequential machine architectures for controlling it. In that way, the design documentation, simulation, verification, and support were all made more manageable. In future projects these tool suites will mature and behavioral component models will begin to serve as design specifications. The ability to solidify the design early in a project means that board designers can fashion silicon systems on boards that are functional on the first pass.

References

1. D.W. Dobberpuhl et al, "The MicroVAX 78032 Chip, A 32-Bit Microprocessor," *Digital Technical Journal* (March 1986, this issue): 12-23.

2. W.I. Fletcher, *An Engineering Approach to Digital Design* (Englewood Cliffs: Prentice-Hall, Inc., 1980).

3. A.F. Hutchings, "The Evolution of the Custom CAD Suite Used on the MicroVAX II System," *Digital Technical Journal* (March 1986, this issue): 48-55.

Anthony F. Hutchings

The Evolution of the Custom CAD Suite Used on the MicroVAX II System

The MicroVAX II chips were designed in only 20 months, due in part to simulation on CAD systems. Digital has a long history of using CAD. Much of the MicroVAX II's CAD suite evolved from tools used on an earlier VLSI VAX design. The higher-level chip functions were debugged using behavioral simulation, after which the circuits were modeled using the reliable SPICE and GRAPES systems. The IV system verified all interconnects and extracted wirelists, while other tools controlled the databases and checked design rules. The next generation of CAD tools must deal with a three-fold increase in chip complexity.

The factors that must be considered when initiating and committing to a new VLSI design are quite complex. They are related in the following way:

Market Requirements/Chip Definition
+
Technology Status
+
CAD Status
+
Engineering Talent Available

Products with long lead-times can accept higher risks in the process chosen for chip fabrication and CAD technology. However, products with short lead-times, such as the MicroVAX 78032 chip, can tolerate virtually no risk in this domain.

One way to reduce these risks is to test the chip designs by simulating their performance before fabrication; another way is to check for all possible, known fabrication process violations before submitting the mask data for manufacture. CAD systems and tools have been developed for this purpose: to discover problems so they can be corrected at minimal cost, both in time and resources. Digital Equipment Corporation was an early user of CAD to decrease the time-to-market for its VLSI products.

The MicroVAX II project needed to rely on a stable CAD system and set of tools while designing the 78032 CPU chip (and its companion floating point coprocessor, the 78132 FPU chip). Much of the stability of the CAD system was derived from work done to develop a multichip set for another VAX microprocessor.[1] We were able to both rationalize and simplify the results of this pioneering effort to suit the needs of the MicroVAX project. Let's begin by discussing this earlier CAD system to see how its use affected decisions made on the 78032 and 78132 projects.

CAD System for Earlier VLSI VAX Design

In many ways, the design process for the earlier VLSI VAX microcomputer set the tone for all subsequent VLSI designs at Digital Equipment Corporation. This process was characterized by the extensive use of simulation, especially high level, or behavioral, simulation. The commitment to high-level simulation was particularly innovative at that time.

Two types of simulation models were used for this earlier microcomputer. The first type was designed as a high-level software breadboard used to develop and check out the

microcode before the chip hardware was available. The second type was developed as a relatively detailed register transfer level (RTL) model of the actual physical partitions and design concepts of the chips themselves. This model was used directly by the logic and circuit designers to develop the switch and circuit-level representations of the design.

One problem with using two models is that the output test vectors have to be checked continually to ensure compatibility between the microcode and chip designs. Thus, although each was optimized to a specific task, the models proved to be somewhat cumbersome to use.

The hub, or kernel, of the data management system was called CHAS.[2,3] This proprietary system was developed at Digital's semiconductor facility in Hudson, Massachusetts, expressly to form the nucleus of an integrated MOS custom-design suite. The CHAS system performs the necessary data management functions on chip design databases and was originally intended to control all the design activities of a chip project. The system embodies many of the "structured top-down design" principles of Carver Mead.[4]

The CHAS system manages the data collected from circuit and logic simulations, layout designs and syntheses, layout verifications, and schematics entry. This central system also provides data protection and conversion functions, as well as generating simulation wire lists.

Decisions Derived from the Earlier Project

From the outset, the CPU and FPU design teams made a number of important decisions based on the experience gained from the earlier project. One driving factor in these decisions was the short time-to-market, which dictated that simplifying the design process was a primary goal.

- The first decision was that there would be only one behavioral, or functional, high-level simulation model of the chip rather than the two used earlier. Thus the functional model was more complicated than the earlier one, but avoided the very time-consuming task of checking the output test vectors. Using one model guaranteed that the microcode development would be in step with the chip design, since both teams had to use the same model.
- The next decision was to carefully control the evolution of the CAD system that was used. Any experimentation with enhancements to existing CAD tools or with brand-new CAD tools would be done only in a controlled environment. One project engineer, trained in software and with CAD experience, was to be responsible for re-verifying the new functionality and "robustness" of all new CAD releases. This approach enabled the team to acquire a vastly superior design rule checker (DRC), which considerably enhanced productivity during the physical design phase of the project.

  This approach also differed greatly from that of the earlier project, although the lessons learned from that project considerably shaped the team's attitudes. For example, the earlier project suffered–for a while–from attempting to use a first-generation layout editor that had too many bugs. (This tool was not in fact used on any part of the final design.) It also experimented with early versions of the CHAS system. These versions did not perform as well as desired for some functions (e.g., the Assembled Block Wirelister). In contrast, the MicroVAX design teams decided to perform all layout on the industry-standard CALMA GDSII layout system, a robust and proven tool.
- The third decision involved the data management of the design database. Rather than use all the features of the CHAS system, we decided to manipulate the design data using the simpler VMS file-management system with its loose but adequate version-control mechanisms. The CHAS system was used, but in the role of tool integrator, linking, for example, the QUICKDRAW schematic editor to the SPICE circuit simulator.[5] The CHAS system also provided a variety of valuable format conversion utilities.
- The final decision was to use one proven tool for interconnection verification. This layout extraction/ verification tool, called IV, performed all the electrical connectivity checking in a very efficient manner.[6] The earlier project had used a combination of bought-out tools and although that verification was very thorough, it was more costly than the single-tool process (IV) used on the MicroVAX project.

The Design Methodology and CAD Tool System

Having made these simplifications, the design team established a fixed definition of their design methodology and CAD tool mapping. This definition was followed faithfully throughout the life of the project.

Figure 1 shows all the activities in the design phase that were supported by CAD tools. The middle column lists each activity; the left column shows the type of data used in this activity and manipulated by the CAD tools which are shown in the right-hand column alongside the actual activity and data they support/use.

The arrows indicate iteration paths where feedback is sent to a higher level. That is, where results are obtained from a checking or verification activity, it may be necessary to go back and modify an earlier set of assumptions and design decisions. For example, in running the DRC, it is highly likely that we will find design rule violations that require us to correct the physical chip layout.

A number of very important paradigms should be noted.

1. The behavioral model of the design was kept current with the logic design of the chip to guarantee the accuracy of the microcode with the chip design.
2. The critical hurdle for the functional correctness of the design was the correct execution of a certain number of VAX macroinstructions under an automated checking process. (The tool used for this process was called AXE, an architectural test-case generator and execution tool, working in conjunction with the DECSIM system, Digital's proprietary multi-level, mixed-mode simulation system). The minimum number of cases was 100,000 tests for each VAX instruction group. In all, more than 1 million tests were executed before the chip was fabricated.
3. The number of iterations during the layout-design phase was minimized.

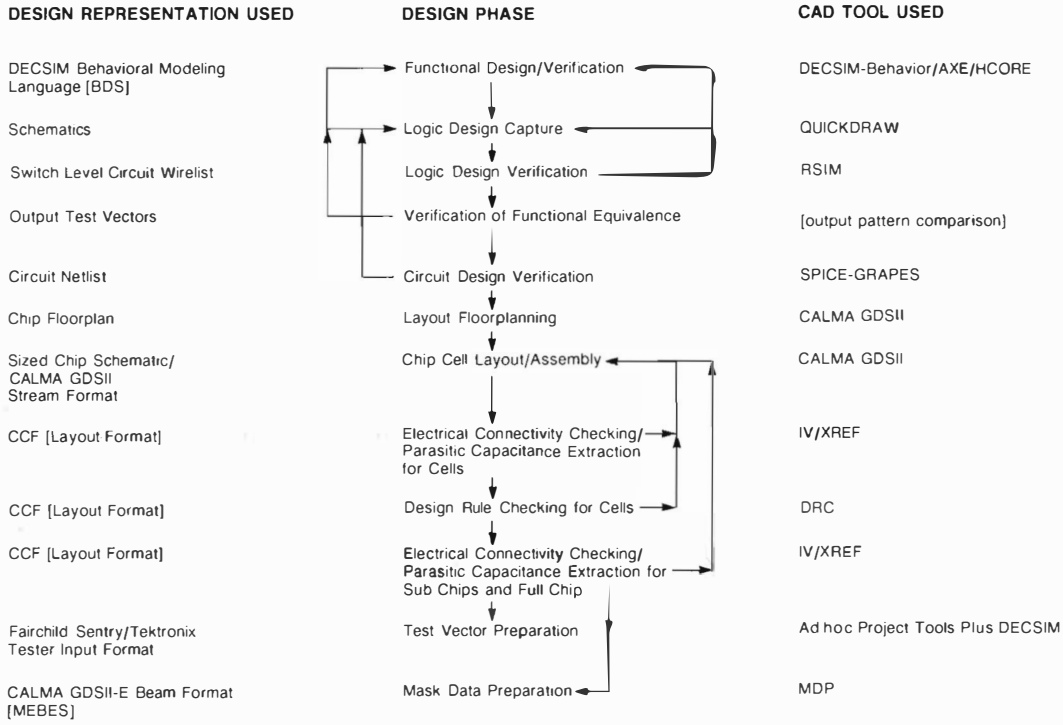


Figure 1 CAD Tools Used in the Design Phase

Changes during this phase are very expensive and the number was kept small by having the design team submit only sized schematics (i.e., ones with transistor width and length specifications that were verified using the logic and circuit simulators) to the layout design team.

4. The mask data was not submitted to the mask shop (or even generated) until all sections on the whole chip were free of design-rule and electrical-connectivity errors.

The Value of the NMOS CAD Suite on the MicroVAX II Project

Figure 2 illustrates the entire CAD suite used on the 78032 and 78132 chip designs.

Use of the CHAS System

As mentioned earlier, the final use of the CHAS system was pared down considerably by the MicroVAX project as compared with its use in the earlier project. The functions used most frequently were

- Schematic wirelisting
- Layout format conversion
- Copying files out of the CHAS database
- Plotting
- Invoking the SPICE circuit simulator and the GRAPES graphical post-processor

Behavioral Modeling and Simulation

A simulation system called DECSIM was used to simulate the behavioral definition of the chip design.[3,7,8] The DECSIM system works interactively and was used to debug the high-level functional design. This system is very reliable and proved to be a vital ingredient in achieving the high degree of accuracy of the microcode.

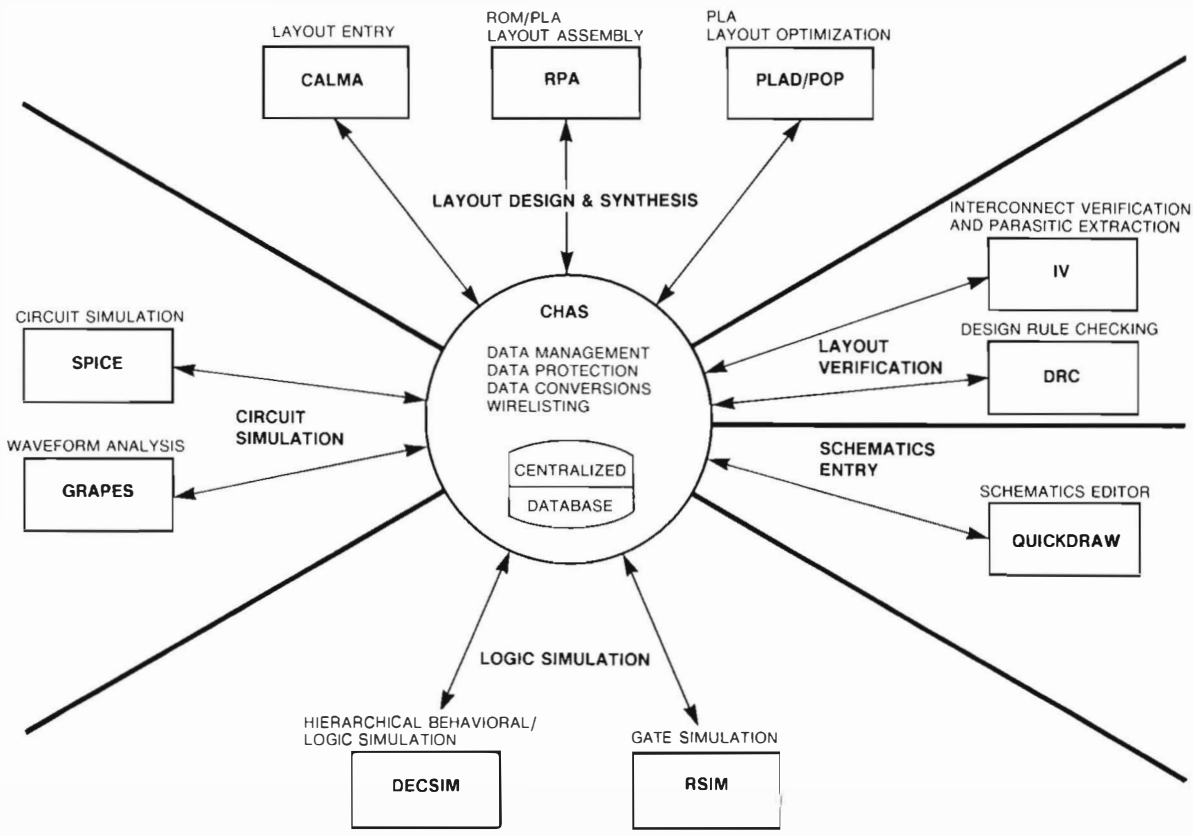


Figure 2 CAD Suite Used on the MicroVAX II VLSI Design

Schematic Capture

A drawing system called QUICKDRAW was used as a schematic editor. QUICKDRAW's greatest assets were its architectural simplicity, reliability, and ease of use. The system permitted schematic entry on low-performance graphics terminals (VT125s). Of course, keyboard entry is not always totally practical for bulk schematics entry, or even good for small schematic changes. However, QUICKDRAW could be accessed from any terminal, was easy to learn in a few hours, and could be used by the whole chip team.

Logic Simulation

As in the earlier project, the MicroVAX team decided that they needed the accuracy of switch-level logic simulation. At this level of representation, the transistors are literally treated as "switches," but with resistance and capacitance attributes. The models can also represent both bidirectionality and charge sharing. At the time, the MOS (switch-level) capability of the DECSIM software was still maturing; therefore, the team decided to use a switch-level simulator called RSIM, developed at the Massachusetts Institute of Technology. RSIM was sufficiently accurate to enable the complete design to be simulated at this level, although its timing aspects could not be used. RSIM's usage, therefore, resembled that of a logic simulation system. The prime role of this stage of the process was to prove equivalence with the higher-level behavioral model, thus gaining functional completeness at a lower, more accurate level of representation. That equivalence was achieved by supplying the same test vectors used in the behavioral phase to the RSIM runs.

Circuit Simulation

An industry-standard system, SPICE, was used for circuit simulation. SPICE was the most accurate mechanism of its kind available for simulating the electrical performance of circuits on the chips. This simulator was used extensively for circuits containing up to 1000 transistors. There were two major advantages of Digital's version of the SPICE system.

1. The device models encoded into SPICE were a very accurate representation of the devices made in Digital's NMOS process. The device equations built into these models were derived in two ways: first, by extracting the operating characteristics of NMOS devices from fabricated test chips; and second, from the results of experiments performed by another team at Digital. That team created models for devices and processes by using a battery of sophisticated simulators, such as MINIMOS, SUPREM, and SEDAN.
2. Throughout the pre- and post-processing stages, all voltage values over time from a SPICE run could be saved and later graphically analyzed by the designers in a proprietary graphical post-processing system called GRAPES. Using this system avoided having to make multiple runs of SPICE and permitted much easier interpretation of the output waveforms. Figure 3 is a sample circuit simulation waveform from the GRAPES system.

Interconnect Verification and Wirelist Extraction

The IV system, partially proven on previous chip design projects, was a major boon to this design team. The system performed several functions.

1. It extracted a wirelist (in SPICE format) from the actual layout database.
2. It calculated the parasitic capacitances for devices and nodes and fed those into the extracted wirelist. That automatic input permitted the final simulations in SPICE to be very accurate.
3. It detected any open and short circuits in the electrical network of the wirelist.
4. It compared the extracted wirelist with the original wirelist (created via the schematics editor, QUICKDRAW) and reported any mismatches in signal or node names, device sizes, and other elements.

This verification and extraction tool performed all these functions much faster and more accurately than any of the connectivity checkers or extractors that were available commercially. The IV system is generally recognized as one of the best in the industry for this purpose.

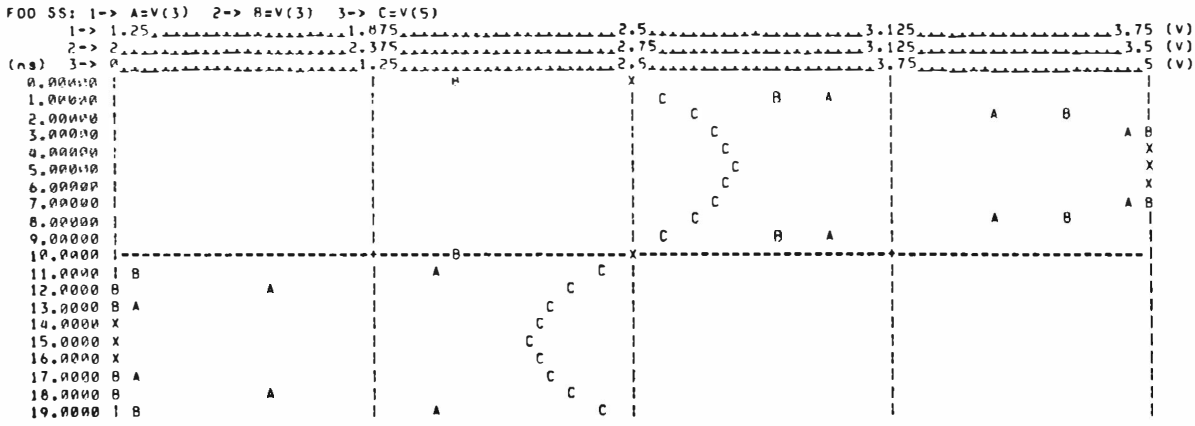


Figure 3 Sample Output from the GRAPES System

The system has some unique data structures and algorithms.

1. It simplifies circuit extraction by converting all shapes into trapezoids. These are very convenient representations that permit IV to thoroughly analyze lateral-node and vertical-device connections.
2. It calculates the parasitic capacitances for both area and periphery, taking into account cell-capacitance effects coming from ever-shrinking device geometries. The system also calculates coupling capacitances.
3. It performs very fast wirelist comparisons (layout to logical), using a unique graph-isomorphism algorithm that isolates errors rather than propagating them.

System Verification

The final system-level verification of the MicroVAX chips was performed using the AXE test-case generator in conjunction with the DECSIM behavioral models. In this way, test cases (which were in fact VAX macroinstructions generated by AXE) were passed to the simulation model for execution. The execution results were then compared automatically with those obtained from running the same test cases on an operational VAX system. The MicroVAX team used AXE in a particularly novel way. Via Digital's Ethernet network, they searched for in-house VAX-11/780 systems with spare capacity on non-prime shifts. Then the team activated AXE on those systems, which generated a tremendous number of test cases. This same approach was used (and continues to be used today on subsequent projects) for running CPU-intensive SPICE circuit simulations on many processors in remote locations.

VLSI CAD Beyond the MicroVAX II Project

Digital's use of the NMOS VLSI CAD suite reached a peak of maturity with the 78032 and 78132 projects. We have been able to make a major process-technology step to CMOS1 with little cost by exploiting the same basic set of tools. That has enabled us to develop a whole new set of VLSI chip products in very quick succession.

However, following Moore's Law, it is time to face the challenge of a two- to threefold increase in complexity for the next generation of chip designs. This complexity means that design teams for new custom chips must be able to design parts with twice the transistor count as the 78032, yet take the same or less time to do it. Figure 4 illustrates the complexity that will be experienced in future chip design projects.

Major productivity improvements in CAD systems must be made to accomplish this doubling of the transistor count. Digital's VLSI CAD Group is now making the following improvements in its custom tool suite:

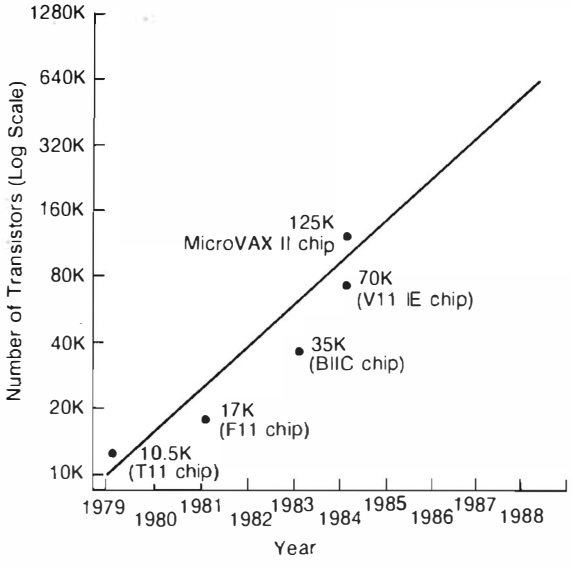


Figure 4 Chip Complexity Projections

- A new system for tool integration and data base management, called KATIE, is being developed to replace the CHAS system. The KATIE system has a simpler, more modular CAD kernel than has the CHAS system, but with much higher performance.
- The DECSIM software is being improved to provide true mixed-mode modeling and simulation (behavioral-gate-switch). Initial results indicate a doubling of simulation productivity, and our aim is to gain equivalent performance in the separate switch and behavioral areas.
- A variety of techniques is now providing up to ten times the performance of the traditional SPICE system for circuit simulation. For example:
  1. An event-driven circuit simulation system called SAMSON,[9] which exploits the temporal sparseness of digital networks, has been developed. SAMSON offers from five to fifty times the performance of SPICE for direct current and transient analyses.
  2. SPICE can be made to run much faster on vector processors and multiprocessors.
  3. A timing verification system called TV can analyze critical paths at the rate of 1000 transistors per minute of CPU time.[10] TV performs within fifteen percent of the accuracy of SPICE, but its speed is several orders of magnitude faster.
- Schematic entry can be improved by running QUICKDRAW on high-performance, high-resolution graphics workstations. The system will support multiwindowing, menus, and pointing devices, as well as provide high-performance wirelisting, with at least a doubling of speed over the version used on the 78032 chip design.
- High-resolution, VAX-based graphics workstations will also be used for custom layout editing, using the in-house developed editor, MEGAN.

Summary and Conclusions

The MicroVAX II project demonstrated a number of valuable lessons about CAD in general and VLSI CAD in particular.

1. The second and subsequent projects that use a particular CAD technology benefit enormously from the experience gained during the first use.
2. As a corollary to the point above, it is imperative that CAD tools and systems be built to endure at least two generations of projects. Otherwise, the cost and difficulties of using these tools will far outweigh the benefits.
3. The CAD teams should use the period of stability during these later uses of the tools to develop the next generation of more powerful tools.
4. Much conservatism exists in the IC industry around the need to archive complete images of all tools (layered products, operating systems, etc.) used in the design of an IC, along with its final mask database. Future chip teams plan to migrate their mask databases to contemporary CAD systems. This process will use the same exhaustive checks and tools used on the original design to ensure that the conversion is thorough. In this way, there will be no need to revert to old copies of outdated systems and tools when making engineering change orders late in the product's life cycle.

5. The close coupling between chip design teams and CAD developers is an invaluable ingredient in the successful completion of chip projects.

References

1. W.N. Johnson, "A VLSI Superminicomputer CPU," *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (1984): 174-175.

2. J.C. Mudge, C. Peters, and G.M. Tarolli, "A VLSI Chip Assembler," in *Design Methodologies for VLSI Circuits*, ed. P.G. Jespers (Rockville: Sijthoff and Noordhoff, 1982), 329-356.

3. A.F. Hutchings, R.J. Bonneau, and W.M. Fisher, "Integrated VLSI CAD Systems At Digital Equipment Corporation," *Proceedings of the 22nd ACM/IEEE Design Automation Conference* (1985): 543-548.

4. C. Mead and L. Conway, *Introduction To VLSI Systems* (Reading: Addison-Wesley, 1980).

5. SPICE was developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

6. W.J. Herman and G.M. Tarolli, "Hierarchical Circuit Extraction With Detailed Parasitic Capacitance," *ACM IEEE 20th Design Automation Conference Proceedings* (1983): 337-345.

7. M.A. Kearney, "DECSIM: A Multi-level Simulation System For Digital Design," *Proceedings of the ICCD Conference on Computer Design* (1984): 206-209.

8. R.R. Rezac and L.T. Smith, "Methodology for and Results from the Use of a Hardware Logic Simulation Engine," *Proceedings of the ICCD Conference on Computer Design* (1984): 457-461.

9. K.A. Sakallah and S.W. Director, "SAMSON: An Event Driven VLSI Circuit Simulator," *Proceedings of the Custom Integrated Circuits Conference* (1984): 226-231.

10. N.P. Jouppi, "TV: An NMOS Timing Verifier," (Thesis, Stanford University, 1982).

Other References

Panel Discussion, R.J. Camoin, Moderator, "Central DA and its Role: An Executive View," *ACM IEEE 20th Design Automation Conference Proceedings* (1983): 3-11.

R.H. Katz, "Managing the Chip Design Database," *IEEE Computer*, vol. 16, no. 12 (December 1983): 26-35.

W.M. vanCleemput and H. Ofek, "Design Automation for Systems," *IEEE Computer*, vol. 17, no. 10 (October 1984): 114-122.

J.C. Foster, "A Unified CAD System for Electronic Design," *ACM IEEE 21st Design Automation Conference Proceedings* (1984): 365-369.

K. Sherhart, M. Vershel, and J. Owen, "The Engineering Design Environment," *ACM IEEE 21st Design Automation Conference Proceedings* (1984): 466-472.

B.W. Lampson, "Hints for Computer System Design," *IEEE Software*, vol. 1, no. 1 (January 1984): 11-28.

Rick Spitz
Peter George
Stephen Zalewski

The Making of a MicroVAX Workstation

Developing a MicroVAX workstation required that graphics hardware and software be designed. The project team kept the hardware simple by using VAX instructions for most of the work. Extensive graphics software bridges the hardware and the graphics applications. The graphics and windowing software, UIS, is the key to that process. UIS supports transparent multitasking with a distributed method for managing regions on the screen. A video device driver manages lists of region descriptors, keeping track of keyboard and mouse changes. The UIS system normally executes in user mode, thus minimizing overhead and utilizing the full performance of the VMS system.

When Digital decided to develop the MicroVAX series, we also began to consider how to build them into a family of low-cost VAX engineering workstations. Experience with the VAXstation 100 provided us with a great deal of knowledge related to workstation requirements. However, its architecture required extensive graphics hardware. This architectural approach was not considered viable for a low-cost, high-volume engineering workstation intended for a single user. Another approach placing greater emphasis on software was illustrated by Xerox's Star workstations, which were in use within Digital.

We decided that combining the MicroVAX processor with a low-cost graphics controller, the VMS operating system, and a good human interface would result in a powerful workstation. The VAX/VMS environment already allowed any VMS application program to run on every member of the VAX family. The MicroVAX system would extend the family to include lower-cost VAX systems. A MicroVAX workstation, in addition to running all existing VMS software, would now provide a base for graphics applications.

In the spring of 1983, a joint task force of hardware and software engineers was formed to determine how this workstation should be built. Our strategy was to design a product based on the MicroVAX I system and evolve it to a mature workstation using the MicroVAX II system.

The task force's objective was to set the overall goals of the project and to make sure that the graphics hardware and software were well integrated. Guided by a strong focus on time to market, the workstation hardware group had the responsibility of building an initial graphics controller. They were also chartered to initiate design work on future hardware graphics controllers with more features and higher performance. The VMS software group took on the role of developing the software components. This paper is written by members of the VMS Development Group; therefore, its primary emphasis is on the software aspects of this project.

Our first task was to make sure that the graphics hardware being defined was suitable for efficient use by the software. Having limited experience with low-cost graphics controllers and workstations, we proposed a strategy of using a very basic Q-bus controller and doing most of the work with VAX instructions. This approach was viable because the VAX instruction set is rich and versatile in the area of character and bit manipulation. It also minimized the risk in developing hardware and provided maximum flexibility for the graphics capabilities. With greater freedom in the software design, we could gain experience and provide better direc-

tion for hardware features needed in future graphics controllers.

Since no MicroVAX CPU had yet been developed, we built a breadboard hardware configuration to do hardware and software evaluations. MicroVAX systems execute a subset of the full VAX instruction set in hardware; however, software emulation of the other instructions allows all VAX software to run transparently. For cost and space reasons, MicroVAX systems were targeted to use the Q-bus for I/O, while most existing VAX systems used the UNIBUS for most peripherals.

The breadboard configuration consisted of a VAX-11/750 system with a UNIBUS-to-Q-bus adapter. We obtained some experimental Q-bus graphics controllers used in the development of the graphics interface for the PRO350 hardware. Using this configuration, we evaluated the performance of text and graphics by implementing a number of software algorithms.[1] This technique treated display memory as standard VAX program memory, and VAX character and bit instructions were used to generate text and graphics. Evaluation of our results showed that this approach was reasonable and the basic performance was acceptable; however, some assists were still needed in hardware.

The VCB01 Hardware Graphics Controller

Taking our results back to the the joint task force, we settled, after several iterations, on a hardware design. The hardware graphics controller was named the VCB01, known internally as the Q-bus video subsystem, or QVSS. Due to space and power constraints in MicroVAX packages, the controller had to fit on a single-quad Q-bus module. It contained 256K of bitmap memory that was fully addressable by any VAX instruction. That amount of memory was more than was needed to fill a full-screen video monitor. The extra memory would allow software graphics routines to operate directly on occluded areas of windows in the video display memory.

Based on inputs from the software evaluation, the hardware would also contain a scan-line map to allow mapping any scan line in display memory onto the physical screen. This technique allows much better scrolling performance, facilitates the management of occluded window areas, and allows the simultaneous support of different windowing systems. A 16 × 16-pixel cursor plane, a separate hardware component, greatly simplified the software logic required to manage the mouse cursor. The pattern is programmable to allow dynamic changes to the cursor pattern, depending on its screen location and the state of the workstation. In addition, a mouse interface and dual UART are provided to connect to a mouse, a keyboard and an optional tablet. The inherent simplicity of the hardware allowed the hardware team to produce the first prototype by the early summer of 1983.

Figure 1 shows a block diagram of the VCB01 configuration.

Software Architecture

The software team was chartered to develop a general software workstation architecture. Our goal was to allow the evolution of future MicroVAX workstations that would address

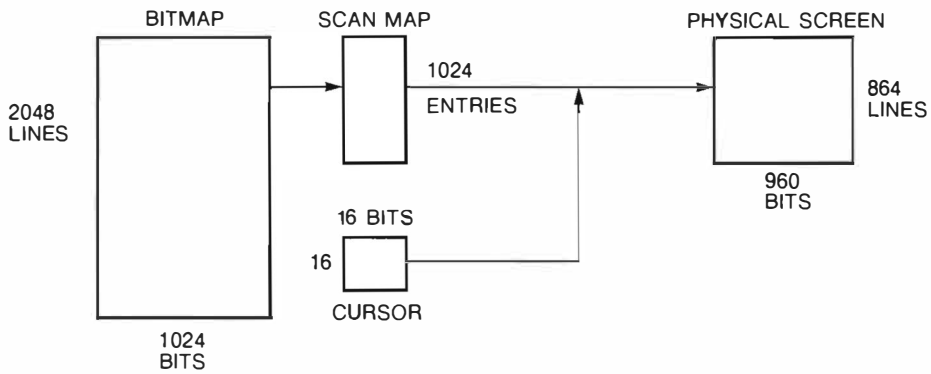


Figure 1 Block Diagram of the VCB01

cost-sensitive markets with basic, inexpensive hardware. We also wanted to improve performance and take advantage of features to be provided by more-intelligent hardware graphics controllers in the future.

Our performance evaluation of the VAXstation 100 architecture pointed out that the central dispatcher needed to manage the windowing activities on the physical screen was a real bottleneck. Therefore, we elected to pursue an approach that used a distributed method to manage regions on the physical screen. In most cases this approach would allow an individual job, called a process in the VMS system, to operate directly on bitmap memory. There is much less overhead than context switching between processes, as required in a centralized screen-manager design.

The software architecture that we defined was implemented by a loadable set of VMS system services know as the User Interface Services, or UIS.[2] UIS provides fundamental graphics services and display list capabilities. Application programs, high-level graphics packages, and VMS's VT100 and TEK4014 emulation drivers all utilize UIS to construct individual windows, as well as for text and graphics functions.[3] A VCB01 device driver is used to manage the physical hardware.[4] The driver is responsible for controlling the keyboard, the mouse (pointer), and the scan-line map.

VCB01 Video Device Driver

The video device-driver software has one primary function: to manage lists of region descriptors. In particular, it keeps three main lists; one each for keyboard input, button transitions, and pointer (mouse) movement.

To be notified about a particular event, an application program posts a request to the driver. The request specifies the type of event desired and the region on the screen. The driver then places this request on the appropriate list. For example, if pointer movement requests are active and mouse movement occurs, the driver will search the list for the entry that has specified a region that the pointer is currently within. The driver then notifies the application that was the last one to specify this area. The notification mechanism used is a software interrupt, known in the VMS system as an asynchronous system trap. This trap interrupts the flow of the specified user process and invokes a user-defined action routine. This technique provides a low-cost, responsive notification to the application.

The keyboard is connected to the device driver by a dual UART on the video controller. A hardware interrupt is delivered to the driver each time a key is pressed. The driver then searches the keyboard list and delivers the character to the process associated with the top entry on the list. All keys are "soft," which means that any key on the main keypad can be defined as any of the possible ASCII character codes. It is also possible to define multicharacter sequences for a given key. The second half of the dual UART is used to support a bit tablet or a serial mouse. These devices need to send several bytes of data for each pointer or button transition. The driver buffers this data until it receives enough to decode an event. Then it searches the appropriate event list and, if necessary, delivers a software interrupt to the application.

The driver supports the capability to specify cursor patterns for a region. When cursor movement is detected, the driver searches a list to determine what the cursor pattern should be for the current location of the pointing device. Once located, the pattern is loaded into the hardware. The video controller hardware then superimposes the pattern onto the appropriate screen area by merging the pattern with the video signal from the bitmap memory. This procedure eliminates the need for a save-and-restore operation in the physical bitmap each time the cursor moves or a write to bitmap memory occurs. The hardware also has the ability to specify two logical operations, NAND and XOR, on the cursor pattern. This ability prevents a white cursor from being lost on a white screen, or a black cursor on a black screen. The driver tests the physical bitmap location that is overlaid by the cursor to determine which logical operation should be used to maximize the cursor's visibility.

A proportional-acceleration movement algorithm is used to minimize the desktop area required for a mouse pointer. The driver accelerates the cursor's movement if the mouse's rate of movement exceeds any of a series of thresholds in a given screen refresh interval. If no acceleration were to occur, it would take a desktop space of approximately 13 by 11 inches to move the mouse both horizontally

and vertically respectively across the screen. With acceleration, a mouse movement of only 2 inches is needed to move across. The acceleration values used are as follows: 1 to 2 pixels of linear mouse movement per screen refresh interval, no acceleration needed; 3 to 4 pixels, accelerate by a factor of 2; 5 to 8 pixels, accelerate by a factor of 4; greater than 8 pixels, accelerate by a factor of 6.

The driver provides an optional console window to allow system-level debugging. The MicroVAX CPU can communicate directly with the video controller during booting and debugging. If this feature is enabled, the top 240 scan lines of video memory will be allocated for the console window. When the CPU wants to communicate with the console, the VMS console driver will map directly to those 240 scan lines. Thus, the console driver emulates a "dumb" terminal in this region. When a function key is pressed on the keyboard, the video driver will map this special console memory onto the top 240 entries of the physical scan-line map, and the operator console will appear. When the key toggles again, the top 240 entries of the scan-line map will be restored.

UIS Graphics and Windowing Software

The decision to use simple hardware meant that software had to be developed to bridge the gap between that hardware and the applications. This software was of critical importance because the hardware designers assumed that a software layer would be needed to support even the most basic graphics functions.

Early in the design process, we decided that this software would provide more than just basic I/O support through the video controller. Like the VMS operating system it was built on, the workstation graphics and windowing software, UIS, would support transparent multitasking. That meant being able to handle simultaneous demands by multiple independent applications on the shared VCB01 hardware resources. Therefore, UIS should be designed to provide two capabilities. First, it should have a library of general-purpose procedures that applications could use to easily access the hardware resources. Second, UIS should contain transparent management and synchronization mechanisms. In that way, independent applications could share both screen space and the use of the system's input devices. This design would also allow the development of UIS application programs on any VAX system, whether it was a workstation or not.

For the initial release of the MicroVMS workstation on the VAXstation I, these objectives were broken down into the following specific design goals:

- Provide routines for creating and manipulating viewports on the video display.
- Support multiple overlapping viewports and manage viewport occlusion transparently for applications.
- Allow simultaneous graphics operations into all viewports.
- Provide a user interface for viewport manipulations.
- Provide routines for creating graphics objects.
- Provide display-list backup for graphics operations so that applications can easily perform operations like "pan" and "zoom."
- Support shared access to the mouse and keyboard and provide routines to notify applications of input events occurring on these devices.

The following sections describe the architecture of UIS and the mechanisms that were used to realize these goals. Figure 2 is a block diagram showing the functions of UIS.

Virtual Displays

The fundamental presentation object manipulated by applications to construct images is the virtual display. All UIS output functions are performed within a virtual display.

The coordinate system of a virtual display is defined in "world coordinates." The world-coordinate system uses the coordinate system of an application as a means of expressing display locations. For example, an application that draws a graph showing population growth versus time may find it convenient to use "Time" and "Number of People" as x and y coordinates. The range of world-coordinate values is specified to the graphics subsystem when the virtual display is created. The coordinates are specified as signed F-floating VAX data types

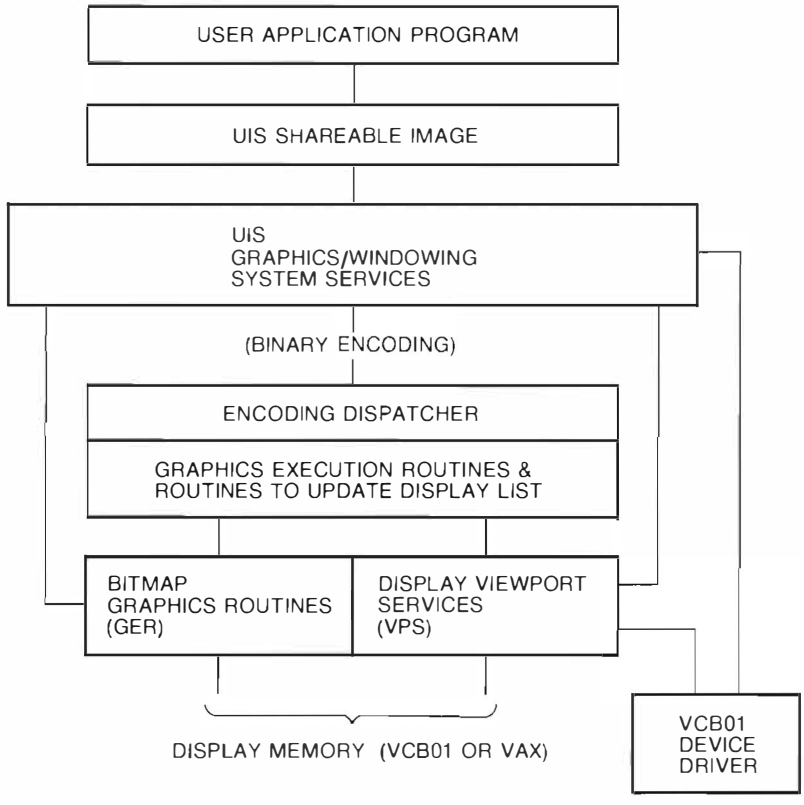


Figure 2 UIS Functional Block Diagram

for reasons of precision and ease of calculation in high-level languages.

A display list is an encoding of the exact contents of a virtual display, independent of the device. Display lists are maintained and used by UIS to achieve the following short- and long-term goals:

- Allow the automatic management of panning, zooming, resizing, and duplicating display windows
- Allow high-resolution printing of virtual displays
- Allow the structuring and manipulation of virtual-display objects
- Allow an application to select an arbitrary output from a virtual display, give it to an "intelligent" cooperating application, or simply store it in a file as generic encoding, and then later replay the generic encoding into a new virtual display

Display lists consist of the following basic objects:

- Output primitives
- Attribute primitives
- Structural primitives

Output primitives map directly onto the UIS output operations (e.g., plot some lines, write a text, draw a circle) and the modifications that they make to a virtual display.

Attribute primitives change the current value of an attribute in an attribute block in order to affect subsequent output primitives. Attribute blocks are used by UIS to specify a set of attribute values for all UIS graphics objects (lines, text, circles). Typical attributes include the writing mode (replace, complement, erase), line style (solid, dashed), and font to use when writing text.

There may be up to 256 attribute blocks addressable at one time. Attribute block

numbers are used and assigned only by the application, except for attribute block 0. This block is a special one that cannot be modified. It provides a set of attributes used as a standard default for text and graphics. Block 0 also provides a template for creating alternate attribute blocks.

Structural primitives allow the hierarchical grouping of attribute and output primitives into graphical begin and end blocks, called segments. Segments allow applications to have access to many more than 256 attribute blocks. While segments inherit current attribute blocks from higher-level segments, modifications to attribute blocks from within a segment cause local copies of the modified attribute blocks to be created. For example, if a particular attribute block is referenced within a segment, then that segment is first searched for the block. If the block isn't found, the search is made in successive outer segments.

The coordinate system, called normalized coordinates, is used both within the display list and when creating generic encoding. Normalized coordinates are used to defer the mapping of a set of world coordinates to specific device coordinates until the actual output device is known. As described in the following section, this mapping to the physical device does not occur until a display viewport is created. This delay is important since output devices have different resolutions. For example, printers typically have much higher resolutions than video monitors.

Since floating point calculations are typically slower than integer ones, normalized coordinates are expressed in units called "Gutenbergs," which are stored as 32-bit integers. A Gutenberg, the same unit used in UIS font definitions, is defined to be 1/7200 inch (.01 points). Their use as normalized coordinates is well suited because they minimize the number of coordinate transformations that must be performed when writing text. Gutenbergs have the desirable characteristics of being both reasonably small–and therefore amenable to good graphics resolution–and very efficient for text operations.

The conversion between world and normalized coordinates is based on the desired physical size and world-coordinate size of the virtual display as specified by the application. When a virtual display is created, the application expresses the desired size of the virtual display in both physical and virtual units. That establishes the relationship between the physical size of the fonts and the arbitrary size of a virtual display's world-coordinate system.

Display Windows and Viewports

A display window is the object used by applications to control how much of a virtual display is available for viewing by the user. This control is accomplished by defining a rectangle specifying the viewable portion of the virtual display.

A display viewport is the area of the physical screen into which a display window is mapped. Display viewports vary in size and may be placed anywhere in the physical screen area. Display viewports always occlude when they overlap. The order of occlusion usually depends on the order in which the display viewports were created. However, the order may be altered by the user through the UIS user interface or by applications using the UIS windowing services.

A display window is created, mapped, and automatically scaled to a display viewport when the application makes a single, routine call to UIS. Note that at the time of the call, the output of the UIS application is directed to a specific physical output device, usually the screen. Scaling can be avoided if the application directs UIS to use the physical size supplied by the application when the virtual display was created. That allows text and graphics to appear in exactly the size and aspect ratio that an application considers ideal.

The amount and size of the image that appears in a display viewport can be controlled by altering the size and position of the display window or the size of the display viewport. The image can be managed by either the application, through UIS, or the user, through the user-interface functions. The following rules govern the image:

- To magnify the image, either the size of the window is decreased without altering the viewport, or the size of the viewport is increased without altering the window.
- To reduce the image, either the size of the window is increased without altering the viewport, or the size of the viewport is decreased without altering the window.

- To change the amount of the virtual display being viewed without scaling, both the window and the viewport size are expanded or contracted by the same amount.
- To pan the image, the window around the virtual display is moved without altering the viewport size or location.

Figure 3 illustrates the mapping that takes place when going directly from a virtual display to a physical display. The left column shows the transformations between the coordinate spaces. The two columns on the right show the way the virtual display is scaled to the final output device.

Virtual Keyboards

Applications use a concept called virtual keyboards to share and individually manipulate the physical workstation keyboard. Virtual keyboards allow an application to get input from the physical keyboard and to modify its characteristics, both in a synchronized manner. Input can be received in either of two forms. First, applications can specify that they be delivered a software interrupt whenever keyboard input occurs. Second, they can periodically poll the virtual keyboard to see if new input has occurred. Certain characteristics can be managed for each virtual keyboard, such as keyclick volumes and keyboard key mappings.

The connection between the physical keyboard and the various virtual keyboards available on the workstation is generally managed by the user. An application could force the physical keyboard to be bound to a virtual keyboard. Typically, however, the application will associate the keyboard with some display viewport and allow the user to manage that connection through the user interface.

Mouse Input

Applications can both solicit and manage input from a mouse with respect to rectangles within display viewports. To do that, an application must specify a world-coordinate rectangle and

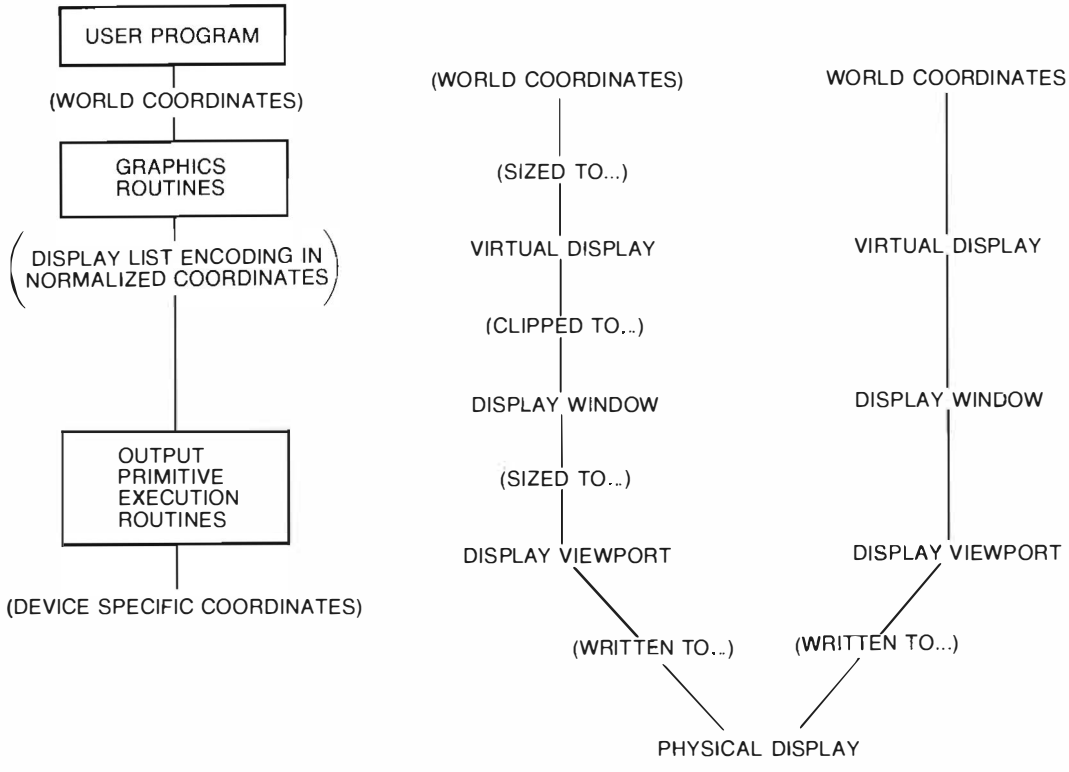


Figure 3 Mapping from Virtual-to-Physical Display

the display viewport to which the rectangle applies. The application then directs the UIS to

- Change the cursor pattern or position when the cursor moves within the rectangle
- Send a software interrupt whenever the cursor moves within or out of the rectangle
- Send a software interrupt whenever a mouse button is depressed or released within the rectangle

Applications can also check the current mouse position or button state at any time.

Implementation Details

UIS was designed with two primary implementation goals in mind. Of course, the first goal was to implement the architecture described in the previous sections. Just as important was the belief that the cost of using UIS had to be as small as possible. The overhead associated with a routine call had to be minimized, and the algorithms and architecture employed by UIS had to be as efficient as possible. UIS also had to be fast because the simple graphics hardware relied upon UIS software to take the place of sophisticated graphics hardware. To meet these goals, the software team made some basic design decisions right at the start. The effect of these decisions on how the design operates are discussed in the following section.

UIS operates in the caller's mode (usually user mode) because the cost involved in changing to kernel mode would be prohibitive. Because UIS operates in user mode, all data structures used by UIS are given user-write protection. This design decision means that timesharing use of the graphics package is possible, but without any security considerations.

Most of the UIS code resides in system space, and UIS routines exist as system services within the VMS operating system. That gives UIS all the desirable performance characteristics of operating system code (i.e., minimal image activation cost, maximum shareability, separately managed paging, etc.).

Fonts are stored in files and treated as system resources. Since several applications are likely to use the same fonts at the same time, UIS font management was designed to optimize font sharing. Fonts currently in use are kept in a font pool in system memory. Upon beginning a text-drawing operation, a process accesses the system font pool to find the required font. If not found in the pool, a font can be loaded into the font pool by searching the disk for the proper font file and then reading it into system memory. Similarly, fonts can be removed from the font pool because they can always be retrieved from disk.

Each virtual display is managed by only one process. That synchronizes the access to virtual displays and display lists and minimizes the effect that graphics applications have on each other. If a second process wants to manipulate the virtual display of another process, then the applications running in the two processes must communicate. The process that created the virtual display must then make modifications to it. This concept is enforced by the fact that the contexts for all virtual displays reside in process address space.

Data structures for display viewports, on the other hand, are kept in system space. That allows a process to change the topology of the viewports on the video display. For example, a viewport bound to a display window that it owns can be "popped" without having to notify every other process of the necessary screen changes. The storage for viewport data structures is allocated from paged pool. However, the storage protection must be changed to user write to allow access by the process-based graphics routines.

Access to those data structures by UIS routines is synchronized using the VMS lock manager. Multiple processes are granted shared read/write access to the physical display as long as they are simply reading from or writing to their own viewports. If a process needs to change the relationships between the display viewports on the screen (e.g., create a new viewport or pop an existing viewport), it must request exclusive read/write access to the physical display. Thus, no synchronization overhead is incurred in the steady state.

Figure 4 depicts the basic use of storage by UIS.

As shown in Figure 4, UIS software is organized into five basic parts.

The first piece of UIS that applications encounter is the UIS shareable image. UIS routines are accessed by applications through transfer vectors in a VMS-protected shareable image. That allows UIS code to increase in size and to change location within the operating system without affecting the applications that use the code. Also, UIS application development can occur on machines where UIS has not been

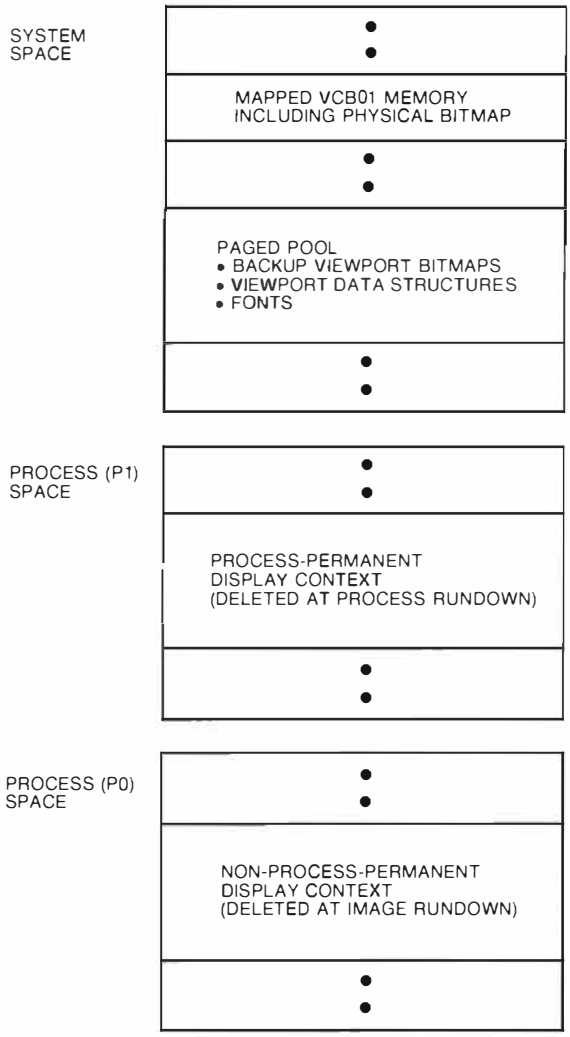


Figure 4 UIS Storage

installed. The UIS shareable image can be used to resolve UIS references at link and image activation time, even if the UIS system services are not present on the system. Finally, because the shareable image is protected, UIS can get control during image rundown and perform some necessary clean-up activities.

The shareable image performs the requested operation by calling the appropriate UIS system service. At this point, user requests are translated into calls to internal UIS routines, and the relevant internal data structures are located. For example, for a typical keyboard operation, UIS would locate the right virtual keyboard and make the appropriate calls to the VCB01 device driver.

For a typical output operation, such as drawing a line, UIS first creates a display list entry. UIS then calls the display list management routines to update the display list and all windows into the virtual display. These routines, in turn, will check with the viewport service routines (VPS) to find the right area of the physical screen in which to draw. Finally, the management routines direct the bitmap graphics execution routines (GER) to draw to those areas.

VPS is more than a simple screen rectangle manager. Its tasks are

- To present the rest of UIS with the "illusion" that viewports are always unoccluded and are contiguous pieces of hardware video controller memory
- To take advantage of VCB01 scan-line scrolling whenever possible
- To provide bitmap backup for occluded windows so that applications are free from the complexities of occlusion management

VPS does this by judiciously using and mixing three different types of video memory: on-screen VCB01 memory, off-screen VCB01 memory, and off-screen VAX memory. VPS also manipulates the entries in the VCB01 video scan-line map to present UIS with a virtual scan-line map, or virtual viewport, for each physical display viewport.

If the physical display has only one viewport, VPS will simply allocate a set of physical VCB01 scan lines and set up the viewport data structures to direct GER to that set. In this case, the physical and virtual viewports will be the same. However, if the display has occluding viewports, VPS will create a virtual viewport in off-screen memory for each physical viewport. Then, at 80-millisecond intervals, VPS will copy the modified contents of the virtual viewports to the physical viewports.

If changes must be made to the VCB01 video scan-line map, then VPS will update them. These changes could be caused by either a viewport that needs to be hardware scrolled or a change in the layout of the viewports on the physical screen. VPS then merges all the virtual scan-line maps and requests an update of the physical scan-line map. Those actions are done in synchronization with the 60-Hz video vertical-retrace interval.

Summary

Our initial goals were to design a workstation product with the MicroVAX I system, thus providing a stable, mature product available for the MicroVAX II system. The joint engineering task force was initiated in the spring of 1983; prototype graphics hardware was available in the early summer. Once that preliminary hardware was ready, the VMS team entered into full-scale development. The VAX/VMS workstation (VWS) product was developed during the fall and winter of 1983, and into the spring of 1984. VWS underwent customer field test with the VCB01 graphics controller, the MicroVMS system, and the MicroVAX I system in the summer and early fall of 1984. The first release of the VAXstation I was available in late 1984. This initial product allowed third-party VAX software vendors to take advantage of the VWS architecture.

Later, the VAXstation II replaced the MicroVAX I CPU with a MicroVAX II engine, thus gaining much higher performance. The MicroVAX II processor entered customer field test in the early spring of 1985, with shipments to customers by early summer. A new VWS software release that supported the VAXstation II was made available shortly afterwards. That VMS software was the fulfillment of this project's long-term goal.

Acknowledgements

We would like to acknowledge the contribution made by Dick Hustvedt to the MicroVAX workstation effort. Dick was instrumental in spearheading this undertaking. The contributions of Cathy Learoyd, Tom Furlong, Rob Scott, John DiMack, Mike Rosenblum, Jake Vannoy, and the rest of the VMS workstation team were also invaluable.

References

1. J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics* (Reading: Addison-Wesley, 1982).

2. *MicroVMS Workstation Graphics Programming Guide* (Maynard: Digital Equipment Corporation, Order No. AA-G110B-TN, 1985).

3. *MicroVMS Workstation User's Guide* (Maynard: Digital Equipment Corporation, Order No. AA-EZ24C-TN, 1985).

4. *MicroVMS Workstation Video Device Driver Manual* (Maynard: Digital Equipment Corporation, Order No. AA-DY65C-TE, 1985).

Nicholas A. Warchol
Stephen F. Shirron

The RQDX3 Design Project

The RQDX3 is a Winchester and floppy disk controller aimed specifically for use on MicroVAX II systems. The designers followed a top-down development process to meet their goals. Trade-offs, some requiring hardware and firmware to be built and tested for reliability, were identified and evaluated early in the project. The RQDX3 has a three-port data buffer to smooth data transfers between the host processor, the controller's microprocessor, and the disks. Four internal subsystems work in parallel to allow maximum system performance.

Design Goals

The project team set a number of specific goals at the start of the RQDX3 design. The greatest need was to improve the performance of the MicroVAX II system over that available with existing controllers, yet greatly reduce the manufacturing costs of the disk subsystem. The following list contains the goals that governed the design of the module:

- Cost–Obtain a manufacturing cost less than half of the best current disk controller, the RQDX2.
- Performance–The controller should not force an interleave of data sectors on the surface of the hard disk drives or limit the performance of the Winchester disk drives. The controller should also avoid wasting system-bus bandwidth on the Q-bus. The controller architecture had therefore to be chosen to allow the highest performance possible while meeting the other design goals.
- Dual Module–The controller should be designed so that it will fit on one Q-bus dual module. This form factor will allow the most flexible system configurations.
- Schedule–First customer shipment would be approximately one year from the project start. Meeting this goal would allow the phase-out of the higher cost and lower performance RQDX1 and RQDX2 modules.
- Testable Design–A high percentage of this module would be testable by providing extra hardware, microprocessor code, and test strategies. This design would help to reduce both manufacturing and maintenance costs.

The Design Philosophy

The team members decided that a top-down approach to the problem was the only way that the design goals could be met. A well structured, well documented design would allow the maximum communication between team members, and it would allow trade-offs to be made early in the design cycle.

The design process used in the project adhered to the following form:

- Set the goals and assign priorities to determine how flexible each one is; that will allow tradeoffs to be made if a goal is not attainable.
- Collect and study any overall system specifications and requirements that apply. This is the time to write the preliminary engineering specification and define the interfaces (both hardware and software) that must be adhered to. Any impulse to go back and change these specifications should be vehemently resisted.

- Analyze the problem and determine the system architecture based on the flow of information and the complexity of the required control functions. If the problem appears too large or is not easy to document or describe, then it should be divided into smaller, more manageable functions. During this phase, operational descriptions are created. Those can be flow diagrams, timing diagrams, state-transition diagrams, or anything that will help to explain how the controller should work. These descriptions should be included as part of the documentation package.
- Look for the solution to each problem while weighing it against the design goals. Iterations between this step and the previous one can be expected in order to meet the goals.

  This part of the process involves looking at the available technologies and other designs to determine what is or is not usable. If other designs have followed the same documentation strategy, then this task is much easier; if they have not, then do not waste too much time trying to "reverse engineer" those designs. The risk of using new technologies must be assessed to determine what impact they would have on the design's cost and schedule.

  The hardware design is documented using drawings called functional partitions. These drawings are a hierarchy showing the interconnection of functional, not physical, pieces of the design. All datapaths and control signals are named at this time. The drawings will be the reference point of the design team and make up a major portion of the design package. Because of the functional nature of these drawings, simulation of the design can be accomplished in a structured form.

  At this time, a technical description document is written to allow others outside the design team to understand the operation of the design. This document is especially useful in training new groups about the design as it progresses from the design phase to the manufacturing phase.
- "Paper debug" the design. This is an in-depth review by the design team before any hardware is built. The process begins with the operational descriptions and follows the documentation hierarchy down to the lowest level of the design. Normal operations and error conditions are checked, and each element is analyzed for test and diagnostic coverage.

  Mistakes found at this stage are much easier to fix on paper than in circuit boards, gate arrays, or software debugging.
- Build a prototype. This process includes the drawing of schematics to show the interconnection of the physical pieces, the layout of circuit boards, the development of gate arrays, and the writing of software routines that interface to the hardware.
- Debug the prototype. If the paper debug was done correctly, this stage should not uncover any disasters. The individual functional pieces of the design can be tested and checked off using the functional partitions as a guide. That systematic method will ensure that the entire design is tested.

The design process is the solution to a multidimensional problem. Therefore, there is probably more than one design that will meet the goals. There is also the probability that it may be impossible to meet all the goals. In this case, some compromise in the goals must be made in order to make a solution possible.

This design problem is like those encountered in most other designs: Make it fast, cheap, small, reliable, and don't take too much time. With each goal being constrained by others, the need for a structured method of finding a solution becomes more important. The way to solve a set of simultaneous equations is not to try a solution and see if it fits, but to use some proven techniques to determine the correct solution. Dividing the overall problem into smaller ones and then determining a solution is probably the most powerful technique that can be applied.

Design Implementation and Testing

Attacking the Goals

Each goal placed some unique restrictions on the design. Thus, it was important to understand the effect of each goal and how flexible the achievement of that goal was. By keeping a constant watch on how the goals were being met, trade-offs could be made very quickly.

The following discussion details each goal and how it was handled:

- Cost–This was the original goal that caused the creation of the RQDX3 project. The cost/performance relationship was higher than desirable for the current disk controllers. A project like the MicroVAX II system, in order to obtain a good market share, needed to improve this relationship by reducing the cost of the disk subsystem. Therefore, it was very important for us to attain our cost goal. To do that we placed a restriction on which components or technologies could be used, and what the assembly cost of the module could be. Maximizing the number of machine-insertable parts therefore became an important consideration.

- Performance–The MicroVAX II system would support the full VAX/VMS operating system. Since it supports virtual memory, the VMS system uses large data transfers in the disk subsystem. We therefore chose to optimize the performance of the controller around these large transfers to improve total system performance. By making the physical disk drive the limiting factor, we evolved an architecture that would allow simultaneous operations in the controller. In contrast, the current RQDX1 and RQDX2 disk controllers limit the data transfer rate between the host memory and the disk drive because of their architecture. The single thread of control in these modules, though adequate for PDP-11 systems, forced an interleave of logical data blocks on the disk surface. That interleaving would hinder the performance of the MicroVAX II system.

  There are also many techniques for reducing the average seek time of the disk drives. These methods include overlapped seeking on multiple drives, rotational optimizations, improved seek algorithms, and various data buffering techniques. We wanted to include as many of these optimizations as possible and, since the goals were driven by the design team, the trade-offs were a little more flexible.

- Dual module–This goal more than any other caused the most problems in the design of the hardware. Many times a solution seemed to meet all the goals but, when a detailed parts count and mock-up were created, there were a few components that just didn't fit on the board. Meeting this goal led to the extensive use of CMOS gate-array technology to meet this size restriction.

- Schedule–We did not have the luxury of setting the date for the project's completion. Because the disk controller was so important to the overall MicroVAX II project, we were given a completion date based on the availability of the MicroVAX II hardware. Of course, this procedure involved a management factor that certainly kept the design team on its toes by being told to see if we could do it. In response, we developed a schedule that would maximize the work that could be done in parallel while keeping the risks at an acceptable level.

- Testable Design–This goal became more important as the details of the design were completed. The module, being driven by an onboard microprocessor, would be capable of self-diagnosis. Therefore, where possible, all internally addressable registers were made to be write/read registers and extra datapaths were added to maximize the amount of logic available to the microprocessor for testing. This goal had to be weighed against the need for limiting the design complexity, cost, and size.

Task Partitioning

The short project schedule forced us to adopt a development strategy that would maximize parallelism in the development of the RQDX3. The first division was made between the hardware development and the microprocessor firmware development. Each major task was further reduced to smaller design functions. In many cases we had to create a model or emulator of some other undeveloped part of the design in order to allow tasks to continue.

Hardware Development

Once the functional partition drawings were created, we had a solution that met the performance and functionality that were required. However, we still did not know if the cost and board area requirements would be met. The design team quickly determined that some custom integrated circuits would be needed to help us meet these goals. Previous experience,

a known process, and quick turnaround made CMOS gate array technology the key to our solution.

Two gate-array devices would be needed, but we had only one gate-array design team on our project. We decided that one gate array would be developed first and a TTL emulator of the second device would be created and used for the module-level testing. In that way, the integration of the firmware under development with the hardware could begin early in the schedule.

The key area in almost any disk controller centers around the design of the phase locked loop and the data separator logic used in recovering the encoded data from the disk surface. We knew at the beginning of this project that our team did not have the experience to design this section. Therefore, we employed the services of outside consultants to this project. They contributed not only their previous experience in data separator design, but also reinforcement and management of the design philosophy taught to us in the past.

Firmware Development

To meet our schedule goal, it was necessary to begin development and testing of the firmware for the onboard microprocessor well before any hardware was ready. The firmware consisted of many modules, the majority of which were independent of the hardware. These modules could be designed, coded, debugged, and tested in parallel with the design, implementation, and debugging of the hardware. Then at a later date, the few remaining hardware-dependent modules could be developed and integrated to form the complete RQDX3 firmware.

Thus, the target system first used for developing the firmware was not the prototype RQDX3 with its onboard microprocessor, but a VAX/VMS system with two software emulators (one for the Q-bus subsystem and one for the disk subsystem). The VMS system was chosen for several reasons: first, it has an extremely nice set of program development tools; second, the VMS disk driver could be adapted to produce a steady stream of stimuli (disk I/O requests) to verify the correctness of the firmware's responses. With only a small amount of "trickery," the VMS system could be "convinced" to use a disk controller built not out of hardware, but out of software; the two emulators mentioned above provided the necessary glue. The emerging RQDX3 firmware could be developed in the context of a normal VMS process, taking full advantage of VMS compilers, linkers, and debuggers. Although it took a lot of time (and many system crashes) to get this technique to work, it greatly speeded up the job of building all the hardware-independent modules. This stage took about fifty percent of the total time spent to develop the firmware.

The next target system was the actual prototype RQDX3 with an in-circuit emulator (ICE) for the microprocessor and a TTL emulator for one of the gate arrays. Hardware debugging was accomplished first by special code written to perform repetitive actions on particular portions of the hardware. Then, the actual firmware, which had been previously developed and was, in a sense, known to work, was loaded into the hardware. The ICE was a great help here since it allowed RAM to be substituted for ROM; that allowed a level of symbolic debugging. At this point in the process, the hardware-dependent modules were built. This stage took about thirty percent of the total firmware development time.

The final target system was the "bare" RQDX3, with no emulators and real ROM. This configuration proved to be identical to the previous one (i.e., no problems were found in replacing the emulators with real devices), but allowed prototype boards to be shipped internally. The firmware of the RQDX3 could now be tested by different operating system groups, and bugs appropriately located and fixed. This stage took about twenty percent of the total firmware development time.

Design Verification Testing

The purpose of design verification testing (DVT) is to assess at an early stage whether a design has any particular implementation problems. To do that, the board is tested against all Digital's applicable standards. First, the layout of the board (the etch) is checked by looking for noise radiation and pickup, and for undershoot or overshoot on clock lines. Then, the board is checked thermally to see if it can withstand both operating and nonoperating environmental stresses. Next, FCC testing is done to measure the radiated frequency spectrum. Finally, the module is shaken and

dropped to ensure that no chip falls out of its socket under normal handling conditions. Feedback from DVT can result in physical changes to the module, perhaps as severe as a new etch layout.

In the case of the RQDX3, a recommendation was made to add resistors to a pair of clock lines in order to dampen undershoot. Fortunately, this alteration did not have much impact on the schedule.

Reliability and Quality Testing

The purpose of reliability and quality testing (RQT) is to demonstrate that the product meets certain minimum reliability standards, measured as mean time between failures (MTBF). The design team specifies the MTBF and also other measures of quality, such as hard and soft error rates, both of which affect the perceived quality of a disk controller product. Then, the RQT team designs a test that will demonstrate whether or not the product meets or exceeds these measurable quantities. Usually that involves building a system (CPU, memory, serial line interface) that includes the product under test. The system runs some level of host software that exercises the product for a large number of hours under various temperature and humidity extremes. Designing these tests is not an easy task, and indeed the RQDX3 had major problems during RQT because of this difficulty. Feedback from RQT can result in hardware changes, or firmware changes, or both. Ideally, if the product is changed, RQT should start again from the beginning. However, schedules will often not allow that and compromises must be made.

A decision affecting all of RQT must be made near the beginning: whether to test the product at the system level or at the module level. Testing at the system level implies that the system MTBF and error rates must be met, and all failures, whether related to the product under test or not, should be counted. Testing at the module level implies that the module MTBF and error rates must be met, and only failures that can be attributed to components under test should be counted. Clearly, module-level testing is preferred since it gives the most information about the new product. However, module-level testing is more difficult because each error has to be investigated to determine its cause and whether or not it should be counted. Furthermore, the burden of proof is on the design team to verify that the error was not caused by their module. (Guilty until proven innocent!)

Weighing all these factors, we decided to test the RQDX3 at the module level; that caused most of our RQT problems. A sealed chamber was used to control the tests of cycling over temperature and humidity extremes. The RQDX3 modules were placed in this chamber, along with the systems into which the modules were plugged. Part of the testing included reading and writing from both floppy disks and Winchester disks. Since these disks could not withstand the environmental extremes inside the chamber, they were placed outside. Early testing showed that this setup did not work, since the disk drives had to be connected to the controllers with lengthy cables, which were susceptible to noise pickup. This configuration was modified to bring the disk drives inside the chamber where they were connected to the controllers with normal cables. That eliminated the noise problem, but now dictated a reduced environmental stress on the RQDX3 module (from class C to class A).

At first, we encountered a higher-than-normal rate of soft errors on the floppy disks. A search for the cause of this problem showed that a combination of two separate but contributing problems were responsible. First, a rare combination of events could cause the data separator for the floppy disk to temporarily fail to lock to the data stream. Second, most if not all the floppy disk drives themselves were not performing correctly. The former problem was fixed by a component change to the data separator; the latter, by testing and repairing those drives that showed the greatest number of soft errors. These two changes reduced the soft error rate for the floppy disks to a level well within the range specified by the design team.

The extensive, and lengthy, RQT also uncovered one bug in the error handling of the RQDX3 firmware that had never been seen in our development lab. The problem could only have been experienced by running many, many modules in parallel. Of course, the purpose of RQT is to catch such problems then instead of at customers' sites.

The RQDX3 Architecture

The mass storage controller protocol (MSCP) defines the communication between the host processor and the disk controller. Communica-

tion occurs using sequences of command packets, generated by the host, and response packets, generated by the controller. The transmission of the packets and logical data blocks that are to move between the host and the controller is defined in the U/Q Storage Systems Port (UQSSP) specification. These two specifications place the following requirements on the controller:

- Two sequential-word register locations on the Q-bus are required. Those are referred to as the status and address (SA) register and the initialization and poll (IP) register. These registers must be able to be assigned at any longword boundary within the Q-bus I/O page.
- The controller must have the ability to interrupt the host processor using a previously loaded vector address.
- The controller must contain enough intelligence to initialize itself, perform internal diagnostics, decode command packets, perform all disk control functions, transfer data, and encode response packets. These tasks are accomplished on the RQDX3 through the use of a DCT11 microprocessor.
- The controller must be able to perform DMA data transfers on the Q-bus. These transfers will be for command and response packets, as well as for disk data.

The diagram in Figure 1 shows the flow of information in an MSCP controller. MSCP command and response packets flow between the memory in the host processor and the on-board microprocessor. Disk data flows between the memory of the host processor and the disk surface. Information dealing with the format of

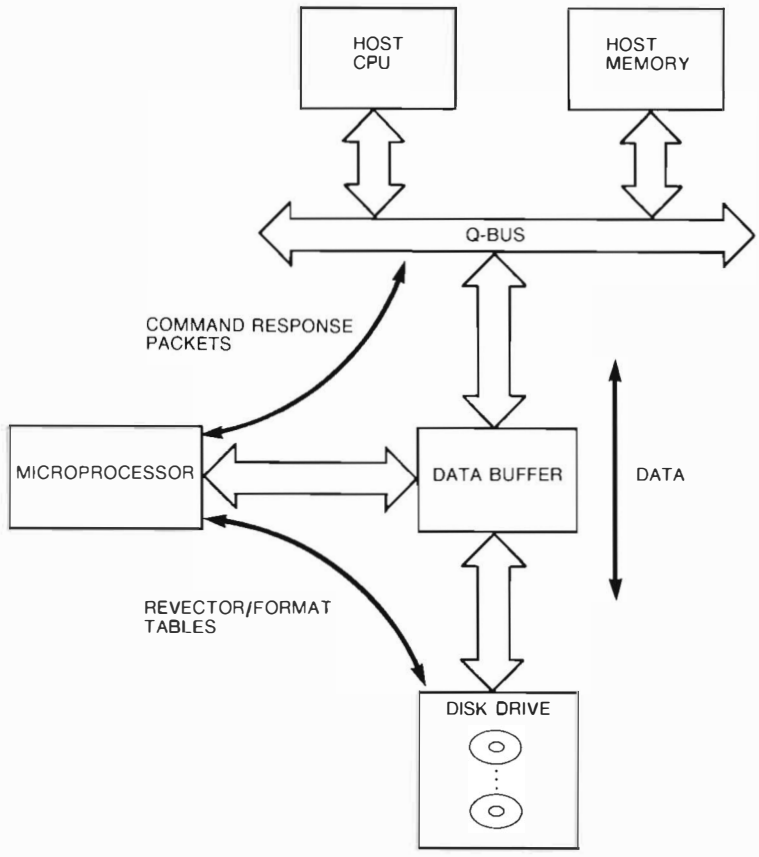


Figure 1 Information Flow in the RQDX3

data on the disk surface (revector tables, format tables, etc.) must be transferred between the disk surface and the microprocessor.

Figure 1 shows a centralized data buffer element. It is used for temporary storage and as a means for smoothing the differences in data transfer rates between the host memory, the microprocessor, and the disk surface.

It was decided to implement this centralized data buffer as a three-port memory system. Three control elements are provided for the transfer of data between each memory port and the appropriate source or destination. These elements are the Q-bus DMA controller, the microprocessor with its internal bus-interface controller, and a VLSI disk controller with an internal DMA interface. The interconnection of these subsystems is shown in Figure 2. Each control element assumes that it has the memory system for its own dedicated use. The arbitration between these elements for access to the memory devices is handled within the memory subsystem.

The Memory Subsystem

The memory subsystem contains a finite sequential-state machine that receives requests for memory cycles from the three ports and performs the memory cycle for the highest-priority requesting port. It is required that any port requesting a memory cycle must have its address and any required data available before

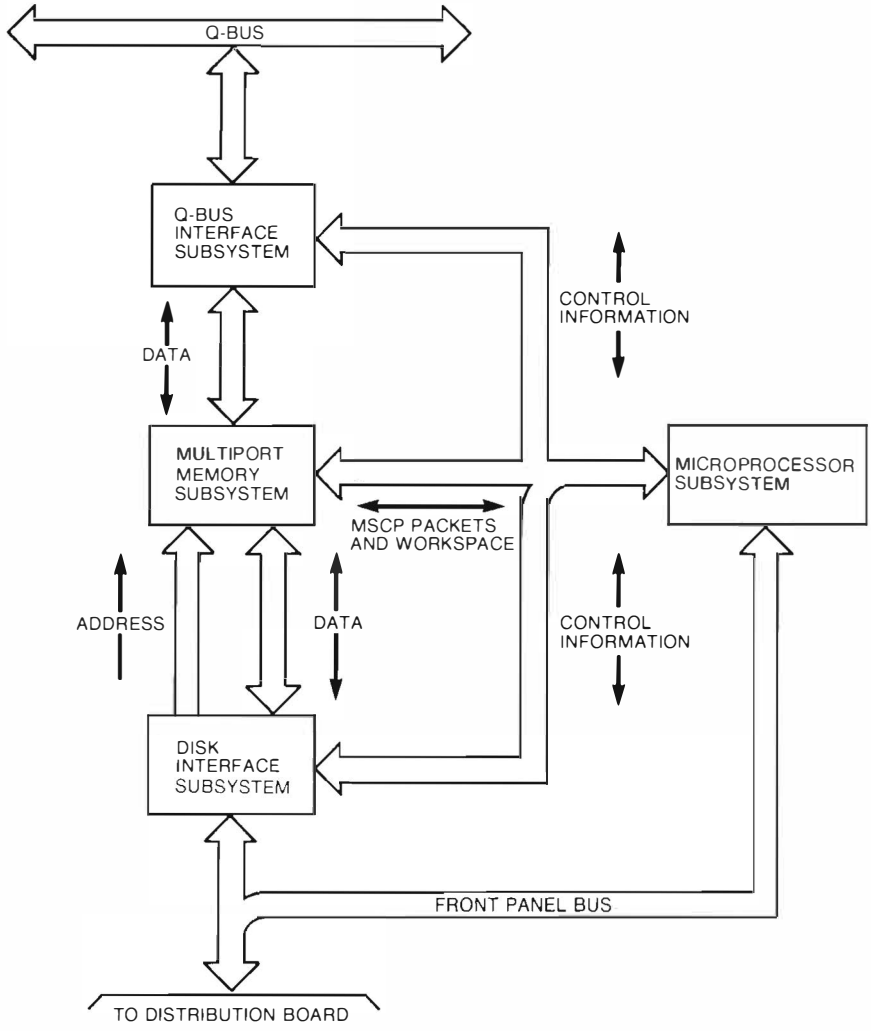


Figure 2 RQDX3 Subsystems

posting the request to the memory controller state machine. The principle function of the memory system is twofold: first, it allows the controller attached to a specific port to deposit data to be written to the memory in a holding register; second, it allows the memory controller to write that data to the RAM devices sometime later. For most read requests, the memory controller performs a prefetch operation when there is an empty output register in one of the ports. This operation is possible because the accesses by both the disk and Q-bus controllers are known to be sequential, with the next address always available to the memory controller.

The port of the microprocessor is an exception to this prefetch operation. The memory controller cannot prefetch the data since memory accesses by a microprocessor are not always sequential. When requesting a cycle from the memory, the microprocessor will be "cycle-slipped" (i.e., wait states added to its microcycle) until the memory controller determines that the microprocessor is the highest-priority requesting device.

The highest priority for memory cycles is given to the disk controller port. Failure to service this port first will cause overrun or underrun errors in the disk controller chip, which has little buffering. These error conditions would cause serious degradation of system performance, since full disk revolutions would be wasted retrying the operations.

The middle priority is given to the Q-bus DMA controller port. This port requires the highest service rate from the system (approximately 700 nanoseconds per request). However, the port is capable of slowing itself if it cannot be serviced in time by the memory controller. Of course, to achieve the highest system performance and most efficient use of the Q-bus, it is desirable that the Q-bus controller never slow down.

The microprocessor is given the lowest priority for memory cycles. That allows the normal operation of data transfer between the disk and host (both disk controller and Q-bus DMA controller active) to be completed as fast as possible. The microprocessor can use any remaining memory bandwidth for its operation. The microprocessor uses the shared memory for both temporary storage and its operational stack. Since its use of that memory will be infrequent, the microprocessor will not be affected by any loss in memory response.

A prototype of the memory subsystem was built to measure the amount of bandwidth available to the individual ports and to determine the effect of arbitration between the ports. A worst-case condition of requests from all ports was created and the bandwidth used by each was measured. With any two ports operating at their full speed, there was no measurable reduction in service rate from that of the ports running independently. When all three ports were operating, the disk port lost no memory bandwidth, the Q-bus port lost only one percent of its requested bandwidth, and the microprocessor lost eight percent of its requested bandwidth.

These observations during worst-case conditions indicated that all three ports are capable of operating at full speed with their normal request patterns. This feature of the RQDX3 allows it to overlap disk data transfers, Q-bus DMA transfers, and microprocessor operations to achieve maximum performance.

The memory controller is implemented using a field programmable logic sequencer (FPLS) and an external input sychronizer. Even though gate-array technology was used for the majority of the datapath on this module, it was felt that building the state machine in the gate array was too risky for the project schedule. The state machine was therefore placed outside the gate array. Only a few gate array pins connect it to the datapath elements that it controls.

The memory controller also incorporates some features to aid in the test and repair of the module. After module initialization, an input signal is asserted to force the memory controller to honor only those requests coming from the microprocessor. Without that, a hardware failure in either the disk controller or the Q-bus DMA controller could constantly request memory cycles and cause the microprocessor to "hang" on its first access to memory. With this signal asserted, the microprocessor can initiate the module diagnostics in a small, isolated environment that enables the microprocessor, ROM and RAM devices, and I/O page registers to be tested. The microprocessor can then clear the signal later in its diagnostics, thus completing the module testing.

The Microprocessor Subsystem

The microprocessor subsystem of the RQDX3 module is made up of a DCT11 microprocessor, 16K words of EPROM memory, a front-panel interface, and a prioritizing interrupt circuit.

Although many different microprocessors could have been used, the choice of the DCT11 was made with the following criteria in mind:

- A 16-bit microprocessor could handle the MSCP requirements adequately, while an 8-bit microprocessor would be strained and a 32-bit microprocessor might be an overkill.
- A multiplexed address-and-data bus would reduce the number of gate array pins required.
- A rich, orthogonal instruction set (PDP-11 system) that could be easily understood should be used.
- The microprocessor should be able to be programmed in a high-level language. Much of the code for this module would be written in the C programming language.
- Relatively fast execution speed is desired.
- Available hardware and software development tools should be used.
- Our past design experience should be exploited to improve the product's time to market.

The Q-bus Subsystem

The Q-bus subsystem of this module is made up of the programmed I/O section, the Q-bus DMA controller section and the Q-bus interrupt section. The Q-bus DMA controller is composed of a finite sequential-state machine and associated datapath elements that are used to perform both block-mode and nonblock-mode Q-bus cycles. The state machine is implemented in a field-programmable logic sequencer rather than a gate array to eliminate the risk of schedule delays due to coding errors. However, the datapath elements needed to support the state machine are contained within the gate array devices. Some of the features of this controller are

- Full 22-bit Q-bus addressing
- A 16-bit DMA word counter
- Q-bus memory parity detection
- Full, efficient implementation of Q-bus block-mode transfers
- A programmable holdoff timer to regulate the Q-bus activity

The Disk Controller Subsystem

The disk controller subsystem had to provide the control and datapath functions for both floppy and hard disk drives in the smallest space and for the least cost. This requirement was satisfied by using a VLSI disk controller device.

The RQDX3 data separator is designed to receive the encoded data stream from the disk and convert it into a binary data stream and clock, both of which are then fed to the disk controller chip. The data separator is designed to operate at three different data frequencies to be compatible with the available range of Winchester and floppy disk drives. The frequencies for each type of drive are as follows:

- 5-MHz MFM encoded data recovery from ST412 Winchester disks (RD5X type)
- 500-KHz MFM encoded data from high-speed, high-density floppy disks (RX33 type)
- 250-KHz MFM encoded data from standard double-density floppy disks (RX50 type)

The data recovery system for the RQDX3 is a unique MFM data recovery circuit that is very close to ideal. In short, with proper matching of the device delays, the recovery window is +50 nanoseconds, or one hundred percent of the window. This almost ideal data recovery is made possible by the following conditions:

- A solid and precise phase locked loop is used.
- The MFM encoding rules specify a 100-nanosecond "null" period after each flux transition. This period is used to reset the edge store and compensation flip-flops of the circuit.
- The VCO output has a fifty percent duty cycle.
- The logic delay paths in the data separator circuits are carefully matched. This matching was accomplished by device matching within the gate array that implements this function. Careful simulation of this logic was carried out to prove this operation.

The Structure of the Firmware

The firmware had to be designed to take full advantage of the parallelism provided by the chosen hardware architecture. Therefore, the RQDX3 firmware consists of a set of cooperating routines, or jobs, each of which performs a dedicated function. Each job has its own stack and thus its own context and state information. Any operations that could possibly run in parallel have been separated and are controlled by separate jobs. A small operating system kernel provides facilities for creating new jobs, suspending and resuming execution of a given job, acquiring exclusive access to shared resources and later releasing those resources, and scheduling jobs to run based upon priority and resource contention criteria. This kernel provides a controlled way of overlapping operations. That effectively means that the RQDX3 can be simultaneously seeking on one or more drives, reading or writing from another drive, and transferring data to or from the host, all while performing calculations relating either to the current transfer or to a pending transfer.

Performance Tests

The main performance goal was to be able to sustain a high data-transfer rate for large transfers. In a typical situation, the VMS system uses the disk to swap, page, and load images. The RQDX3 is tuned so that these operations are completed as rapidly as possible. Maximum sustained data transfer rates of 420KB per second have been measured, compared to 170KB per second on the RQDX2. Such workloads are atypical, though, and do not give a good indication of overall system performance. When tested with a workload of from one to fifteen users on a MicroVAX II system, the RQDX3 is faster than the RQDX2, but slightly slower than the KDA50. This relationship is more in line with the performance based on theoretical calculations. A user workload generates a lot of seeking, and the RD-class disks controlled by the RQDX2 and RQDX3 seek more slowly than the RA-class disks controlled by the KDA50.

Higher performance can be gained by splitting the disk activity among two, three, or even four disks. The RQDX3 has the ability to keep all four drives seeking at the same time. For small transfers, seek time dominates, and an increase in system throughput of thirty-five to forty percent can be realized. For large transfers, seek time is still important but decreases in significance; the increase in system throughput may only be twenty percent. The RQDX2 does not take advantage of separate system and user disks; however, the RQDX3 will.

Higher performance on a single drive can be achieved by queuing multiple requests to the RQDX3. The MSCP protocol allows these multiple requests to be automatically reordered by the controller to reduce the average seek time. For example, the controller could always choose the request with the shortest seek time instead of the first request in its queue. An increase in system throughput of thirty to forty percent occurs when the number of outstanding I/O requests increases from one to twelve.

Summary

The RQDX3 design project came close to meeting all its design goals. There were 40 working units exactly one year after the project began. However, problems in the reliability test setup, which delayed the manufacturing startup, caused our first customer shipment to slip. The cost, performance, and module-size goals were all met to the satisfaction of the design team. The high yields in manufacturing can be attributed to the quality of both the design and the manufacturing process. Without the structured design process and the team's adherence to it, this project would not have been successful.

References

1. W.I. Fletcher, *An Engineering Approach to Digital Design* (Englewood Cliffs: Prentice-Hall, 1980).

Kathleen D. Morse
Lawrence J. Kenah

The Evolution of Instruction Emulation for the MicroVAX Systems

The MicroVAX CPU, the 78032 chip, implements a subset of the VAX instruction set, yet the operating system must support the full set. To accomplish that, the MicroVMS developers decided to emulate the missing instructions–floating point, packed decimal, and character string instructions–in software. Since hardware and software were developed in parallel, a VAX-11/730 system, with its microcode rewritten to make it act like MicroVAX hardware, was used as a test vehicle. The performance measurements indicated excessively long execution times. The hardware design was extended to assist the software emulation task. The final emulator was also used in the ULTRIX-32 and VAXELN systems.

When Digital Equipment Corporation decided to implement the VAX architecture[1] in silicon, it was clear that the entire instruction set could not be implemented on a single chip. To determine what could be implemented, a team of software and hardware engineers was formed to identify the best subset of the VAX instructions that would fit. As a consequence, the software engineers had to find ways to provide support in the operating system for those instructions removed from the base machine. This paper discusses how that emulation support was provided.

MicroVAX Architecture

The amount of microcode needed to implement an instruction is a good measure of the amount of space needed on a chip to implement the same instruction. Microcode size thus became one measure used in determining which instructions to move off the chip. A second criterion was the frequency with which particular instructions are used. For example, integer and logical instructions are used very heavily and their frequency of use is independent of the application area. Floating point instructions appear most frequently in scientific and engineering computations. Packed decimal instructions are more common in certain commercial applications. Eventually, by balancing these considerations, the engineers identified a subset of the VAX instruction set that would fit on one chip. That subset became the definition of the MicroVAX architecture. (The subset architecture also differed from the full VAX architecture in such areas as the console subsystem.)

Once the MicroVAX architecture was completed, the hardware and software teams began independent development efforts. Since a major project goal was to minimize the time to market, one hardware team investigated a MicroVAX implementation (the MicroVAX I system) that used semicustom logic instead of a single chip. A second hardware team started the design of the MicroVAX chip itself[2], and a third team initiated the design of the implementation (the MicroVAX II system) that would incorporate that chip. At the same time, the software teams began their investigations of how to enhance the VMS, ULTRIX-32, and VAXELN operating systems in order to run these new machines. The software designs were influenced in part by the need to implement and test the missing-instruction software emulation before any hardware was available.

Operating System Support

The major difference between the software architectures of the MicroVAX and the full VAX systems is the group of instructions that were

not implemented in the chip hardware. This group consists of

- Floating point instructions
- Packed decimal instructions
- Character string instructions

(The MicroVAX architecture included the MOVC3 and MOVC5 instructions because they were heavily used in fundamental routines, such as copying or filling memory arrays.)

Each of the three operating systems was supported by a different design group. These groups had to decide which course of action to take to accommodate the reduced number of instructions that would be implemented in microcode. The following alternatives were the most realistic courses to take:

1. All compilers and assemblers could be changed to eliminate all uses of the missing instructions.
2. Emulation subroutines that applications could link into their programs could be supplied. (VMS used this method on early VAX models that did not include hardware support for the G and H floating point data types.)
3. The emulation subroutines could be implemented so that their use would be invisible to application programs and even to most of the operating system.

The VMS Decision Process

The VMS design team began a study to determine the extent to which the missing instructions were used in the operating system code, including all the various VMS utility programs. As expected, the character string instructions were used most frequently and, in fact, were more widely used than expected. The CMPC3, CMPC5, and LOCC instructions were the most frequently used string instructions, occurring almost everywhere that ASCII text was manipulated (for example, in device names, file names, and DCL commands). All software that included some kind of bitmap (about six to ten different areas, ranging from the file system to memory management) used the SCANC and SPANC instructions. A large number of table-lookup designs (including DCL and utility command parsers) used the MATCHC, MOVTC, and MOVTUC instructions. Finally, the CRC instruction was used by the BACKUP utility and by the DECnet code.

Very few data types were used outside their realms and only a few unexpected sequences were found that used the missing instructions. One example was the use of the CVTLF instruction in the VMS kernel to determine the smallest power of 2 larger than a given integer. A second example was the use of the CVTLP instruction in the FORTRAN run-time support library as a quick method for converting binary representations to text.

Once the extent of the missing instruction usage was determined, the design team considered the number of compilers that were supported by the VMS operating system. In all, over fifteen different languages are supported.[3] The first alternative, changing the compilers and assemblers, would require that the code generators for each product be changed. Moreover, new versions of the VMS operating system and all its layered products would have to be generated using these new compilers. That would involve a significant investment of manpower, not just to enhance the compilers, but to provide ongoing support to maintain each product. In addition, two variants of each new version of each product would have to be produced. A likely side effect was that these changes would probably cause other development groups to limit most layered products to the MicroVAX subset on all VAX machines. In that way, each group would have to maintain only one version of their product.

Another consideration was the effect that the first or second alternatives would have on the marketing of MicroVAX systems. Customers and Digital's software engineers had become accustomed to developing software on one machine and executing it transparently on any other machine in the VAX family. That would not have been possible under either of the first two alternatives.

Through this reasoning process, it became obvious that the correct choice was the third alternative, to design for software emulation and make it transparent to both applications and operating system code. While requiring a concentrated effort to write the emulation support, the overall effort for software emulation was much smaller than removing the use of the missing instructions from existing software and compiler code generators. The effort was also

isolated. While some new code was needed, the number of changes to existing components was minimized. These changes were confined to the exception handler and the startup routines for the operating system. Finally, transparent emulation of all missing instructions would guarantee that systems implementing the MicroVAX architecture would be fully compatible with the VAX family of machines.

Implementation

As mentioned earlier, the MicroVAX program was geared to a tight time-to-market schedule. That made it highly desirable to develop the hardware and software in parallel as much as possible. The VMS design team decided to implement the emulation code and debug it long before the hardware design specifications for a particular MicroVAX implementation were written. In this way, the emulation code would be finished and working by the time the first MicroVAX hardware was ready to be debugged.

Design of the Emulator

At this point in the project, several decisions were made relating to the design and implementation of the MicroVMS instruction emulator. The emulation routines would be developed and tested by the VMS Development Group. These routines would attempt to avoid features or coding techniques specific to the VMS operating system. Thus the same emulation source code for the instructions could be used later by the ULTRIX-32 and VAXELN Development Groups.

The emulation support was divided into two pieces. The first supported character string and packed decimal instructions (including CRC and EDITPC); the other, floating point data types. From the beginning of the MicroVAX effort, system configurations would be offered that provided some sort of floating point support in hardware.[4] That fact influenced the design of the two pieces in the emulator.

Software support for floating point was viewed as a technique for running programs that contained small amounts of floating point computation. Applications that depended heavily on floating point operations would likely be run on systems that had floating point support in the hardware. Conversely, applications that depended heavily on packed decimal or character operations did not have a hardware option at their disposal. The decimal/string emulator reflects that in several places where space is sacrificed in an effort to speed up the emulation subroutines.

Structure of the Emulator

Once the two pieces were designed, the actual coding began. Each of the two emulation components was further divided into an operand decode piece and an instruction execution piece.

The operand decoder was a straightforward finite-state machine. It parsed the instruction stream one operand at a time, placing results into registers "appropriate" to each instruction. The register assignments were usually made by examining the expected register contents after each instruction had completed its execution. For example, the final state of a CMPC5 instruction suggests that R1 and R3 be used as pointers to the two character strings, while R0 and R2 contain the initial sizes of the strings.

The instruction execution routines were simple subroutines that accepted input parameters in registers and produced output conforming to the architectural specification of the instructions. For example, after the execution of an ADDP4 instruction, R0 and R2 contain zero, R1 and R3 locate the addend and sum strings, and the other registers are preserved.

At the outset, several other decisions were made that simplified the design and implementation of the emulator.

- Emulation support was provided transparently by being implemented at a very low level in the operating system.
- Emulation subroutines were executed in the access mode of the missing instruction.
- The existing emulation support for G and H floating point data types would serve as a base for full floating point emulation support.

Transparent Support

To emulate the missing instructions transparently, the emulators had to become an integral part of the operating system. They were loaded into system space during the system bootstrap and connected directly to the reserved-opcode exception vector in the system control block. Whenever a reserved-opcode exception occurred, the emulator would distinguish the

execution of a missing instruction from other illegal opcodes. Missing instructions would cause a control transfer to the appropriate emulation subroutines. Other illegal opcodes were passed on to the operating system as exceptions. Since the host operating system provided support in a transparent fashion, existing programs could execute on a MicroVAX system without being changed.

Access Mode of Execution

The reserved-opcode exception handler had to begin its execution in kernel mode, as defined by the VAX architecture. However, if the emulator routines continued in that mode, the address validation rules demanded that not only each operand but also each byte in a character string be probed for read or write access before that operand could be used. Because of the excessive cost of these operations, we decided that the emulator routines would execute in the access mode in which the missing instruction was used. If an operand or string was not accessible, an access violation exception would occur, which could be intercepted for special processing by the emulator.

The Use of Existing Routines

An emulator for G and H floating point data types already existed. Instead of completely rewriting this emulator to accommodate all four data types, it was restructured to separate its operand packing and unpacking routines from the arithmetic and conversion operations. Then, additional packing and unpacking routines were added for F and D floating point data types. Also, the overall structure of the floating point emulator was changed from a condition handler to an integral piece of the operating system. (A condition handler executes only within user programs, while an integral component would receive control whenever a missing floating point instruction is executed.)

Initial Testing

It was obvious that a testbed was needed to enable the design team to debug the emulation software. Some method was needed to force the emulation software to gain control in order to execute the missing instructions. Since the VMS macro assembler can substitute a macro for an instruction opcode, macros could be used to cause the assembler to take special action whenever it encountered any of the missing instructions.

A set of macros was written that caused special object code to be generated whenever any of the missing instructions was encountered. This special object code consisted of a byte containing the illegal opcode FE(hex), the opcode for the instruction, and all the operand specifiers. When one of these instructions was executed, a reserved-opcode exception was generated. A special exception handler would then advance the PC from the byte containing the FE opcode to the actual opcode. Control was then passed to the instruction emulator. One of these macros is listed in Figure 1.

Using these macros, programs written in assembly language could be reassembled and executed using software emulation for the missing instructions. Thus any existing VAX processor, such as a VAX-11/730 system, could be used as a testbed for the software emulation.

Results of Initial Tests

One key factor to determine was the increase in execution time required by software emulation for different parts of the operating system and for application programs. To determine these differences, the VMS Performance Group at Digital ran standard instruction-timing tests against the emulation code. Because these tests were run on an existing VAX processor, the execution times for emulated instructions could be compared to those done in hardware on the same VAX processor. These test results showed that it took about ten times longer to emulate character string instructions than to execute them in hardware.

To determine the reasons for this disparity, the design team performed a close inspection of the emulation code. Quite quickly it became obvious that, for the simpler string instructions, the operand decode required as much time as the instruction execution. To speed up the emulated instructions, hardware support was requested by the MicroVMS team.

To support this request, we made a list of the operand types for the missing character string and packed decimal instructions. There were only 5 operand types in all 27 instructions. These operand types were already being used by instructions that were a part of the MicroVAX subset, such as MOVC3 and MOVC5. A meeting of the hardware and software teams

```
        .title  locctst
        $opdef

; Redefine the LOCC opcode with a new LOCC macro

        .opdef  locc_fe <<op$_locc@8>!^xfe>,rb,rw,ab

        .macro  locc    char.rb,len.rw,addr.ab
        locc_fe         char.rb,len.rw,addr.ab
        .endm   locc

desc:   .ascid  "This is a test"        ;Test data for LOCC

; Test program to try a LOCC instruction.

        .entry  start_here,0            ; Entry point for test program
        locc    <#^a" ">,desc,@desc+4   ; Generate an emulated LOCC
        movzwl  #1,r0                   ; Standard exit status code
        ret                             ; Exit from program
        .end    start_here              ; End of test program
```

Figure 1 Test Program with Macro for LOCC Instruction

concluded that there would be little cost to the underlying hardware if these operands were decoded before a missing instruction exception was signaled.

Design of New Emulation Exceptions

The result of that meeting was that two new exceptions were added to the MicroVAX architecture as emulation assists. Since the hardware could easily decode the operands for the character string and decimal string instructions, they were defined as the ones that the new exceptions would support. Thus, two of the three instruction types not implemented in hardware could now be handled effectively. The third type, floating point instructions, would continue to cause reserved-opcode exceptions, since their operands could not be decoded without significant additional hardware support. (A separate floating point unit, the MicroVAX 78132 chip, provides this hardware support for three of the four floating point data types.)[4]

The first exception is generated whenever a character string or decimal string instruction that is not in the hardware subset is executed. The process causes the hardware to decode the operands and push the exception parameters onto the current stack. The exception parameters are depicted in Figure 2.

The second exception occurs only when one of the emulated instructions is executed and the first-part-done (FPD) bit is set in the program status longword (PSL). The VAX architecture allows many instructions (including all the decimal and character string instructions) to be interrupted after partial execution. The original operand specifiers cannot be decoded again because the register contents may have been altered to store the intermediate results. When this second exception occurs, the exception handler unpacks the intermediate results and resumes execution at the point where the instruction was interrupted.

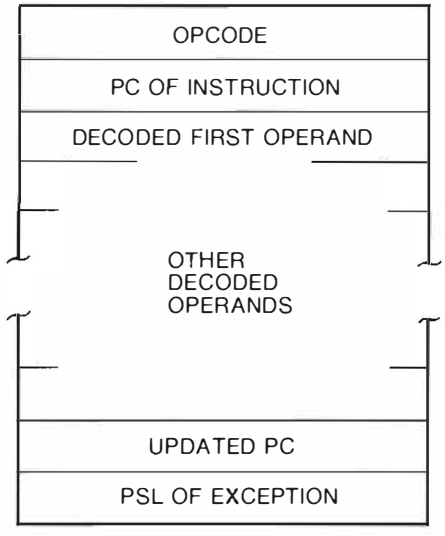


Figure 2 Exception Parameters for Emulation Assist Exception

Note that this second exception can occur only when an access violation has already occurred during instruction emulation. In that case, the operating system's access violation handler transfers control to the emulator. Enough intermediate state is stored in the registers to allow restarting the instruction, at which time the stack is restored to its state when the instruction began execution. Then the exception PC is changed from a PC inside the emulator to the PC of the original instruction that triggered emulation. Finally, control is passed back to the operating system's exception reporting mechanism. (Page faults, device interrupts, and the like are invisible to the user and require no special handling. That is, there is no need to pack the state into the registers and alter the saved PC.)

Final Design of the Instruction Emulators

The final design produced emulation support in two pieces: one for the missing floating point instructions; the other for packed decimal and character string instructions. Although the two emulator programs supported different data types, their overall design contained many common threads. This section describes the common design philosophy, as well as the step-by-step operation of each emulator.

Common Design Philosophy

Nearly all the emulation code executes in the access mode in which each missing instruction was originally executed. The stack associated with that access mode is used as a working storage area for the emulation routines.

The emulation of missing instructions is nearly invisible to programs in the sense that memory and register contents are identical to those obtained on full VAX implementations. The only difference between the emulated and hardware implementations is in the time required to complete an instruction and in the stack remnants from the emulator's temporary storage area. (Memory locations at small negative offsets from the top of the stack are specified as UNPREDICTABLE in the VAX architecture.)

The two emulator pieces share a common philosophy, if not common code, in regards to the two memory management faults. One fault is made in response to an invalid page and the other when a reference is made to a page that is not readable or writable as required.

No special treatment is required for page faults (translation-not-valid faults). If an invalid page is referenced by the emulator, a page-fault exception is reported to the operating system. The PC in the page-fault frame points at the instruction within the emulator that referenced the invalid page. After the operating system makes the page valid, execution resumes with the faulting instruction.

References to pages that are not accessible (access-violation faults) are more complicated than the page faults. Access-violation faults, unlike references to invalid pages, are visible at the program level. When the emulator intercepts the exception, the faulting PC points at the emulator instruction that references the inaccessible page. The stack contains working storage that must be removed and saved registers that must be restored. In that way, the exception looks like an access violation generated on a full VAX implementation. For most floating point instructions, an access violation implies that the state of the machine will be reset to its state when the instruction began. For the decimal, string, and POLYx instructions, the instruction can be left in a partially completed state. The intermediate context is stored in the registers and the FPD bit is set in the saved PSL. This bit allows the emulator to resume these instructions at the point where they left off, rather than restarting them from the beginning (assuming that the access violation can be resolved).

Floating Point Emulation Support

The program that emulates the missing floating point instructions in software differs in several details from the decimal/string emulation routines. In floating point emulation, the functions are performed in the following order:

1. Execution begins in kernel mode as a result of a reserved-opcode exception.
2. If the exception occurs in a mode other than kernel, the exception parameters are copied to the stack of that access mode. Further emulation takes place in that access mode.
3. Each operand is decoded.

4. Floating point operands are unpacked into exponent and mantissa.
5. The operation (arithmetic or conversion) is performed.
6. If the result is a floating point number, the resulting exponent and mantissa are packed into a single number.
7. The result is stored and the exception dismissed.

Before the exception is dismissed, the floating point emulator examines the opcode of the next instruction. If it is also a floating point instruction, then control is passed back to the beginning of the emulator to begin the operand decode for the next instruction. This technique saves the overhead of dismissing one exception and immediately generating an identical reserved-opcode exception.

The nature of floating point operations allows many instructions to accomplish their results by sharing different routines. There are routines that can unpack and pack each of the four floating point data types. There are also routines that perform the various arithmetic and conversion operations. Because these routines operate on unpacked numbers, the routines are independent of the initial data type.

The floating point emulation routines support all four floating point data types. Thus the routines can be used with all MicroVAX systems and other VAX systems that do not implement all four floating point data types in firmware or hardware.

Decimal/String Emulation Support

The emulation of a character string or packed decimal instruction proceeds as follows:

1. Execution begins in the access mode in which the missing instruction was originally used.
2. Operands are moved from the stack into registers and control is passed to an instruction-specific routine.
3. Some instruction results (for example, from MOVTC, MOVTUC, and packed decimal arithmetic and conversions) are stored while these routines are executing.
4. The routine executes until an input or output string is used up, at which time it completes the storage of results. Execution is resumed with the next instruction.

Because the decimal/string emulator relies on hardware for its operand decode stage, the lookahead technique used by the floating point emulator cannot be used for decimal and string instructions. If the instruction following an emulated instruction also requires emulation support, the following sequence takes place:

1. The first exception is dismissed.
2. The next instruction is executed.
3. The operands of that instruction are decoded and stored on the stack.
4. The decimal/string emulator regains control.

Since these instructions perform many unrelated operations, there is little code that can be shared between their emulation routines.

Testing and Debugging

The main problem in testing the emulation software initially was that there was no MicroVAX hardware available during most of the implementation cycle. Thus we had to develop techniques to simulate the hardware in order to begin the tests. There were two chief techniques used to test and debug the emulator. First, instruction-specific routines were tested as user-mode programs in a normal program development environment. Second, the exception handler front-end was tested on a VAX-11/730 system that was modified, by rewriting some of the 11/730 microcode, to act like a MicroVAX system.

Instruction-Specific Testing

Microcode written for a particular implementation (both VAX and MicroVAX systems) can be used only on that particular machine or a simulation of that machine. However, macro-level code can be executed on any VAX processor. Therefore, since the emulation routines were written in macro-level code that executes on any VAX processor, "normal" debugging

techniques could be used for part of the debug effort.

A set of test programs was constructed that would run on other VAX processors (11/730, 11/750, and 11/780). These test programs would call each instruction-specific subroutine and compare the results (memory contents, register contents, and settings of the condition codes) with the output from the corresponding instructions executed on those processors. These tests allowed the basic algorithms to be debugged even before they were plugged into the emulator. The set of tests was limited only by the choice of input data for each instruction.

The first set of tests uncovered most algorithmic problems but did not exercise the error paths (such as inaccessible source or destination strings). The code to handle these error conditions was written later in the development cycle. Neither the absence of these error paths nor errors in edge conditions (such as zero-length strings) prevented the VMS system from executing.

Another benefit of a macrocode implementation was seen during the debug of the edge-condition problems. Since the instruction emulation routines were just an extension of the operating system, the debugging tools used for other operating system code could be used to debug the emulator.

Testing the VAX-11/730 Breadboard Implementation

The availability of the two new emulation exceptions changed the strategy for debugging the emulation code. The software solution used to obtain preliminary results was unable to mimic the new exceptions invented to assist the emulation. Therefore, a new testbed was needed to accommodate the debugging process. The testbed had to decode the operands and generate the appropriate exceptions to pass control to the software emulation code. One way to perform these functions was to alter an existing VAX system, such as the VAX-11/730 processor.

The 11/730 is an entirely "soft" machine; that is, all its microcode is loaded at powerup rather than being resident in ROM. By altering that microcode, the design team could make the 11/730 look like the architecture in a MicroVAX system. The required changes were simply a matter of removing the microcode for instruction execution while leaving that for operand decode. To finish the alterations, the design team had to write a new "exception generator" to create the emulation exceptions.

At this time in the project, the first real MicroVAX hardware would still not be available for nine months. Therefore, the VMS design team decided to undertake the modifications to the 11/730's microcode and to build the testbed. We estimated that this effort would take one to two months, since the VMS developer had to learn to write microcode. That meant that the software emulation code would still be completed long before the first MicroVAX hardware was ready.

The microcode source programs were acquired from the 11/730 microcode team and assembled using the latest version of the microcode assembler. The 11/730 microcode was structured as separate modules for different functions (for example, floating point, compatibility mode, exceptions, memory management, and so on). Due to the lack of a "linker," label files that allowed routines to be called across modules had to be created. To speed the development, the design team wrote several FORTRAN tools that automatically generated new label files. In addition, command files were built that correctly created a new set of binary microcode files from a set of modified sources.

The next step was to change the 11/730's microcode. Since it had to exist in a limited amount of RAM space, the new code could not be added without removing some existing code. Therefore, we decided to replace the compatibility mode microcode with a new routine to generate the emulation exception. Some new flags were added that, at the developer's choice, would allow different classes of instructions to be emulated (i.e., decimal string, character string, or floating point). Finally, to boot the VMS system on this MicroVAX version of an 11/730, we had to enhance the VMS bootstrap code to load the emulation exception handlers and connect them to the appropriate exception vectors.

Now the software emulation code, from the exception handler all the way down to instruction execution, could be debugged. The best measure of the success of this venture was made when MicroVAX hardware was finally available. The customized VAX-11/730 system was such a good testbed, not only for the

instruction emulator but also the rest of the MicroVAX I support, that it took a mere four days to get the VMS system running.

Other Test Mechanisms

The initial testing of the instruction emulator consisted of a set of programs and sample input data for each of the missing instructions. While providing routines that worked in almost all cases, these tests did not exercise some of the more exotic edge conditions. Those included very long or very short strings, illegal operands, or strings that were not readable or writable. Once MicroVAX hardware was available, several new testing techniques could be used to exercise the emulator.

Operating System Code

More testing was provided by running the operating system code with the emulator providing character-string and packed-decimal support. The VMS Development Group has a large set of regression tests that exercise most success and error paths within the operating system. These tests plus normal daily use by the VMS development community ensured that extensive testing of the instructions used by the VMS operating system was performed.

Once the VMS system was running, the ULTRIX-32 and VAXELN Development Groups requested the source code for incorporation into their systems. These systems exercised parts of the emulator that the VMS system did not use. The ULTRIX kernel uses a small number of packed decimal instructions (ASHP, ADDP4, SUBP4, and EDITPC) for some of its arithmetic and formatting support. When the ULTRIX-32 operating system first exercised the emulator, several bugs were detected and corrected.

Compiler-Generated Code and Associated Tests

The base operating systems used packed-decimal and floating point instructions in a small number of cases. These instructions received better testing using programs written in COBOL and FORTRAN. The compilers and their validation tests were used to test the emulator routines from the time they were first written until they finally shipped.

Architectural Conformance

Even such continual testing is no guarantee that each instruction executes according to the VAX architecture specification. Most of the testing described so far exercised the success paths of the emulation subroutines. The error paths, especially the code that intercepted and modified access violations, required a different set of tests.

CPU Diagnostics

For each CPU designed by Digital, a set of CPU diagnostics is written that exercises as much of the central processor as possible. Included in these diagnostics is an instruction-set exerciser that tests for proper behavior in at least some of the interesting error cases. The CPU diagnostics for the MicroVAX I served as the primary test for the access violation handler in the decimal/string emulator.

AXE Verification Program

All new VAX computers at Digital are tested with an architectural verification tool known as AXE. AXE programs are used to determine whether or not the machine conforms to the VAX architectural specification. AXE accomplishes this testing by subjecting each VAX instruction, with many combinations of operands, to a variety of error conditions. These conditions include inaccessible operands, instructions or operands that cross page boundaries, and unusual operands.

When the MicroVAX instruction emulator was subjected to AXE testing, the only bugs that remained involved an instruction restart following an access violation.

Results

As a result of this strategy, the software emulation code was completed and fully debugged before the first real MicroVAX hardware was finished. The ULTRIX-32 and VAXELN operating system groups were able to take the VMS emulation code and convert it to work under their operating systems. That took much less effort than was required for the VMS development team to implement that code. With this technique, bugs found in the instruction-execution logic in one system could be corrected in all three operating systems.

A second benefit of this engineering effort was seen by the hardware designers. The revised VAX-11/730 microcode sources and microcode tools were further modified to create a MicroVAX CPU chip simulator. The simulator allowed the MicroVAX CPU boards to be tested before any MicroVAX chips were actually available.

The biggest gain of all was that no application software, compilers, or operating system code had to be rewritten to avoid the use of the missing instructions.

References

1. *VAX Architecture Reference Manual* (Bedford: Digital Equipment Corporation, Order No. EK-VAXAR-RM-002, 1983).

2. D.W. Dobberpuhl et al, "The MicroVAX 78032 Chip: A 32-bit Microprocessor," *Digital Technical Journal* (March 1986, this issue): 12-23.

3. *VAX Software, Languages and Tools Handbook* (Maynard: Digital Equipment Corporation, Order No. EB-27240-48, 1985).

4. W.R. Bidermann et al, "The MicroVAX 78132 Floating Point Chip," *Digital Technical Journal* (March 1986, this issue): 24-36.

Steven E. Boone
Guenter E. Schneider

The TK50 Cartridge Tape Drive

A streaming tape drive, the TK50 subsystem, provides fast backup and data transfer for small computers like the MicroVAX II system. A single-reel cartridge, using half-inch magnetic tape, stores 100 megabytes of data. A unique tape transport system automatically threads the tape when the cartridge is inserted. The drive reads and writes data in a serpentine manner, going the entire tape length first on one track, then another. For high data integrity, the TK50 subsystem employs a sophisticated error-recovery algorithm, reading data after writing it and rewriting any corrected data farther down on the tape. The Q-bus controller, the TQK50, contains complex firmware conforming to Digital's Storage Architecture and controlling data transfers between the CPU and the tape.

As the performance of computer systems expands while their size shrinks, many factors demand special attention. One major factor is storage systems. Over the past few years, disk drives have made dramatic advances, providing storage capacity of hundreds of megabytes in very small and relatively inexpensive packages. Since the predominant technology for today's disk drive is based on the fixed-media concept, some means of providing system backup and data transfer capabilities is required. Magnetic tape systems are still the most viable way of providing these capabilities.

Ease-of-use considerations require that a backup/transfer device be matched in capacity to the supported disk systems. It should also be extremely reliable, fast, and very cost effective. This paper describes a peripheral subsystem, the TK50 magnetic cartridge tape drive (Figure 1), that meets all these requirements.

Design Goals of the TK50 Subsystem

The TK50 cartridge tape subsystem was conceived to meet the needs of the MicroVAX II and similar computer systems. A study of tape products then available indicated that existing quarter-inch cartridge drives did not provide either the performance or the capacity required to back up the large capacity disk drives supported by these systems. Existing drives also lacked the reliability and data integrity required to complement the designs of our new microsystems. Therefore, Digital designed the TK50 cartridge tape subsystem to meet the needs of the MicroVAX II system and other small to mid-range computers.

A wide variety of factors defined the design goals of the TK50 subsystem. It had to fit into a standard 5 ¼-inch form factor and provide high capacity with high data integrity. The desire for mechanical simplicity, reliability, and low cost, while maintaining good performance, dictated a streaming tape design. The TK50 subsystem had to be compatible with the Q-bus, and the TK50 controller had to support the Tape Mass Storage Control Protocol (TMSCP) of the Digital Storage Architecture.

Our investigations led to the concept of an automatic-threading, single-reel cartridge that utilized the established medium of instrumentation tape. This tape supports high bit densities and fast tape speeds, allowing great latitude in specifying the performance and capacity of the TK50 subsystem. We also decided to use

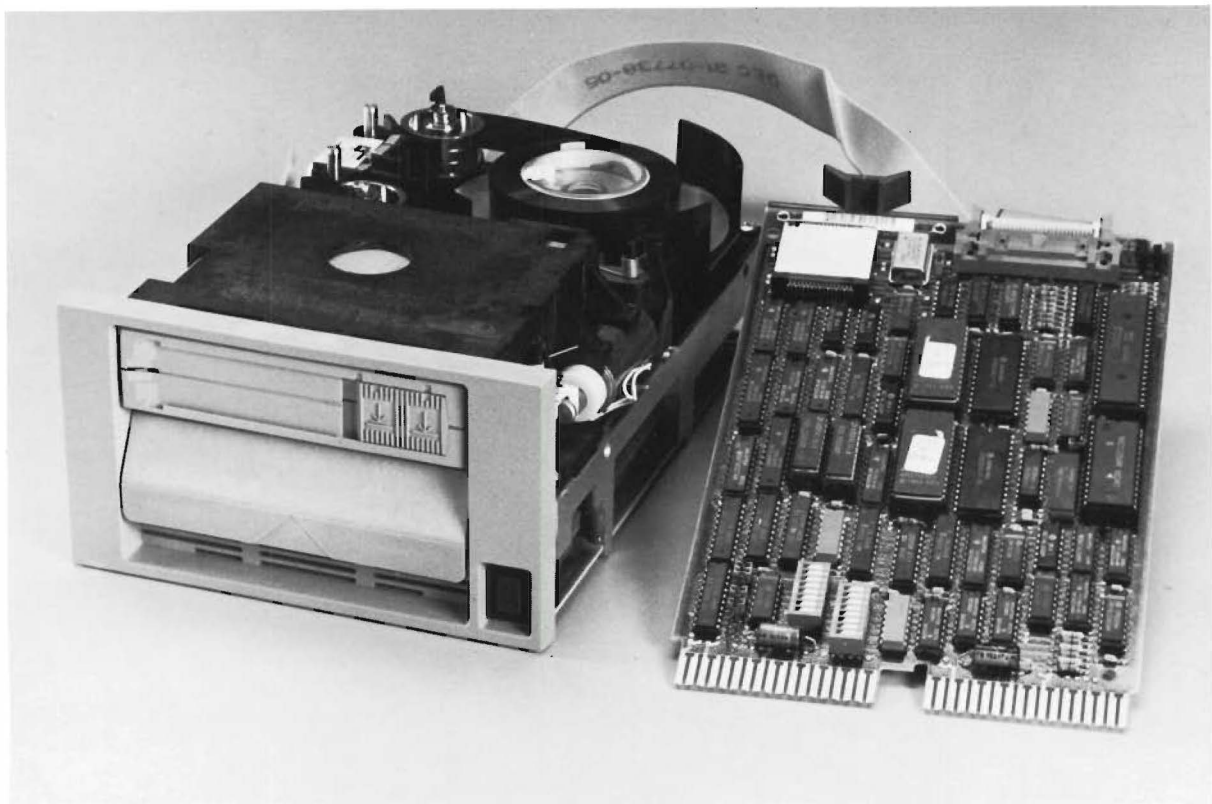


Figure 1 The TK50 Tape Drive

half-inch tape, rather than quarter-inch, to maximize capacity.

The requirement of the MicroVAX II system, as well as our desire to minimize risks in a first-generation product, dictated that the tape capacity should be 100 megabytes (MB).

System Design

The TK50 cartridge tape subsystem was developed with three major components:

- A tape cartridge, called the CompacTape Cartridge, that houses 600 feet of half-inch tape and supports the auto-threading feature of the transport mechanism
- A unique streaming tape transport featuring auto-threading and a microprocessor-controlled servo-system
- An intelligent, microprocessor-based Q-bus controller that supports TMSCP

CompacTape Cartridge

The CompacTape Cartridge is unique in many ways. First, it provides a large amount of data recording surface for its volume. The cartridge has approximately two hundred and fifty times the recording surface area of a single-sided 5 ¼-inch floppy disk. Moreover, compared to the only commercial tape product then available to fit the 5 ¼-inch form factor, the CompacTape Cartridge is four times as efficient in utilizing tape volume in relation to cartridge volume. The cartridge is designed to maximize the volume of tape in the standard form factor of the 5 ¼-inch drive. The cartridge, shown in Figure 2, contains a single reel with the tape occupying forty percent of the cartridge's volume. The tape is ½ inch wide, .001 inch thick, and 600 feet long.

Second, the CompacTape Cartridge is a completely enclosed device that never exposes the media to the environment, thus greatly enhancing the data reliability of the entire subsystem.

Third, the CompacTape Cartridge allows automatic tape threading once it is inserted into the TK50 tape drive. This auto-threading function is a key feature of the mechanical design of the tape transport.

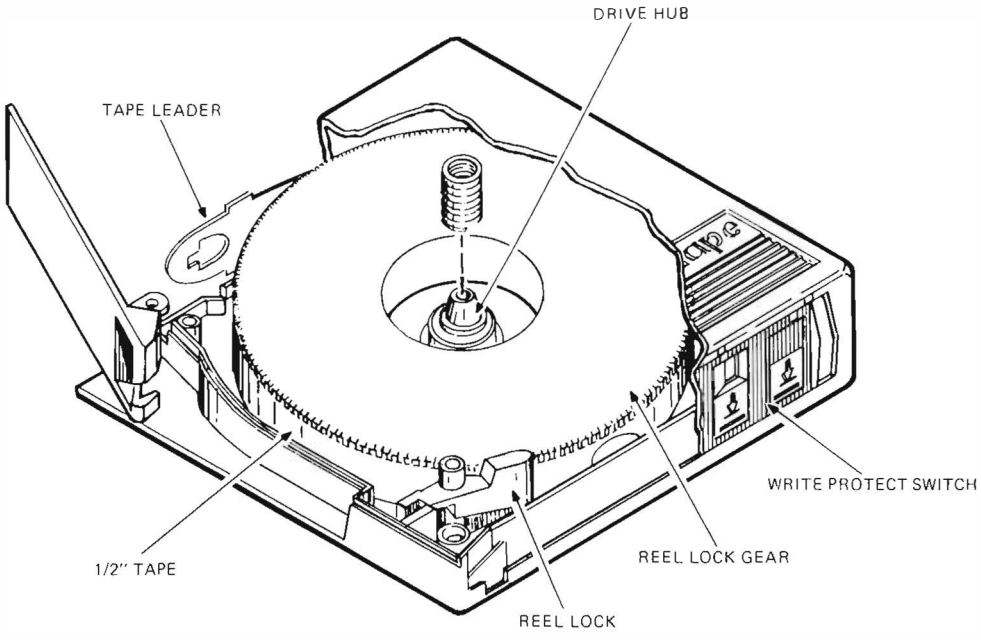


Figure 2 The TK50 Tape Cartridge

The auto-threading works in the following way. When a cartridge is inserted into the drive, the tape must be threaded around the tape guides, over the read/write head, around the take-up reel, and then fastened to the reel hub. Two leaders are used to accomplish the threading, as shown in Figure 3. One, made of .007-inch Mylar, is attached to the BOT end of the tape in the cartridge; the second is attached to the hub of the take-up reel in the drive. This second leader has an arrow-shaped tip that reaches from the reel, through the tape path, and into the area that will be occupied by the tip of the first leader when the cartridge is inserted. During the insertion process, the arrow-shaped tip is moved by a cam into the opening of the cartridge leader. Tension is then applied to lock the leaders together. This "buckle" is now ready to be pulled through the tape path and wound onto the take-up reel.

This buckling process is accomplished by two links in the drive, in conjunction with a constant tension applied by the motor to the take-up leader. One link uses a cam to move the two leader tips into each other. The other link holds the take-up leader in the correct position and retreats at the right instant, allowing the motor to cinch the buckle. The entire process

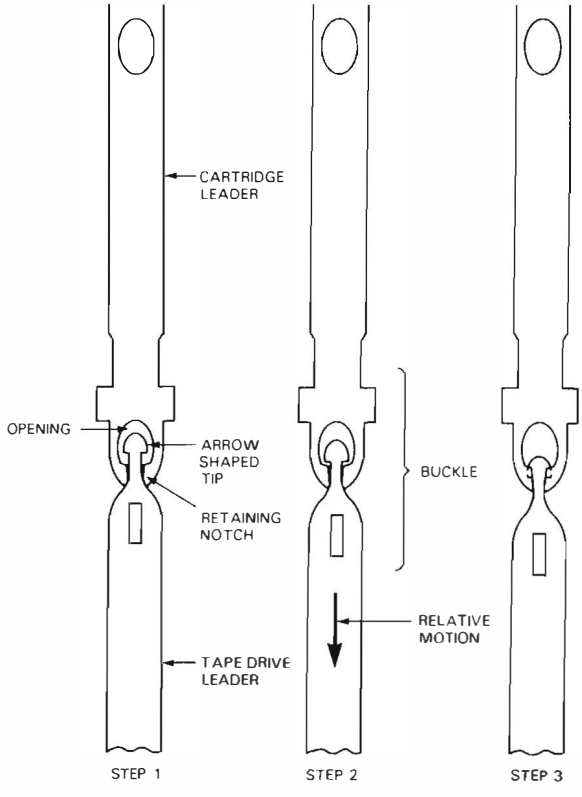


Figure 3 Engagement of Drive Leader to Cartridge Leader

happens during the last half-inch of insertion as the cartridge enters the drive. (See Figure 4.) This linking takes place without any tape being spooled out of the cartridge.

When the tape is rewound into the cartridge for removal from the drive, the two ears on the cartridge leader come to rest in a pocket in the cartridge shell. When the cartridge is removed from the drive, two opposing locks hold the reel in this position. The toothed locks engage with the teeth on the outer diameter of the reel flange. Thus locked, the tape stays tightly wound and the leader tip is kept in the correct position for a subsequent buckling process.

Tape Transport

The TK50 tape transport (Figure 5) consists of two major components: the tape drive and a single printed circuit board assembly.

The tape drive encompasses the mechanical and electromechanical components to read data from and write data to the magnetic tape. The drive's major components include

- The magnetic read/write head and its linear positioner

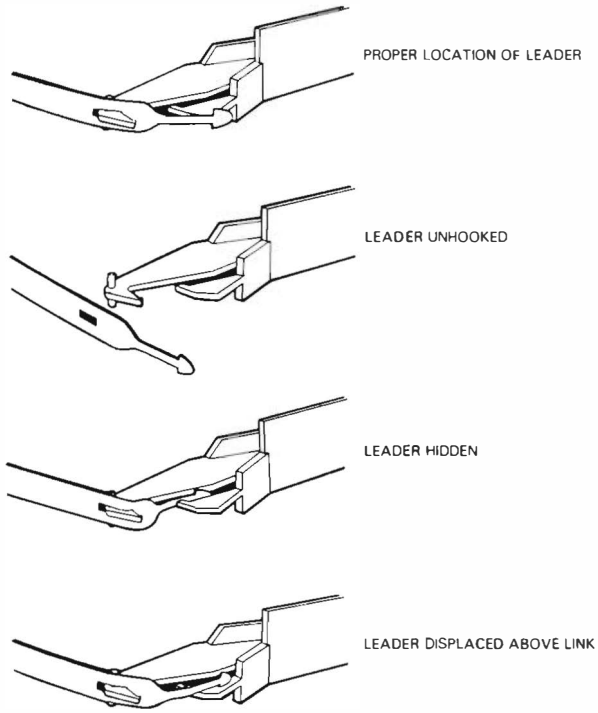


Figure 4 View of Leader Shown in Four Positions

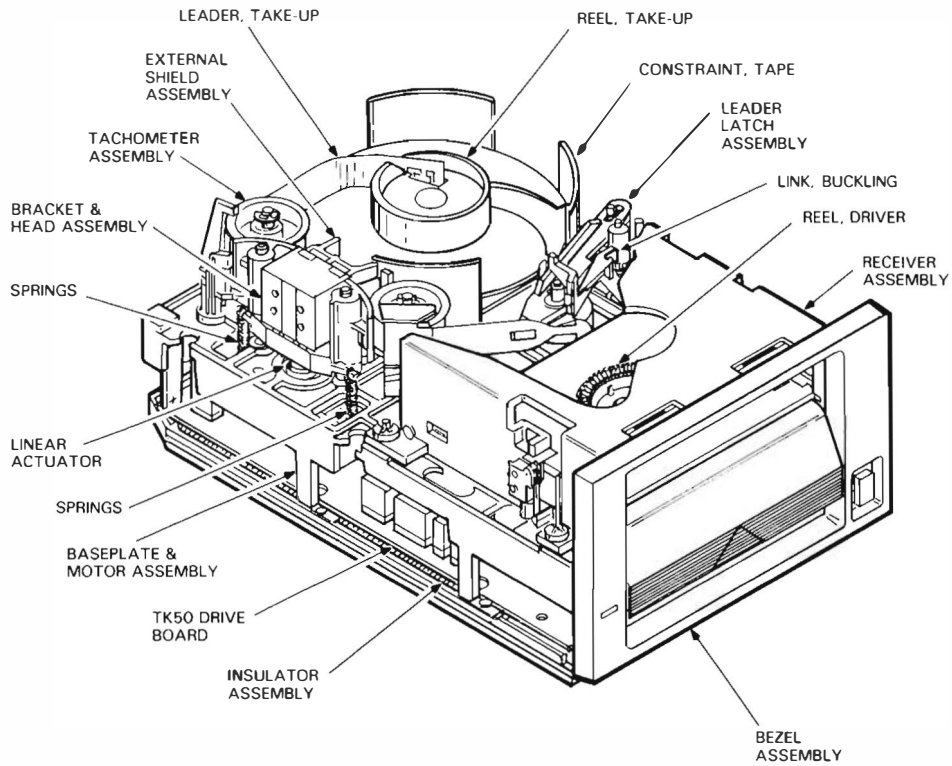


Figure 5 TK50 Tape Drive Transport

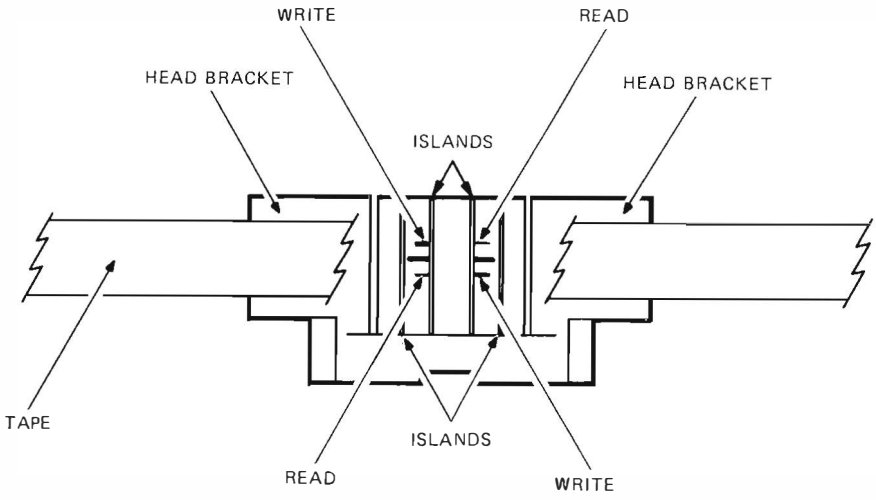


Figure 6 TK50 Read/Write Head (Top View)

- The cartridge threading mechanism
- The take-up reel and its motor
- The drive hub mechanism, which interfaces to the CompacTape Cartridge, and its motor
- The tachometer, which provides feedback to a microprocessor, the 8051, for tape speed control
- Various sensing devices that monitor and control the handling of the tape as it passes over the read/write head

Read/Write Head

The read/write head is designed with four islands that are in contact with the tape (Figure 6). The tape forms a polygon as it contacts these four areas. Each island bends the tape by an angle of six degrees. Over its width, each island is curved by an amount corresponding to the radius of the natural curvature of the tape under working tension, thus assuring good surface contact (Figure 7). The narrow islands limit any temporary liftoff (due to contamination) to very short sections of tape, and they clean the tape as well.

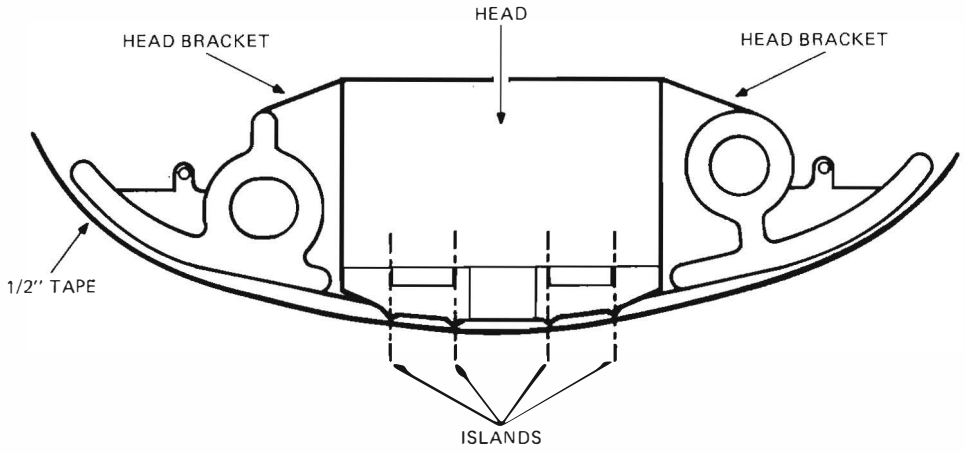


Figure 7 TK50 Read/Write Head (Side View)

Except for the ferrite cores, the entire head block is made of ceramic material to ensure long life. The two inner islands contain the read/write cores; the two outer ones direct the tape to the inner ones so that uniform contact between the tape and the head is provided. On the upper part of the head assembly are two gaps, a write gap (.018 inch wide) followed by a read gap (.008 inch wide), that read and write data when the tape is moving forward. Two corresponding lower gaps read and write data during reverse tape motion. The lower gaps cover the odd tracks and the upper gaps cover the even tracks; thus, the head has to traverse only half the tape width, helping greatly to keep the height of the drive within limits. The track spacing is .019 inch.

Auto-Threading

As the cartridge is inserted, its door opens, exposing the cartridge leader. Then, as described earlier, two plastic arms in the drive act to buckle the cartridge's supply leader to the drive's take-up leader. The rest of the auto-threading process is handled by the drive's motors, sensors and microprocessor.

Tape motion and tension control is accomplished through two microprocessor-controlled brushless direct-current motors. One of these motors is connected directly to the take-up hub; the other to a drive hub designed to interface to the CompacTape Cartridge.

The engagement of the cartridge hub with the drive motor shaft is accomplished by a pair of gears that transmit torque and simultaneously center the reel (Figure 8). A plastic hub with one set of teeth is attached to the spindle; another set of teeth is molded on the underside of the cartridge reel hub. A clutch gear engages both sets of teeth to drive the reel. To facilitate the insertion or removal of the cartridge, the clutch gear is axially retracted out of engage-

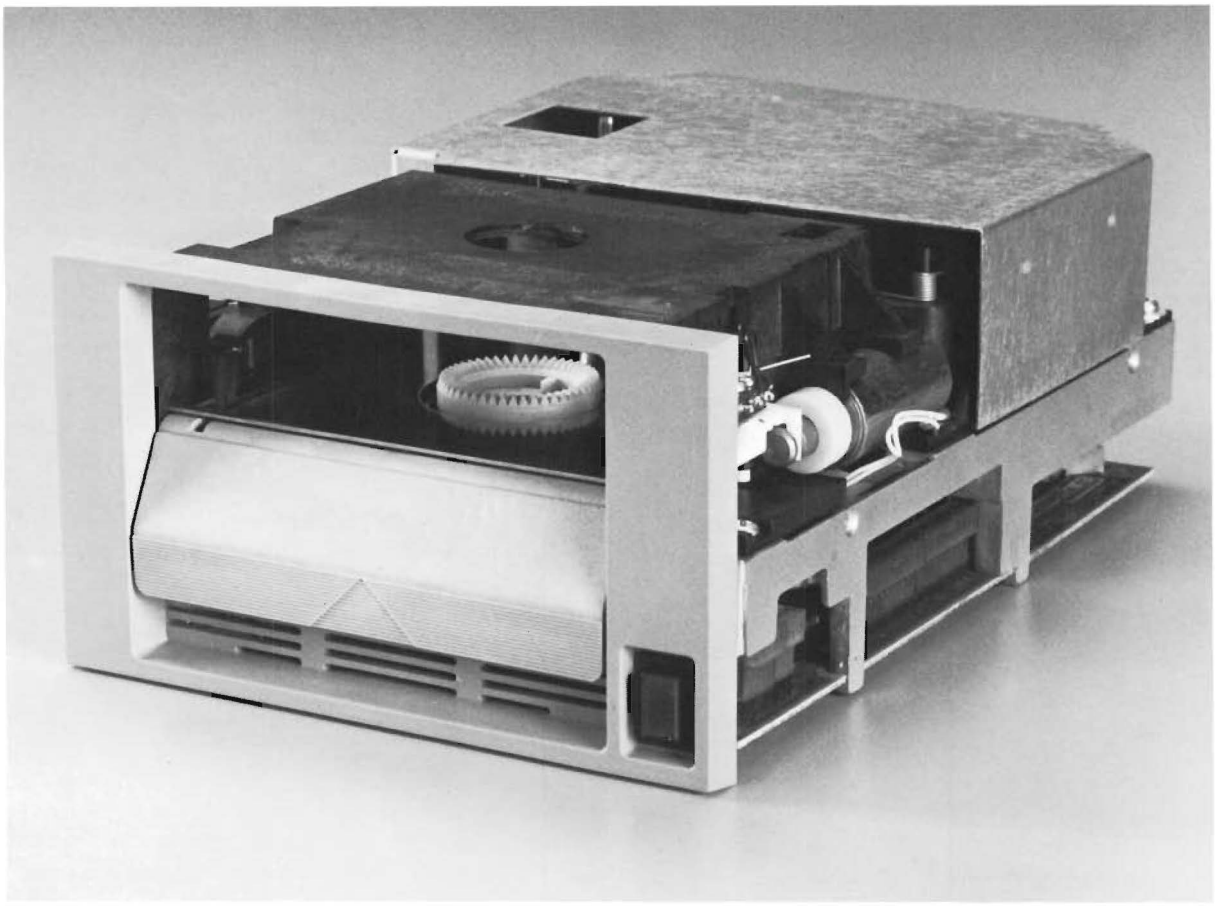


Figure 8 TK50 Door Assembly

ment. The clutch gear is activated by the operator's lowering or raising the handle. When the handle is lowered, the spring-loaded lower gear engages the reel and lifts it slightly into the cartridge to eliminate contact between the rotating reel and the stationary shell (Figure 8).

This clutching arrangement has a big advantage because it allows mechanical simplicity and easy operation of the drive. The cartridge is inserted by the operator into a channel (receiver) that puts the two leaders into a coplanar relationship. The entire linking process is thus accomplished by merely sliding the cartridge into the receiver slot. A solenoid-activated interposer locks the cartridge in place when it reaches the end position in the receiver. When the front handle is then lowered, the drive gear rises to mate with the cartridge reel. A set of fingers simultaneously enters the bottom of the cartridge to release the reel locks, thus allowing the tape to move. The operator accomplishes all these actions with one hand.

After a tape cartridge is inserted into the TK50 drive, the operator presses a button and the 8051 microprocessor on the printed circuit assembly initiates the threading process. The reel motors, under microprocessor control, slowly put tension on the tape to accomplish the process. The buckled leaders and a length of tape are pulled through the drive and onto the take-up reel. Auto-threading is complete when the BOT hole in the tape is detected by a photo-transistor. When the auto-threading operation ends, the microprocessor will have received pulses from a tachometer attached to one of the rotating tape guides. Through the information derived from the tachometer, the microprocessor can maintain proper tension and tape speed.

After the tape is positioned at BOT, the controller requests a calibration procedure. This procedure sets up the drive to ensure that proper values for the read circuitry gain and head stepper alignment are obtained. This calibration provides one of the key features of the TK50 subsystem: the ability of a user to exchange media between different TK50 tape drives without the need for adjustments.

Once calibrated and at BOT, the TK50 drive is ready to read or write data. The drive writes data in a serpentine fashion over the entire length of the tape. The upper part of the read/write head writes data on one track down the entire length of tape until it reaches a logical EOT marker. (The logical EOT marker is a preset tachometer count; the physical EOT marker is a hole in the tape.) The tape direction is then reversed and the other lower write core will write data in the other direction for the entire length of the tape until a logical BOT is reached. The direction of the tape is then changed to forward, the head is stepped up by 19 mils, and the upper write core is again used to write data. Figure 9 illustrates the physical tape configuration.

Drive Circuitry

The printed circuit board assembly is built around an 8051 microprocessor. The 8051 and associated circuitry provide the intelligence to interpret commands, provide servo control for the reel motors, perform tape calibration procedures, and monitor various status inputs. The read/write circuits necessary to translate data to and from the tape's MFM format also reside on the board. Figure 10 illustrates a simplified block diagram of the TK50 drive board.

Write data comes into the drive's logic board via the differential signal cable from the controller board. The data enters the shift register, which accepts the serial data and outputs a five-bit parallel data pattern into a programmable array logic (PAL) device. The data is clocked through the shift register by a 500-KHz clock. (500 KHz is the write pulse rate, or data rate.)

The PAL first accepts the five parallel bits from the shift register. Then the PAL generates the pre-compensation, as required, and translates the data into the MFM format recorded on the tape. A constant current source of 15 milliamps is applied alternately to each core of the active write head, resulting in the flux transitions necessary to write data on the tape.

To enhance data reliability, the TK50 subsystem reads data just after writing it. This technique uses the read head (positioned immediately behind the write head) to read the data from the tape as soon as it has been written. (See Figure 7.)

The read data is sent back to the controller, where the communications interface performs CRC processing. If an error is detected, the controller rewrites the block that contained the error. The rewritten block is placed farther

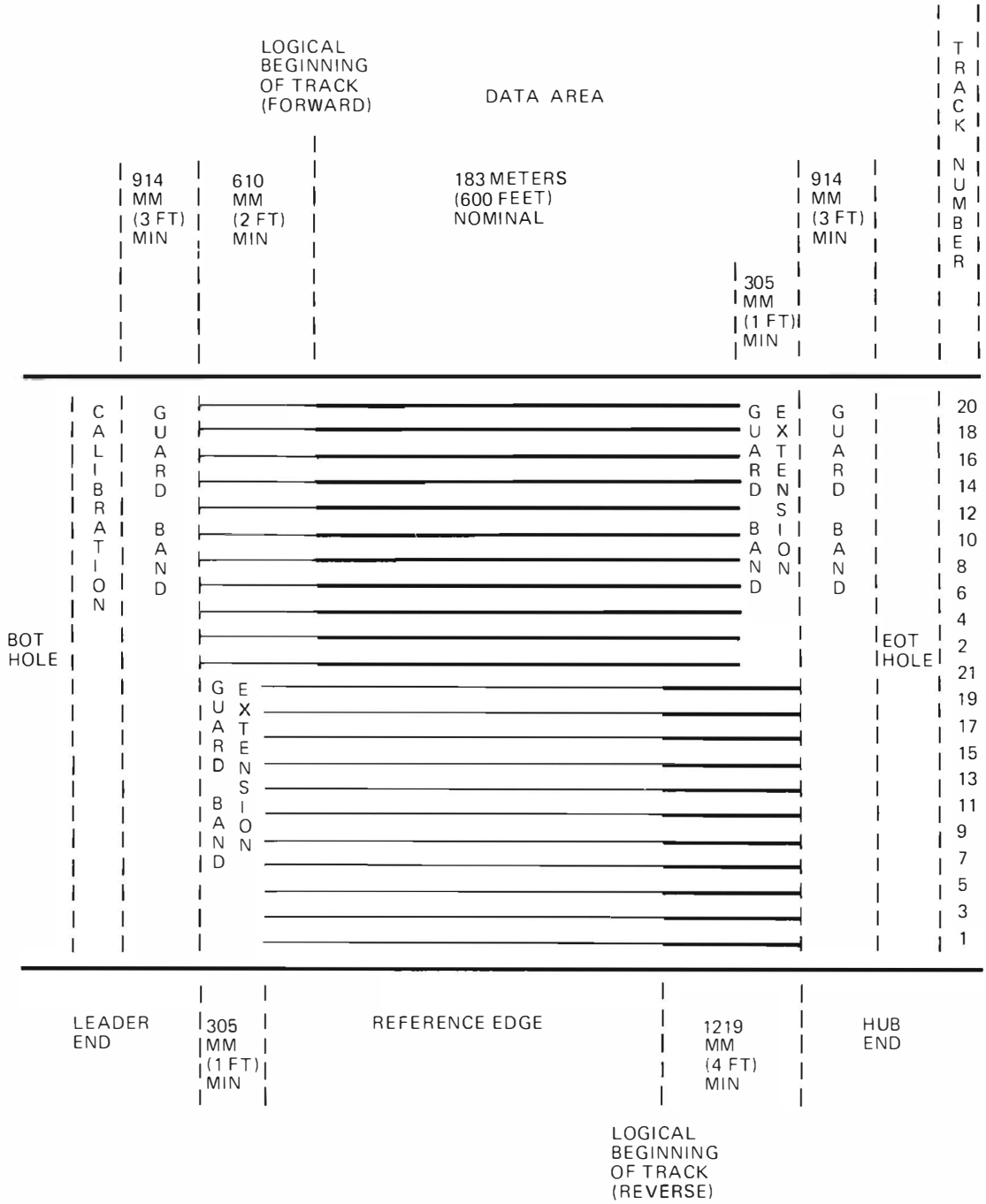


Figure 9 Physical Tape Configuration

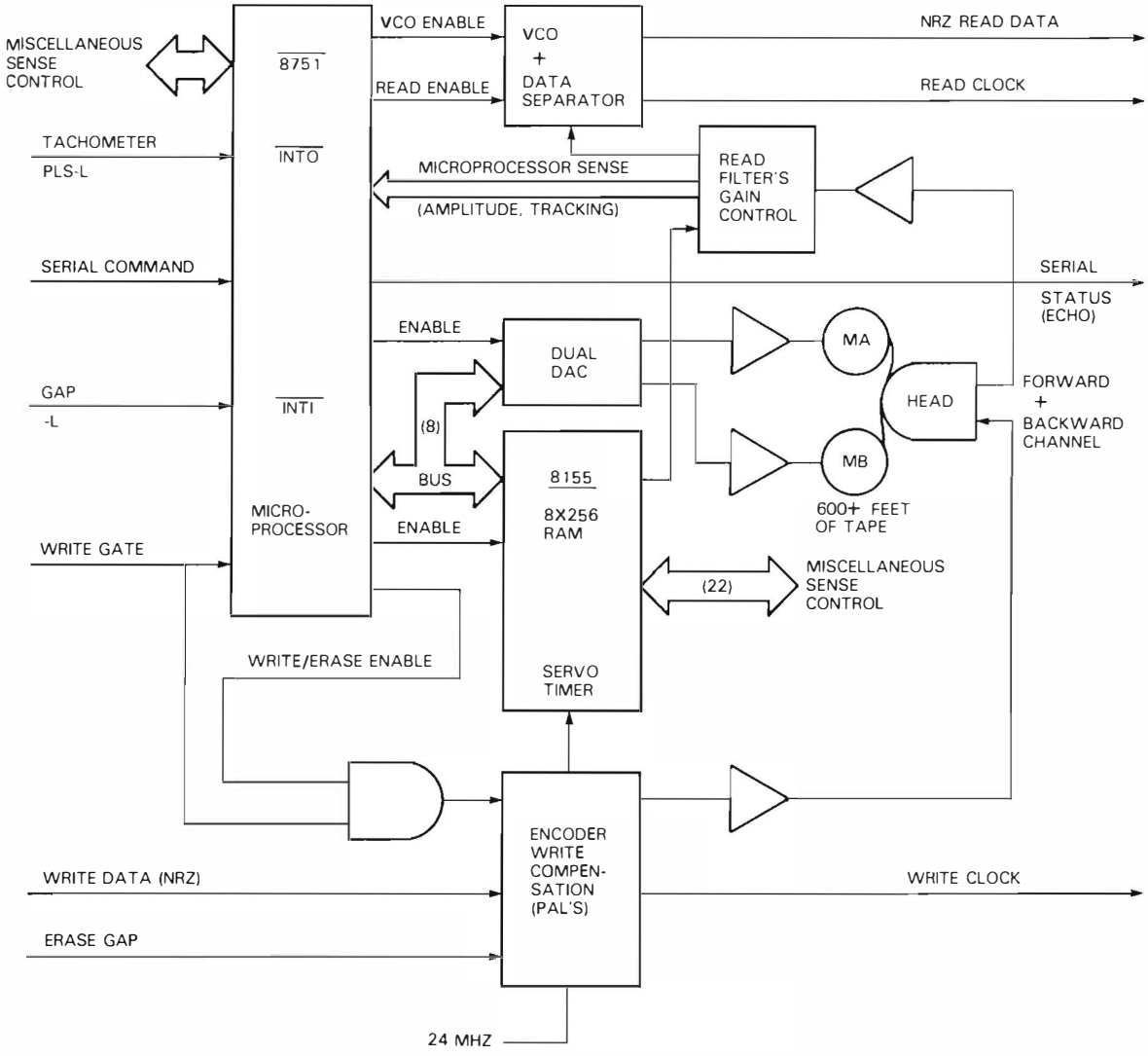


Figure 10 Block Diagram of the Drive Board

down on the tape to avoid the performance loss resulting from the drive's having to move the tape back and rewrite over the data block containing the error. The controller firmware is able to detect these rewritten blocks during a subsequent read pass for data recovery procedures, thereby enhancing system integrity.

Read data signals from the read head are fed to the differential preamplifier circuit and in turn to the read amplifier. The gain of the preamplifier is automatically set during calibration to maintain an optimum signal level. The signal from the read amplifier is then passed to a differentiated, linear-phase, low-pass filter. A zero-crossing detection circuit produces a digital signal, consisting of a single pulse for each detected zero crossing, that represents data read from the tape.

The digital data is then sent to the phase lock loop (PLL) circuit where the clock signal is recovered and the MFM data is decoded. The PLL consists of two PALs, a voltage-controlled oscillator, and some analog circuitry.

The read-data pulse from the read amplifier circuit is used in conjunction with the 500-KHz write clock to optimize the "lock time" for the PLL. Whenever there is a gap (no signal) going into the PLL, it will lock onto the 500-KHz clock signal. This locking is done so that the loop-filter integrating capacitor is kept at a constant voltage. This process minimizes the phase-lock time during the preamble.

When the READ ENABLE signal is asserted, the PLL waits for the synchronization (sync) bit. When the PLL detects the transition, it clocks the sync bit and data onto the serial line to the controller and starts sending back the read clock. The sync bit signals the communications processor on the controller to start processing the following data and the CRC check-word, and to check for a matching CRC.

Q-bus Controller

The intelligent interface between the TK50 tape transport and the Q-bus is designated as the TQK50. Figure 11 is a block diagram of the

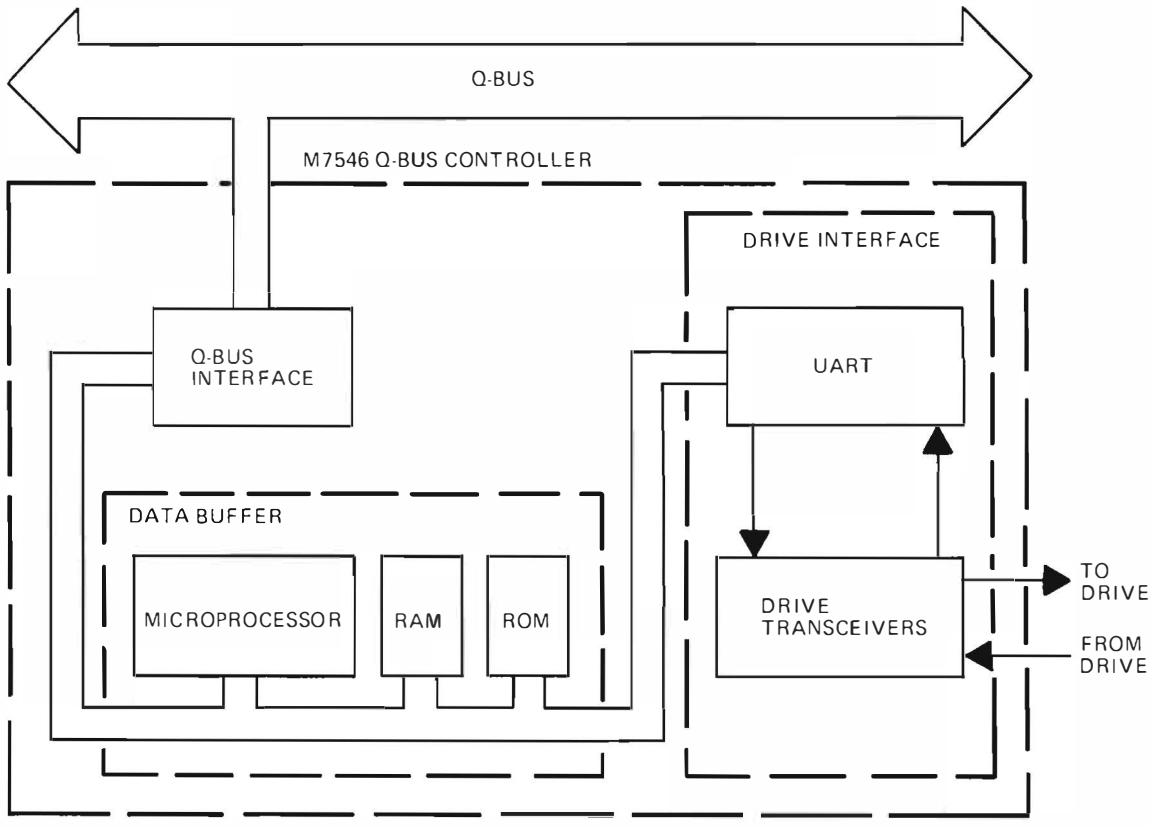


Figure 11 Block Diagram of the TQK50

TQK50. The interface is a Q-bus-compatible dual board based on the 80186 microprocessor. In conjunction with 32 kilobytes (KB) of highly complex firmware, the 80186 and its associated hardware perform the following functions:

- Interface the controller to the Q-bus (via single-word and DMA transfers)
- Translate and process TMSCP command packets and responses
- Provide data format and error recovery processing
- Control the general operation of the tape transport mechanism
- Support the serial data link between the controller and drive

Hardware

The Q-bus interface is controlled primarily by an 80186 microprocessor and an 82S105 field programmed logic sequencer (FPLS), which is a high-performance LSI device capable of performing complex logic functions. Using the 82S105 FPLS sequencer allowed us to create an efficient, flexible design in a very small space. The FPLS and microprocessor are responsible for maintaining the strict Q-bus protocol during DMA transfers to and from the controller. The DMA transfers and interface interrupts are processed very quickly due to the high performance of the microprocessor and FPLS. This high performance makes possible the data rates needed to support tape streaming and lessens the criticality of the DMA latencies in the host system.

Assisting the FPLS is an 80186 microprocessor operating at 6 MHz. The 80186 is a highly complex, 16-bit microprocessor; it is responsible for all the command, control, and data processing for the TQK50. A microprocessor with the 80186's performance is required due to the large number of complex tasks that must be performed within very short time frames (e.g., ECC processing during inter-block gaps on tape). The high level of integration available with the 80186 was a key factor in its selection. In addition to the CPU, the 80186 contains three onboard timers, an interrupt controller, address decoding, and two DMA channels. Also important in the selection of the 80186 was the availability of sophisticated development tools and efficient software support packages.

The 80186 microprocessor is supported by numerous components that include SSI, MSI and PAL devices. Furthermore, the program store and the workspace/data buffers are provided by 128-kilobit (Kb) EPROMs and 64Kb static RAMS. A total of 32KB of program store and 16KB of buffer is available to the 80186.

Communications between the TQK50 controller and the TK50 tape transport take place over a pair of full-duplex, differential, serial lines. A multiprotocol communications processor (NEC 7201) is used to process the serial-to-parallel and parallel-to-serial conversions. One full-duplex channel, operating at 187.5 kilobaud, communicates the command/status information between the controller and the transport. The other channel provides the data communications path, supported by data-link error checking via CRC-16. This second channel operates synchronously at 500Kb per second. The NEC 7201 communications chip supports DMA transfers to and from the 80186 and operates in a priority-interrupt mode.

Firmware

The most complex component of the TK50 subsystem is its firmware. The 32KB of firmware contained in EPROM are partitioned into five major functions:

- The PORT/Q22 (Q-bus) for data transfer control
- The SERVER for TMSCP command processing
- The TOS for tape transport control and formatting
- The ECC for error detection and correction
- The ROD for resident onboard diagnostics

The PORT/Q22 firmware controls data transfers between the controller and CPU, and also maintains the command queue processing. Up to four TMSCP commands can be queued, allowing the host to set up a series of operations for execution while it continues with other processing. DMA transfers of up to 64K-1 bytes can be made, allowing an effective, low-overhead data transfer between the subsystem and CPU memory space.

The SERVER firmware is responsible for translating and executing the wide variety of TMSCP commands. These commands provide a very structured environment within which control,

status, and data transfers are accomplished. TMSCP is a packet protocol that uses a command-response sequence. Each pair of command-response packets contains information pertaining to the internal command as well as various command modifiers, status fields, and subsystem parameters. All levels of information, from the command sequence number to command status to hardware and firmware revision levels, are provided in TMSCP. In addition to assembling and processing this information, the SERVER firmware uses values, such as physical and logical record numbers, to validate information being processed from the tape.

SERVER has an additional mode that supports the Diagnostic Utility Protocol (DUP). DUP provides a set of commands that allow detailed tests of the subsystem to be performed. DUP operates in conjunction with the resident onboard diagnostic module.

The TOS (tape operation support) firmware controls the transfer of data between the tape transport and the buffers allocated by SERVER. This control is accomplished through formatting operations and through physical control of the tape transport mechanism.

The TK50 subsystem is a streaming tape drive that was designed to operate in an efficient block-mode environment. The TK50 subsystem relies on logical information written on the tape to determine the tape's physical and logical positions. The physical and logical contexts are maintained by the TOS firmware and written into special control fields embedded in the TK50 tape format. Information contained in these fields includes physical object number, logical object number, tape-mark number, byte count, sequence control number, track number, and block type. This information is processed by TOS to maintain the physical and logical contexts between the subsystem and the data on the tape.

During streaming operations, context processing is the primary function of TOS. However, when the host system is unable to process data at a sufficient rate to maintain the streaming operation (45KB per second), TOS must provide complex positioning control. Whenever the host system falls below the required data transfer rate, TOS must stop the tape. Since the TK50 subsystem was mechanically optimized for streaming, any stopping and starting of the tape is a time-consuming and imprecise operation. Moreover, the TK50 subsystem lacks the inter-record gaps that are used for positional information in traditional 9-track tape drives. The TK50 subsystem must rely on data read from the tape to locate its position.

When the host system resumes data processing, TOS must reposition the tape by a sequence of reverse, stop, forward, and read. After locating the last data block processed on the tape, TOS continues with the host's request. The host's failure to process data at a sufficient rate is costly in terms of system throughput. This situation requires increased complexity in the subsystem design.

TOS provides a padding function to help compensate for insufficient host processing power. With padding, TOS allows data latencies of up to 63 milliseconds before reverting to the repositioning mode. During this data latency period, pad blocks are written to the tape in 9-millisecond increments. That allows the tape to continue streaming. The trade-off is improved performance at the expense of slightly reduced tape capacity (512 bytes per pad block). If the 63-millisecond period is exceeded, TOS stops and performs a reposition to the point of the last data block. When additional data arrives, TOS overwrites any previously written pad blocks. In practice, this pad function enhances performance and seldom reduces tape capacity by more than ten percent.

The ECC firmware provides the means to detect and correct errors. To provide a high level of reliability, the TK50 subsystem is designed to allow only one unrecoverable error in every 1×10^{11} bits read. This is equivalent to one unrecoverable error in every 125 cartridge reads. To achieve this goal, ECC implements error-detection and error-correction schemes. Error detection is based on the CRC-16 method, which is supported by the hardware communications device. This industry-standard method has been proven to be very efficient in this environment.

To implement the error-correction function, ECC processes serial-formatted data to and from the tape. Data is written to and read from the tape in 512-byte blocks. Each block is grouped into 8-block units, called data entities. Within an entity, the four even-numbered data blocks (0,2,4,6) and the four odd-numbered blocks (1,3,5,7) are protected by longitudinal checksum blocks. An entity, therefore, consists of ten blocks: data blocks 0 through 7 and ECC blocks

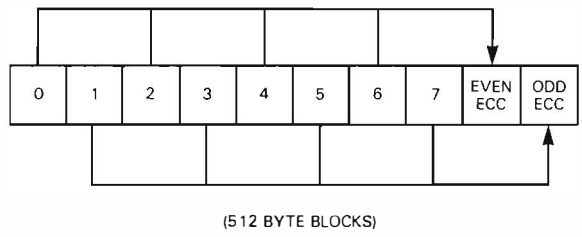


Figure 12 Entity of Ten Blocks

8 and 9. Figure 12 shows the arrangement of the ten blocks.

This technique, coupled with record-level checking by SERVER and the host operating system, insures the complete integrity of the user's data.

The ROD (resident onboard diagnostics) firmware provides additional support for the TQK50. When the subsystem is initialized, the firmware executes a series of go/no-go tests that validate the functionality of the controller. Ninety-eight percent of the TQK50's functionality is covered by these tests, excluding the Q-bus and drive-interface logic circuits. More extensive diagnostics that fully test the TK50 subsystem are available under the DUP. Having the diagnostics resident in firmware allows the running of integrated tests that interact at levels not permitted from the system interface. That avoids the difficulties in supporting down-line loadable code in various run-time environments.

Summary

Designing the TK50 cartridge tape subsystem and turning it into a product was a significant challenge. The effort proves that good performance, high reliability, ease of use, and extraordinary data integrity can be achieved in a cost-effective manner. These qualities will continue to be required as computer systems increase in performance and capacity.

To that end, the TK50 cartridge tape subsystem is but the first of a family of cartridge tape products. Work is continuing on the development of subsystems with higher performance and greater capacities. Interfaces to computer systems other than those based on the Q-bus have been or are being developed to meet the expanding needs for greater storage capacity.

Acknowledgements

Designing the TK50 cartridge tape subsystem required a multitude of disciplines involving scores of individuals. Each member of the TK50 program team contributed time, energy, and personal commitment to yield a successful product. The authors wish to acknowledge those contributions here.

Raymond J. Lanza

Porting ULTRIX Software to the MicroVAX System

The ULTRIX system, written in the C programming language, was ported to the MicroVAX II processor by a multistep process. This involved establishing a cross-development environment, building a bootpath, porting the ULTRIX kernel, and writing special device drivers. The remaining software was ported after those steps were completed. To minimize ULTRIX design changes, the system's I/O architecture was mapped into the MicroVAX physical address space so as to mirror the equivalent mapping on larger VAX systems. Some MicroVAX instructions must be emulated in macrocode. The emulator used in the MicroVMS software was adapted for use in this ULTRIX software.

The UNIX system came into existence in 1969 at the AT&T Bell Laboratories in Murray Hill, New Jersey. The initial system was written in assembler and ran on a PDP-7 system that was loaded from paper tapes. From late 1970 to early 1971, the UNIX software was reimplemented for the PDP-11 system using a cross-assembler running on the original PDP-7 system. In 1973, the kernel was rewritten in the C programming language. Since that time the system has undergone many changes and is still the subject of much research.[1] Today, there are two major 32-bit variants of the original software: 4BSD, developed at the University of California at Berkeley; and System V, from AT&T Corporation. Digital Equipment Corporation's original ULTRIX-32 product is a direct descendant of 4.2BSD.

In 1983, Digital decided to develop and distribute a UNIX software product. At that time, 4.2BSD was the only virtual-memory UNIX operating system running on VAX processors. It is still the only UNIX software derivative to provide network support. These features were the key factors in deciding to use 4.2BSD as the basis of the ULTRIX-32 system.

Development started in the fall of 1983 on one of the first 4.2BSD distributions, and the final product was released in April 1984 as ULTRIX-32 V1.0. In the current version of the product, we have combined the two UNIX system derivatives by adding the system services of the AT&T version to the original ULTRIX-32 system. To that base we have added reliability and maintainability features, as well as new-processor support. The resulting system, one of the industry's most powerful and versatile UNIX software versions, spans the full VAX system price/performance range.

Porting the UNIX System

"Porting" is the process of implementing an operating system on a new processor. The UNIX system has been ported to more processors than any other system in existence. It runs on all classes of machines, from 8086 microprocessors to the CRAY-2. For VMS and RSX systems and the like, porting normally means a major rewrite because significant parts of them are written in low-level languages, usually macro assembler. Rewriting one of these systems is so expensive that either the effort would not be undertaken or the new system would be written from scratch.

The UNIX system is different. It is written in a single high-level language, C,[2] and has been

structured to be as processor independent as possible. However, vestiges of its PDP-11 heritage are still apparent.

All 32-bit versions of the ULTRIX-32 system are built from a common set of source files. The kernel files are organized into machine-dependent and machine-independent parts. The differences between the VAX and MicroVAX versions of the system are resolved through the use of conditional compilation and linking. The present kernel sources for the MicroVAX version are as follows:

Files	Language
209	C headers
315	C source
21	Assembler source

The 21 assembler source files can be further broken down as follows:

Files	Purpose
14	MicroVAX subset and floating point emulator
3	Templates for rpb,scb,spt
3	Macro definitions
1	Initial startup code (locore.s)

The last and most significant file is locore.s, which contains the initial startup code and a few critical routines needed for process management.

Bringing the UNIX system up on a new processor is normally done in multiple steps by a small team. The difficulty and extent of the work involved is directly related to the architectural differences between the versions for the existing and target processors. Our team consisted of three people, later joined by a fourth. The first was responsible for the compiler and subset emulator. The second did the software installation and verification for the first version of the product. Later, he was responsible for some device drivers. The author of this paper did the kernel port and other device drivers. The fourth person assumed responsibility for installation.

Bringing the ULTRIX-32 system up on a processor involves the following steps:

1. Establish a cross-development environment.
   - C language
   - Native assembler
   - Linker
   - Debugger
2. Build a boot path.
3. Port the kernel and a few key programs.
4. Write special device drivers.
5. Port the rest of the system.

The Cross Development Environment

When porting to a new architecture, it is necessary to develop a set of tools that produces code for the target system. These tools constitute a cross-development system for software generation and often become the basis for the eventual native environment. Their construction is normally the first step in the porting process. In the MicroVAX case, the cross-development tools were not necessary, for reasons explained below.

The MicroVAX system is a subset architecture with the majority of the string manipulation instructions missing.[3] MicroVAX systems can also be configured without floating point support in the hardware. Our challenge, which was also shared by the VMS and VAXELN Development Groups, was to provide an execution environment for user programs that was completely compatible with larger VAX systems.

By closely examining the instructions produced by our C compiler, we found that, with the exception of the floating point instructions, not one missing string instruction was created. Further examinations revealed that the only place where any of the missing instructions were used was in a handful of output formatting routines. As an interim solution, the affected routines were rewritten to eliminate the missing instructions.[4]

The Boot Path

MicroVAX systems contain the virtual memory boot (VMB) program in ROM. Normally this program loads the VMS system but has been enhanced to perform an alternate initial program load operation, called a boot-block boot.

This operation is the mechanism used to boot the ULTRIX system and is based on block number 0 of the boot disk being in a special format. Booting is a multistage process.

1. VMB first checks for an ODS-II file structure.[5] In the default case, VMB will perform a "sniffer boot," which consists of first checking the removable media, then the fixed disks, and finally the Ethernet. The system can also be booted from the TK50 cartridge tape drive and a special PROM board.

2. If an ODS-II file structure is not present, VMB looks for a valid boot-block image in the first block on the disk. This block contains a table that specifies the size and location of the secondary boot image. If the table is valid, VMB reads the secondary boot image into memory and transfers control to the image. (If the table is invalid, control is transferred back to step 1 above.)

3. The secondary boot image on ULTRIX systems is a program that locates, reads, and executes the tertiary boot program from an ULTRIX file system. The functionality of the secondary boot is severely constrained because it resides outside the file system in a fixed-size (7.5KB) area adjacent to the boot block.

4. The tertiary boot is capable of loading and running other programs. Unlike the secondary boot program, it supports interactive terminal I/O and can prompt the user for an alternate program to load. As a default, the tertiary boot loads the operating system kernel, called vmunix,[6] from the boot disk.

5. After the steps above have been completed, the kernel is in memory and ready to run.

The two boot programs are part of the stand-alone system, which in itself constitutes a porting problem that is not very different from porting the kernel. The problems encountered are similar, although simplified, because the stand-alone system runs with the interrupts and memory management disabled. The stand-alone system is not nearly as flexible as the kernel.

Porting The Kernel

The VAX Architecture Standard (Digital Standard 032) specifies the VAX instruction set, memory management, and process environment. However, the standard leaves many other areas open for change. These areas are typically ones that need to be supported on each new processor. For the MicroVAX system, it was necessary to address problems in the following areas:

- Startup code
- I/O architecture
- Console support
- System clock
- Missing instruction emulation

Initial Startup Code

After the kernel is loaded into memory, control is transferred to the initial startup code. This is entered with the processor interrupt priority "raised" to disable the interrupts, and with memory management turned off. The code sets up the memory management system and then "handcrafts" the processor to run the first VAX process. The majority of this code is located in a single assembly language file, called locore.s. In the case of the MicroVAX system, the instruction emulator and several changes to the I/O system required special mapping support during startup. (This support is discussed in the last section of this paper.)

In addition to the startup code, locore.s contains time-critical routines that use the VAX process-management instructions. Some of them contain a casel instruction based on the processor type for processor-specific operations. Those routines had to be extended to include the MicroVAX processors.

I/O Architecture

VAX processors do not contain I/O instructions; instead, device and device adapters exist in various sections of the physical address space of the processor. The control and data registers for these adapters appear as memory locations and are accessed using normal instructions. A key element of system software for any new processor is support for these devices and their associated address spaces. As an example, the physical address space of the VAX-11/780 system is pictured in Figure 1.

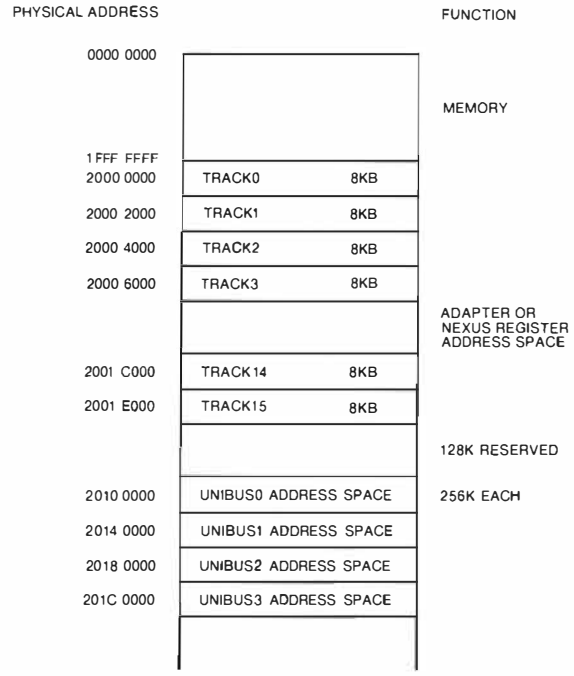


Figure 1 VAX-11/780 Physical Address Space

Each of the UNIBUS spaces can be further broken down as shown in Figure 2.

The physical address space of the MicroVAX II system is somewhat simpler, as depicted in Figure 3.

With the exception of the memory sections, the address spaces of the two processors appear to be very different. In fact there are a surprising number of similarities, as shown in Table 1.

The NEXUS space is where the adapter control and status registers reside. In the case of a UNIBUS adapter, the registers that control the mapping from the bus to main memory are located in the NEXUS space. The equivalent MicroVAX area, called local register space, also contains the mapping registers for the Q-bus to main memory.

These physical address spaces are eventually mapped into virtual addresses through entries in the VAX Page Table. The result is pictured in Figure 4.

One development goal that we set for each new processor support project is to minimize the changes necessary in the operating system. In the case of the MicroVAX II system, we examined the differences in the physical address spaces between that system and larger VAX systems. Although the names, sizes, and positions were different, they are functionally equivalent on both the small and larger systems. As a result, we "coerced" the local register space into the NEXUS map, and the Q-bus memory and I/O spaces were arranged to look like a large UNIBUS adapter. With this approach we were not forced to drastically alter the kernel's view of the machine, thus minimizing changes to other portions of the kernel.

A similar situation existed with respect to the Q-bus map. A device installed in a UNIBUS adapter sees an 18-bit address for a 256KB

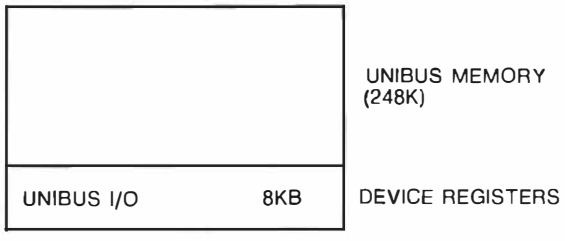


Figure 2 VAX-11/780 UNIBUS Space

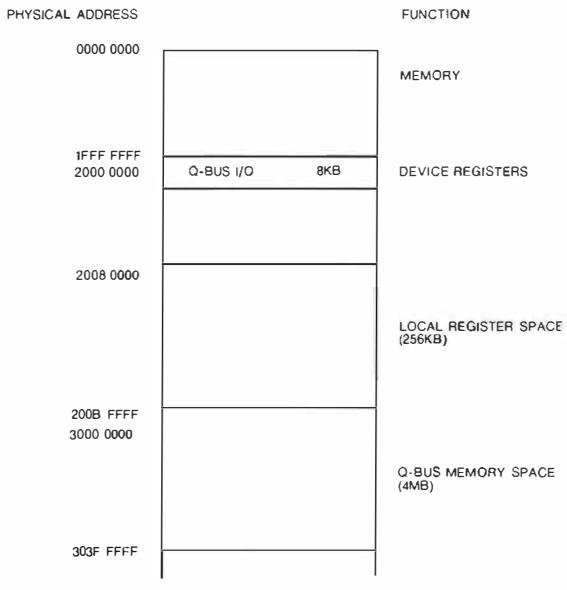


Figure 3 MicroVAX II Physical Address Space

Table 1 Comparison of Physical Address Spaces for the VAX-11/780 System and the MicroVAX II System

Physical Address Spaces				
VAX Function	**Size**	**MicroVAX II Function**	**Size**	**Purpose of Function**
Memory	2MB-64MB	Memory	16MB	Execute Programs
NEXUS	8K each	Local Register	256K	CPU and Bus Control Registers
UNIBUS Memory	248K	Q-bus Memory	4MB	Device Memory
UNIBUS I/O	8K each	Q-bus I/O	8K	Device Registers

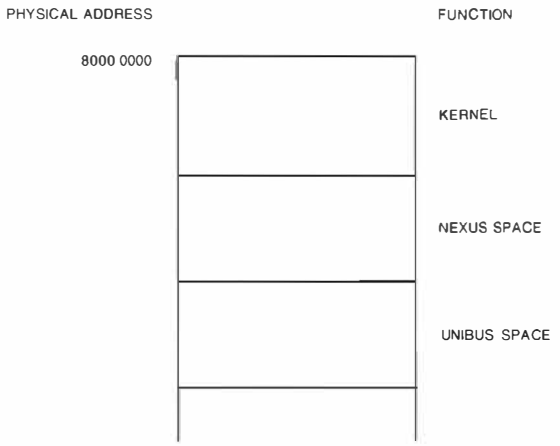


Figure 4 Result of Physical-to-Virtual Mapping

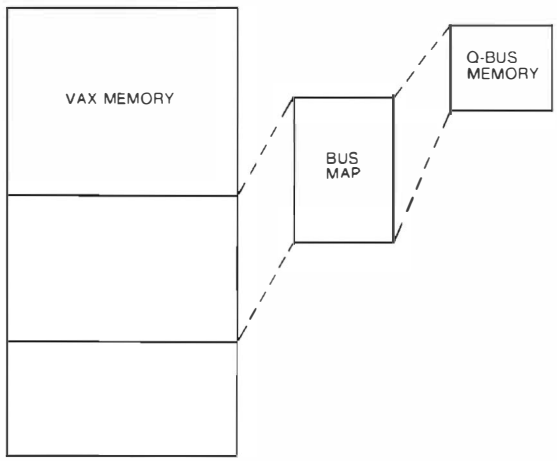


Figure 5 Q-bus Memory Mapping

address space. The adapter has a set of registers that maps this 256KB space onto the much larger VAX memory space. These registers perform the equivalent function that is provided by VAX Page Table entries. In effect, they "virtualize" the memory that devices access. Figure 5 depicts this mapping.

The MicroVAX II system contains a similar set of registers with the principal difference being that it has enough to map all four megabytes of main memory. Although that appears advantageous, it in fact posed a serious problem. The ULTRIX system dynamically allocates the bus-mapping registers from a central routine. It would have been easy to modify this routine to "know" about the extra registers. The problem encountered here was that these allocation routines return a word that is encoded as shown in Figure 6.

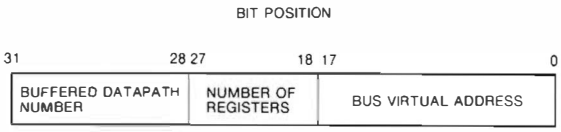


Figure 6 Coding of Allocation Routine Word

The upper part contains the number of the buffered datapath allocated, the middle is the number of registers used, and the lower is

the bus virtual address. The format of this 32-bit word is known by all device drivers that do DMA transfers. To change the word to use all the map registers available meant that the virtual address portion would need 22 bits instead of 18. That would have required corresponding changes in each of the device drivers. To determine the severity of these problems, we did some tests to see if the 18-bit format would be a limiting factor. Fortunately, we found that there were always registers available.

The end result of the mapping and map-register allocation scheme was that UNIBUS device drivers could be left unchanged as long as the Q-bus hardware was compatible with the UNIBUS versions. We took advantage of that fact and thus were able to support the TSV05, DHV11, and RL02 disk subsystems without any impact on the development schedule.[7]

Console Port

Traditional VAX systems have a separate processor that performs console functions. This processor is used to control the main CPU and replaces the older-style front panel. Instead of having switches for halt or run, the console runs a program that provides halt, run, examine, and initial program load capabilities. Programs running in a VAX system can communicate with the console through an internal processor register. Commands sent in this register are used by the operating system to reboot and restart the machine.

The MicroVAX system is different: the console functions are handled by the microprocessor, the MicroVAX 78032 chip, which runs a program resident in ROM. Like the larger VAX systems, a register is used to communicate with the console. A code can be placed in this register. When a subsequent HALT instruction is executed, execution switches to the console program in ROM, which then examines the code in the register.[8] In fact the register is actually a memory location in RAM that is backed up by batteries.

The ULTRIX system contains a reboot and halt routine that is accessed by a privileged system call. That routine was modified to communicate with the console program.

System Clock

The ULTRIX system keeps track of the current time by counting clock interrupts from the 10ms interval timer. The time is kept in memory as an unsigned integer; it is initialized from the time-of-year (TOY) register during system boot. The time is set by a privileged program through standard system calls and can be read by normal user programs. That set procedure is normally done by the system manager using the DATE command. DATE converts the time from a format of year, month, day, hour, minute, and second to the integer format needed by the system call.

User enters:		System converts to:
	– set →	
yymmddhhmmss		Integer
	← read –	

where yymmddhhmmss = Year, Month, Day, Hour, Minute, Second

The MicroVAX system does not have a TOY register; instead, it has a watch chip backed up by a battery. The chip contains a number of counters that correspond to the year, month, day, hour, minute, and second. We could have modified the system call or added a new one to explicitly set the MicroVAX TOY clock. That would have avoided the conversion to integer format, given that the user has to enter date and time information in the format needed by the watch chip. However, it would have meant that we needed two versions of the DATE command, one for existing systems and the other for the MicroVAX system, to use the new format. To avoid that, we borrowed the conversion routines from the DATE command and used them in MicroVAX versions of the system time-setting routine. The irony here is that the date is now converted twice. The integer format is present on either side of the system call.

User enters:		System reads:
	→ →	
yymmddhhmmss	integer	yymmddhhmmss
	← ←	

Missing Instruction Emulation

As mentioned previously, the MicroVAX hardware implements a subset of the full VAX instruction set. Most string instructions are missing and are emulated in macrocode instead of implemented in hardware. The emulation code could have been placed in libraries, where it could be linked with user-level code. To do that, however, would mean that linked

images from other VAX systems would not run on a MicroVAX system, thus violating one of its basic objectives.

Rather than using libraries, we chose to use an emulator designed by the VMS Development Group and ported that emulator to the ULTRIX system.[9] The emulator links with the kernel and is almost completely invisible to user programs. It is supported by new traps in the hardware that help to decode each missing instruction. When the kernel or a user program executes one of the missing instructions, a trap occurs and the emulation code takes over. That happens without changing mode; in other words, if an emulation trap occurs in a user program, the emulator is entered in user mode, not kernel mode like other traps. The result is user-mode execution of code in the kernel address space. (Unlike the VMS system, the entire ULTRIX kernel is normally unreadable by user programs.) The startup code now initializes the pages containing the emulator so that they can be read and executed by user-level code.

As stated earlier, the end result is a combination of hardware and software that is almost completely compatible with systems running the full VAX instruction set. In fact, executable images from other VAX systems can run without relinking. The only point of incompatibility is that the emulation code runs on the user stack when one of the missing instructions is executed by user code. (We have seen one customer application that was affected by this situation. The application used knowledge of its past usage of the stack to do "garbage collection" and was confused by the intermediate results of the emulation code. That is normally not a problem; the ULTRIX-32 and ULTRIX-32m kits have over 500 user-level programs. They are compiled and linked once on a full VAX system and then run without modification on the MicroVAX system.)

Summary

In porting the ULTRIX system to the MicroVAX processor, we opted to maintain compatibility with other versions of the system, wherever possible. We choose not to support hardware features if they violated internal or external interfaces. Therefore, we were able to deliver a broader range of peripheral support with a minimum of development. The end result–the MicroVAX system–combines hardware and software to provide customers, including developers of software device-drivers, with a product that runs all VAX programs for a fraction of the cost of a larger VAX system.

References

1. A detailed history of and supplemental information about the UNIX system can be found in the *AT&T Bell Laboratories Technical Journal,* vol. 57, no. 6 (July/August 1978) and vol. 63, No. 8 (October 1984).

2. Some programmers consider C to be a low-level language; in fact, it has proven to be more than adequate for programming an operating system like the UNIX system.

3. D.W. Dobberpuhl et al, "The MicroVAX 78032 Chip, A 32-Bit Microprocessor," *Digital Technical Journal* (March 1986, this issue): 12-23.

4. This work was done long before the first hardware prototype was developed.

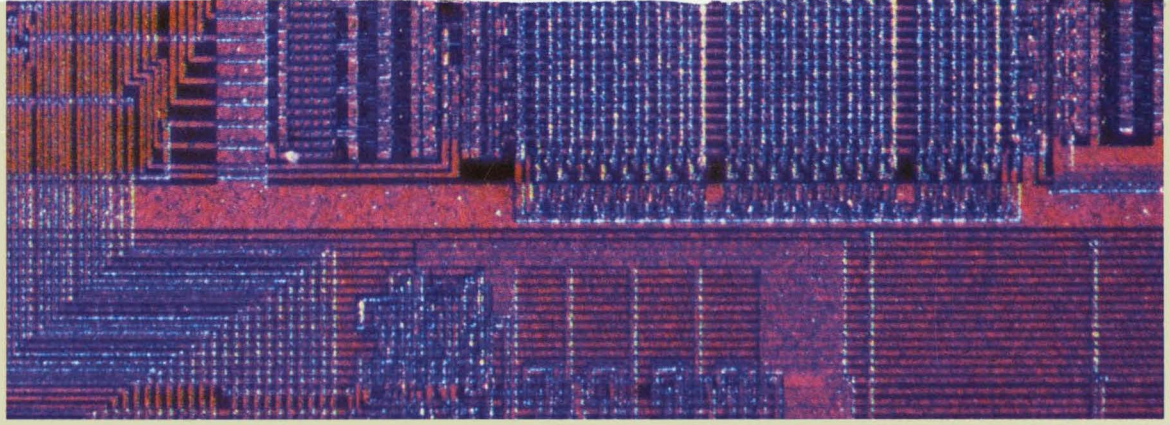
5. ODS-II is the VMS on-disk file structure.

6. The AT&T versions call this file "unix," while the Berkeley versions call it "vmunix," denoting "virtual unix."

7. However, we did have to expend time and energy to do additional configuration testing.

8. The MicroVAX 78032 chip never halts; it is running either ROM console code or programs in RAM.

9. K.D. Morse and L.J. Kenah, "The Evolution of Instruction Emulation for the MicroVAX Systems," *Digital Technical Journal* (March 1986, this issue): 76-85.



Printed in USA EY-3474E-DP Copyright © March 1986 Digital Equipment Corporation

digital™