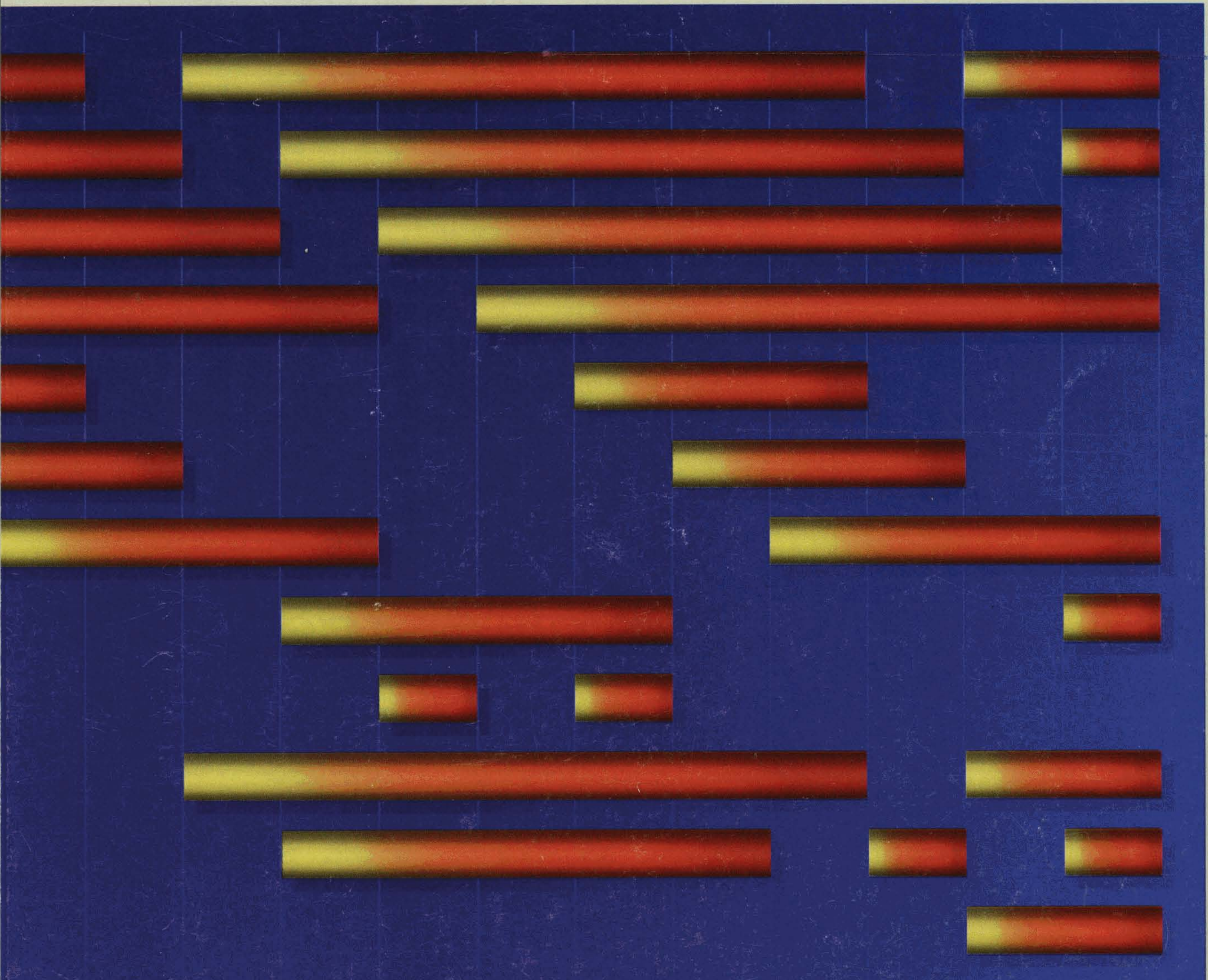
VAX 9000 Series

Digital Technical Journal

Digital Equipment Corporation



Volume 2 Number 4

Fall 1990

Editorial
Jane C. Blake, Editor
Barbara Lindmark, Associate Editor

Circulation
Catherine M. Phillips, Administrator
Suzanne J. Babineau, Secretary

Production
Helen L. Patterson, Production Editor
Nancy Jones, Typographer
Peter Woodbury, Illustrator and Designer

The *Digital Technical Journal* is published quarterly by Digital Equipment Corporation, 146 Main Street MLO1-3/B68, Maynard, Massachusetts 01754-2571. Subscriptions to the Journal are $40.00 for four issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to The *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to DTJ@CRL.DEC.COM. Single copies and back issues are available for $16.00 each from Digital Press of Digital Equipment Corporation, 12 Crosby Drive, Bedford, MA 01730-1493.

Digital employees may send subscription orders on the ENET to RDVAX::JOURNAL or by interoffice mail to mailstop MLO1-3/B68. Orders should include badge number, cost center, site location code and address. U.S. engineers in Engineering and Manufacturing receive complimentary subscriptions; engineers in these organizations in countries outside the U.S. should contact the Journal office to receive their complimentary subscriptions. All employees must advise of changes of address.

Comments on the content of any paper are welcomed and may be sent to the editor at the published-by or network address.

ISSN 0898-901X

Documentation Number EY-E762E-DP

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

Cover Design
Digital's VAX 9000 mainframe system is the theme of this issue. Our cover depicts several simple instructions flowing through the VAX 9000 instruction execution pipeline. High performance was achieved by breaking the VAX instructions into small simple tasks that could be pipelined efficiently. Concurrent operation on up to six instructions simultaneously resulted in a execution rate of one simple VAX instruction per clock period.

Gloria Monroy of the High Performance Systems Group designed the cover graphic, which was implemented in cooperation with David Comberg of the Corporate Design Group.

Contents

11 *Foreword*
Carl S. Gibson

VAX 9000 Series

13 *Design Strategy for the VAX 9000 System*
David B. Fite Jr., Tryggve Fossum, and Dwight Manley

25 *VAX Instructions That Illustrate the Architectural Features of the VAX 9000 CPU*
John E. Murray, Ricky C. Hetherington, and Ronald M. Salett

43 *Semiconductor Technology in a High-performance VAX System*
Matthew J. Adiletta, Richard L. Doucette, John H. Hackenberg, Dale H. Leuthold, and Dennis M. Litwinetz

61 *Vector Processing on the VAX 9000 System*
Richard A. Brunner, Dileep P. Bhandarkar, Francis X. McKeen, Bimal Patel, William J. Rogers Jr., and Gregory L. Yoder

80 *HDSC and Multichip Unit Design and Manufacture*
Peter B. Dunbeck, Richard J. Dischler, James B. McElroy, and Frank J. Swiatowiec

90 *The VAX 9000 Service Processor Unit*
Matthew S. Goldman, Paul H. Dormitzer, and Paul A. Leveille

102 *The Unique Features of the VAX 9000 Power System Design*
Derrick J. Chin, Barry G. Brown, Charles F. Butala, Luke L. Chang, Steven J. Chenetz, Gerald E. Cotter, Brian T. Lynch, Thiagarajan Natarajan, and Leonard J. Salafia

118 *Synthesis in the CAD System Used to Design the VAX 9000 System*
Donald F. Hooper and John C. Eck

130 *Hierarchical Fault Detection and Isolation Strategy for the VAX 9000 System*
Karen E. Barnard and Robert P. Harokopus

Editor's Introduction

Jane C. Blake
Editor

The VAX 9000, Digital's first mainframe computer, is the topic of papers in this issue of the *Digital Technical Journal*. As engineers writing for this issue relate, the primary goal of the project from the initial product strategy through manufacture was to design and build a very high-performance, highly reliable VAX system.

Design engineers applied both CISC and RISC techniques to achieve high levels of performance for this tightly coupled multiprocessor system. In the opening paper, Dave Fite, Tryggve Fossum, and Dwight Manley explain the strategy behind the design. They begin with an overview of the system, the technology, and CAD tools, and then describe the redesign of VAX instructions into small tasks which can be efficiently pipelined. The authors also touch upon three additional aspects of the VAX 9000 system: the integration of vector processing into the VAX architecture, new error handling techniques, and performance modeling.

One measure of performance is the number of instructions processed per cycle. The average number of cycles per instruction is less than five, which is nearly half the instruction execution rate of previous VAX systems. To illustrate the architectural features that enable this level of performance, John Murray, Rick Hetherington, and Ron Salett have selected a small sample of VAX instructions. They describe the instruction flow through the pipeline, how instruction features combine to work on a single macro, and how stages of the pipeline interact.

In addition to the architectural improvements, machine performance is enhanced at the semiconductor level by a new generation of semicustom and custom integrated circuits that support a low cycle time. Matt Adiletta, Dick Doucette, John Hackenberg, Dale Leuthold, and Dennis Litwinetz give an overview of the bipolar technology used in the system. They then describe the methods used to implement the 77 different gate array chips, the five custom chips, and the self-timed RAM architecture.

An additional performance improvement for numeric computations is the VAX vector architecture and is treated in the paper by Rich Brunner, Dileep Bhandarkar, Frank McKeen, Bimal Patel, Bill Rogers, and Greg Yoder. They discuss the architectural model and particulars of the VAX 9000 implementation, which affords numerically intensive applications performance four to five times greater than can be achieved by the scalar processor.

To ensure that the system performance gains at the semiconductor level were not diminished but were instead enhanced by packaging and interconnects, engineers developed several technologies unique in the industry. The technology behind the high-density signal carrier and the multichip unit are explained in the paper by Pete Dunbeck, Rich Dischler, Jim McElroy, and Frank Swiatowiec.

Equally important to performance in the new 9000 is system reliability as evidenced by the introduction of the service processor unit. In their paper about the service processor, Matt Goldman, Paul Dormitzer, and Paul Leveille relate how the MicroVAX-based system embedded within the 9000 detects, isolates, and corrects problems without interrupting the system.

High system availability was also one impetus in the design of the power system. Some of the unique features of the power system, such as redundant regulators, improved load sharing and simulation, are discussed by Derrick Chin, Barry Brown, Charles Butala, Luke Chang, Steve Chenetz, Jerry Cotter, Brian Lynch, Raj Natarajan, and Len Salafia.

The two papers that close this issue address the topics of CAD methodology and system diagnosis. Don Hooper and John Eck describe a CAD methodology that combines advanced rule-based AI techniques with an object-oriented database. The new methodology saves logic designers significant time and reduces errors. A complex system such as the VAX 9000 requires improved system diagnosis capabilities to achieve the desired high system availability. Karen Barnard and Bob Harokopus demonstrate how a new scan system, in combination with scan pattern testing, and symptom-directed diagnosis achieve this necessary diagnosis capability.

The editors thank Rick Hetherington of the High Performance Systems Group for not only writing a paper but for his help in coordinating this issue.

Jane Blake

Biographies

Matthew J. Adiletta Matthew Adiletta is currently contributing to the implementation of a new processor architecture and performing a technology evaluation to determine the technology for the implementation. He joined Digital in 1985 to work on a high-performance RISC architecture. Matt was not only the architect for the VAX 9000 system, but he also implemented the integer and floating point multiply and divide units and developed an ECL custom chip process. He holds one patent and has several patents pending. Matt received a B.S.E.E. (honors, 1985) from the University of Connecticut.

Karen E. Barnard A senior software engineer with the High Power Business Unit CPU Development Group, Karen Barnard wrote the read-only memory-based diagnostic for the VAX 9000 service processor unit's scan control module and developed the scan pattern diagnostic for the VAX 9000 CPU and SCU. Karen also worked on the debugging structural test process for the VAX 9000 kernel environment. Prior to joining Digital in 1986, Karen was with Data General Corporation. She received a B.S. (1983) in computer science from the Worcester Polytechnical Institute.

Dileep P. Bhandarkar As technical director for RISC systems, Dileep Bhandarkar is responsible for leading the architectural direction of RISC products. He joined Digital in 1978 and was responsible for managing the evolution of the VAX architecture. Dileep was the chief architect for VAX vector processing and coarchitect of Digital's RISC architecture. He holds one patent for his work at Digital and has several patents pending. His degrees in electrical engineering include a Bachelor of Technology from the Indian Institute of Technology and an M.S. and a Ph.D. from Carnegie-Mellon University.

Barry G. Brown The concept of designing DC-to-DC converters as system elements rather than individual "power supplies" was introduced into the high-power systems products by Barry Brown. He created and developed a highly flexible, high-reliability DC-to-DC conversion system for the VAX 9000 series. Barry designed, implemented, and verified the power system for the VAX 9000 Model 200 systems. He was a principal engineer for the Codex Corporation before coming to Digital in 1984. Barry is a graduate of Woolwich Polytechnic and Harlow Technical College.

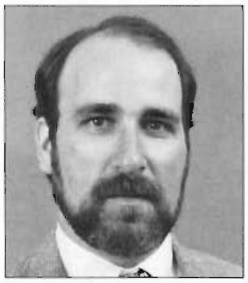
Richard A. Brunner As a principal engineer, Richard Brunner is the architect currently responsible for the engineering refinement and control of both the VAX and VAX vector architectures. He is the editor of the *VAX Architecture Reference Manual* and coauthor of the *VAX Vector Handbook* and several papers on the VAX vector architecture. He received a B.S. (high honors, 1984) in electrical engineering from Case Western Reserve University and an M.S. (1987) in computer engineering from Rensselaer Polytechnic Institute. He is a member of IEEE and Tau Beta Pi.

Charles F. Butala Presently responsible for the power system design and architecture of the VAX 9000 Model 400 systems, Charles Butala is a consulting engineer in the Information Systems Business Unit Power Systems Group. Since he joined Digital in 1976, he has been responsible for several power system design projects, including the VAX 8600 system. He is a member of IEEE and Tau Beta Pi, and holds honorary society membership in Eta Kappa Nu. Charles received a B.S.E.E. (1968) from Illinois Institute of Technology and an M.S.E.E. from Northeastern University.

Luke L. Chang After receiving his M.S. in electrical engineering from Virginia Polytechnic Institute and State University in 1988, Luke Chang joined the Power Systems Technology and Regulations Group. He is currently a hardware engineer and is responsible for developing simulation tools to perform high-quality software design verification tests for the next generation DC-to-DC power converters. Luke's previous responsibilities include transient analysis and testing of the VAX 9000 memory power distribution system, and power system cost reduction studies.

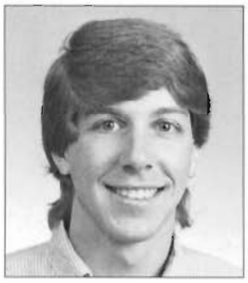
Steven J. Chenetz As a principal engineer in the Information Systems Business Unit Power Systems Group, Steven Chenetz is currently working on the H7390 for a high-power VAX system. He previously was a member of the design and development teams for the H7380 of the VAX 9000 system, the H7188 environmental monitoring module for the VAX 8600 power system, the VAX 8600 clock distribution system, and signal integrity for the VAX 8600 system. Steve joined Digital upon graduation from Rensselaer Polytechnic Institute in 1981. He has an M.S.E.E. from Northeastern University (1987).

Derrick J. Chin Derrick Chin is the engineering manager for several Information Systems Business Unit power groups and is design engineer of the VAX 9000 processor's DC power distribution system. His association with Digital began in 1961, and he has participated in many projects, from the PDP-1 and the DECsystem-10 to the VAX 8650 systems. His responsibilities have ranged from development of precision displays, circuit design, and core and semiconductor memories to environmental monitoring modules and power systems. He holds a B.S.E.E. (1959) from MIT.



Gerald E. Cotter Principal engineer Gerald Cotter is a member of the Information Systems Business Unit Power Systems Group. He was the project engineer and coarchitect of the VAX 9000 power control system (PCS). Jerry was the PCS interface to Customer Service and Support Engineering, Manufacturing, and Service Processor Unit Groups. He participated in development of the PCS and power system test strategies and the initial design of the T01060 power and environmental monitor module. His previous work includes the VAX 8600 system's power and control subsystem.

Richard J. Dischler In his position of systems engineer for the High Performance Systems Group, Richard Dischler worked on the VAX 9000 signal integrity project. He also was a member of the project team for the electrical design of HDSC and micropackaging for multichip units, planar boards, and connectors for the VAX 9000 system. Rich held similar responsibilities in the development of the VAX 8600 system. He joined Digital in 1982, and his previous experience was at Applied Research Laboratories. He holds a B.S.E.E. (1982) from Pennsylvania State University.



Paul H. Dormitzer As an undergraduate at Harvard University, Paul Dormitzer gained experience with the UNIX operating system by working as a programmer and operator. Upon receiving his B.A. in computer science in 1987, he joined Digital's High Performance Systems Group. He is currently an engineer in the High Performance Business Unit CPU Engineering Group. Paul's primary responsibilities are in the development of error recovery processes for high-power systems, such as the VAX 9000 system.

Richard L. Doucette Since joining Digital in 1979, Richard Doucette has been a member of several high-performance systems project teams. As a senior engineer on the VAX 8600 team, he helped introduce the Motorola Macrocell Array I (MCA1) technology into Digital and was responsible for its design analysis and characterization in the system. As engineering manager on the VAX 9000 team, he was responsible for the incorporation of MCA3 technology, custom chips, and self-timed RAM components in the system. He holds a B.S.E.E. (1973) from the University of Maine.

Peter B. Dunbeck Peter Dunbeck is an engineering manager in the High Performance Business Unit Technology Research and Engineering Group. He held various positions on the VAX 9000 program between 1985 and 1990, including technology program manager and design engineering manager for the multichip unit. Before joining Digital in 1984 as a manufacturing engineer, Peter developed energy conservation programs for Thermo Electron. He holds a B.S. (1977) in mechanical engineering from Virginia Tech and an S.M. (1979) in aeronautics and astronautics from MIT.

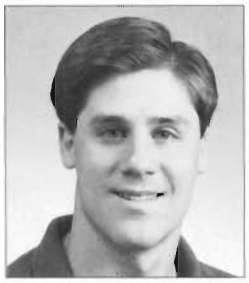
John C. Eck The development of the majority of the physical design CAD tools used in the VAX 9000 system was managed by John Eck. He is a software engineer manager in the High Performance Systems CAD and Diagnostics Group. John was employed as the manager of the Automated Design Department of Badger Company before coming to Digital in 1984. He holds a B.S. (1964) in physics and an M.S. (1966) in aeronautics and astronautics from MIT, and an M.B.A. (highest honors, 1984) from Babson College.

David B. Fite Jr. Consultant engineer David Fite was a member of the initial architecture team for the VAX 9000 system. He developed the architecture for the branch prediction, instruction fetch, and instruction decode for the VAX 9000. His previous work includes responsibility for prototype debugging on the VAX 8600 system. Dave joined Digital in 1982. He has one patent and several patent applications pending. He is a graduate of Worcester Polytechnic Institute with a B.S. (honors) in electrical engineering.

Tryggve Fossum Tryggve Fossum is the system architect of the VAX 9000 system. He received a B.S. (1968) from the University of Oslo and earned his Ph.D. (1972) from the University of Illinois. Tryggve joined Digital in 1973 and worked on the design of high-end computers, notably the VAX-11/780 system. As a project leader on the VAX 8600 team, he guided the design of the floating point accelerator. He has also worked on several research projects, including an early raster scan graphics workstation, and a workstation with an integrated disk system.

Matthew S. Goldman As a senior engineer on the VAX 9000 project team, Matthew Goldman designed the scan control chip, which contains the control logic for the VAX 9000 scan system. He was also the responsible engineer for all VAX 9000 service processor hardware. Prior to joining Digital's High Performance Systems CPU Group in 1986, Matt was a design engineer for Raytheon Company. He is a member of Tau Beta Pi and Eta Kappa Nu. Matt holds a B.S. (highest honors, 1983) and an M.S. (1988) in electrical engineering from Worcester Polytechnic Institute.

John H. Hackenberg In 1968, John Hackenberg came to Digital as a technician on the KI-10 project, leaving after two years to serve in the armed forces. He returned to Digital in 1971 and worked on the designs for various high-end systems, including the KL-10. As a consulting engineer on the VAX 8600 project, he worked in the area of signal integrity. John was the project leader for the MCA3 gate array used in the VAX 9000 system and is currently developing a bipolar gate array. He holds a B.S.E.T. (1979) from the University of Lowell.



Robert P. Harokopus A cum laude graduate of the University of Michigan, Robert Harokopus received a B.S. (1986) in computer engineering and is now studying for an M.S. in computer engineering from Boston University. Bob is a senior software engineer and joined Digital in 1986. He developed the symptom-directed diagnosis software used in the VAX 9000 service processor unit. Bob also developed software for the HIDE CAD tool and SCEPTER automatic test pattern generator, both of which were used in the VAX 9000 design project. He is a member of Tau Beta Pi and Eta Kappa Nu.

Ricky C. Hetherington As a principal engineer with the High Performance Systems Group, Ricky Hetherington is currently the project leader of the translation buffer and cache design of the VAX 9000 system. He holds one patent and has several patents pending on the various design features of the VAX 9000 M-box. Rick joined Digital in 1982 as a senior engineer in Digital's Large Computer Group. He has a B.S. from Pennsylvania State University.

Donald F. Hooper Don Hooper is a consulting engineer in both logic design and CAD disciplines. He initiated and led the development of the Synthesis of Integral Design program, Digital's first synthesis tool. Before coming to Digital in 1979, he was architect for the Itel 7031 mainframe and cache designer for the Itel Advanced System 4. He is a graduate of Don Bosco Technical Institute. Don holds patents in speech recognition circuits, the tag and queuing system for Digital's first pipelined CPU, and the control storage pipe for the VAX 8600 system. In addition, he has several patents pending in logic synthesis.

Dale H. Leuthold A member of the technical staff of the Integral Circuit Design Group, Dale Leuthold led the design team for the VAX 9000 vector register chip. He is currently working on random-access memory development for high-speed mainframes. Dale was responsible for bipolar integrated circuit design at Signetics Corporation and Trilogy Systems Corporation before coming to Digital in 1986. He holds one patent and has one patent pending. Dale received a B.S. from Oregon State University.

Paul A. Leveille In his nearly ten-year relationship with Digital, Paul Leveille has specialized in the development of high-power systems, particularly the VAX 8600 and VAX 9000 systems. As a principal engineer in the High Performance Business Unit, he helped define the VAX 9000 service processor subsystem and was responsible for developing the scan control firmware and portions of the service processor application software. Paul's previous responsibilities include console diagnostics, firmware, and application software.

Dennis M. Litwinetz The project leader for the design of four standard cell and custom chips for the VAX 9000, Dennis Litwinetz is a consulting engineer in the High Performance Business Unit. He has previously participated in the design of two standard cell chip designs for the VAX 8600 system. He joined Digital in 1967 as a technician for the DECsystem-10 Engineering Group. Dennis has a patent pending for the VAX 9000 self-timed register file design. He received a B.S.E.E.T. from Lowell Technological Institute and an M.S.C.E. from the University of Lowell.

Brian T. Lynch Brian Lynch is a principal hardware engineer in the Information Systems Business Unit Power Systems Group. In this position, he designed and developed the H7382 bias power supply used in the VAX 9000 system. He is presently working on power solutions for future high-performance systems. Prior to joining Digital in 1972, Brian was responsible for power converter and analog module design at Intronics. He has a B.S.E.E. (1978) from Worcester Polytechnic Institute.

Dwight Manley As a principal engineer on the VAX 9000 project, Dwight Manley was responsible for all of the performance modeling of the VAX 9000 CPU design. His present responsibilities include writing code for a Digital Extended Math Library product. Dwight joined Digital in 1979 as a member of the Systems Performance Analysis Group. Prior to that time, he worked as a systems programmer for the Bell Telephone System. Dwight has a B.S. (1971) in mathematics from the University of Massachusetts and an M.S. (1976) from Northeastern University.

James B. McElroy Jim McElroy is the multichip unit operations manager. His work on the VAX 9000 system began with interconnect and packaging, followed by the management of the physical technology efforts. He then became the manufacturing systems program manager for the introduction of the VAX 9000 system into manufacturing. Before joining Digital in 1976, Jim worked at RCA on packaging and interconnect design for military computer systems. He received a B.S.M.E. and an M.S.M.E. from Northeastern University.

Francis X. McKeen The project leader for the V-box unit of the VAX 9000 system was Francis McKeen. Prior to working on the VAX 9000 system, he wrote microcode for the VAX 8600 and VAX 8650 systems. Frank is a principal engineer and has been with Digital for seven years. He holds one patent and has several patent applications pending. Frank received a B.S.E.E. from Northeastern University and is a member of IEEE.

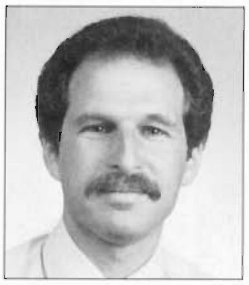
John E. Murray The coauthor of *Microarchitecture of the VAX 9000,* John Murray is a consulting engineer in the High Performance Business Unit. He served as project leader of the design team for the I-box unit of the VAX 9000. He joined Digital in 1982. John's previous employer was ICL in the United Kingdom, where he was a design engineer. He received a B.Sc. (1969) from Warwick University. He holds one patent and has several patents pending.

Thiagarajan Natarajan Thiagarajan Natarajan is manager of a DC-to-DC converter group in the Information Systems Business Unit. His group develops a high-density and highly reliable DC-to-DC converter, associated hybrids, semiconductor components, and the distribution system for the next generation, high-performance VAX systems. Raj's prior experience includes positions at General Electric, Bell Laboratories, and Perkin Elmer Corporation. He has a Ph.D. in electrical engineering, has been awarded one patent, and has authored approximately seventeen technical papers.

Bimal Patel Principal engineer Bimal Patel joined Digital in 1986 as a senior engineer. His primary responsibility since that time was the design of the V-box unit of the VAX 9000 system. Bimal was previously employed as a senior engineer in the CPU Design Group of Prime Computer, Inc. He has an M.S. in computer engineering from Boston University.

William J. Rogers Jr. William Rogers is an engineer in the VAX 9000 CPU Group, where he developed the design of the control logic of the V-box unit for the VAX 9000. Prior to working on this high-performance system, Bill was a member of the SASE Support Engineering Group. He joined Digital in 1986 and is a member of IEEE and Tau Beta Pi. He received a B.S. (1986) in electrical engineering from Michigan Technological University.

Leonard J. Salafia The development of the AC front end for the VAX 9000 system was the responsibility of Leonard Salafia, who is the manager of the AC Power Interface Development Group. His previous work at Digital includes supervising the development of storage system power products for the Central Power Supply Engineering Group and for the Storage Systems Power Group. Len worked for General Electric prior to coming to Digital in 1980. He holds a B.S.E.E. (magna cum laude, 1969) from the University of Hartford and an M.S.E.E. (1976) from Rensselaer Polytechnic Institute.

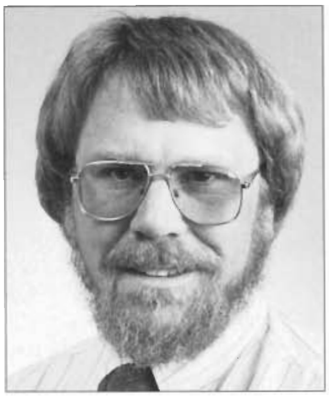


Ronald M. Salett As a consulting engineer in the High Performance Systems Group, Ron Salett is currently leading the development of a new high-performance CPU. As a project leader for the VAX 9000 system, he was responsible for the architecture, design, and microcode of the execution unit. Since joining Digital in 1977, Ron has also worked as an architect and project leader on low-end integrated PDP-11 systems. He holds two patents. Ron holds a B.S.E.E. (1975) from Carnegie-Mellon University and an M.S.E.E. (1979) from Worcester Polytechnic Institute.

Frank J. Swiatowiec In 1988, Frank Swiatowiec became HDSC operations manager, with the primary responsibility to transition Digital's new HDSC technology to volume production. He was one of the engineering managers responsible for the definition and development of the HDSC. Frank had over 15 years of experience in the semiconductor industry when he joined Digital in 1986. While with Motorola Corporation, he was awarded four patents on ECL circuit designs. Frank holds a B.S.E.E. from the University of Illinois and an M.S.E.E. from Arizona State University.

Gregory L. Yoder Gregory Yoder is a senior hardware engineer with the High Performance Systems CPU Engineering Group. His primary responsibilities on the VAX 9000 system included the design and testing of the V-box unit, and prototype system debug, for which he received an excellence award. He also assisted Manufacturing in producing and installing external field test VAX 9000 machines. Greg joined Digital in 1988, after participating in a one-year co-op session at IBM. He holds a B.S.E.E. from Pennsylvania State University.

Foreword



Carl S. Gibson
VAX 9000 Program Manager

This issue of the *Digital Technical Journal* is a collection of papers describing the technologies, designs, and design methods employed in Digital's VAX 9000 mainframe/supercomputer, which was introduced in the fall of 1989.

The VAX 9000 system embodies hundreds of innovations in most areas of design, manufacture, and service. In selecting papers for this journal, we have attempted to reflect the immense scope and variety of this program, which ranks among the largest and most complex in the history of our industry.

In the summer of 1983, a small group of us set about to determine what it would take for Digital to develop a true mainframe. We felt that a mainframe VAX would be a powerful addition to Digital's product family. The products that we have created took form, changed, and evolved over the months and years as technical challenges yielded to innovations, rigor, and discipline. An undertaking on this scale necessarily undergoes numerous transitions as new data emerges, assumptions are tested, and alternatives are eliminated. Technical breakthroughs built upon one another incrementally as we pressed the design closer to our goals. The primary objectives of very high system-level performance and world-class reliability drove the design process and the changes that emerged.

The planar logic packaging is illustrative of how changes and improvements built upon one another. The reliability benefits of minimal connections precipitated a logic packaging design change from stacked modules in dual backplanes to the planar array. This change—an optimization for reliability—in the end actually helped performance and maintainability. Ultimately, though not envisioned at the time, the adoption of the planar array had a significant impact in that this structure enabled impingement air cooling and elimination of the bulky liquid system that was part of the initial design. The final design of the VAX 9000 system reflects, in myriad forms, this continual process of successive refinement toward shared goals.

Design changes notwithstanding, our primary strategy remained constant. The reader will note that, while we innovated aggressively in CPU structure, implementation technologies, and design methodologies, we preserved full compatibility with the VAX, Digital storage, and Digital networking and cluster architectures. We wanted Digital and our customers to be able to enjoy very high performance levels in a product that was compatible with prior investments. Therefore, we drew as much as possible from existing products and designs from many Digital development groups. As a result, the VAX 9000 system incorporates Digital's standard XMI bus and popular BI, CI, and NI system-level interconnects. The system runs VMS and ULTRIX operating systems, VAX layered products, and all of our customers' and independent software vendors' tools and applications. This capability proved especially rewarding when in the final months of the project, our own VAX 9000 prototypes, running our unmodified CAD tools, accelerated the processing of the inevitable last-minute changes.

High-performance computation fundamentally requires two key ingredients: short machine cycle times and maximum computational work performed in each cycle. The semiconductor and multichip unit papers describe how we minimized the VAX 9000 cycle time by use of fast circuits, high-density packaging, and high-speed interconnects. These papers are complemented by architecture descriptions through which the authors present the innovative features that minimize the number of cycles required to execute the VAX instruction set. These papers present the sophisticated pipelining techniques and vector processing capabilities incorporated in the VAX 9000 system.

Equal in importance to the computational capabilities of the product are the service and control features of the system. Papers covering the VAX 9000 service processor and the system's fault management capabilities provide the reader with insights into these important aspects of the product.

The development strategy for the VAX 9000 system was explicitly formulated to deal with enormous technical and project complexity. Complexity itself was the single most formidable challenge facing the team. Apparent from the outset, was the fact that such an ambitious product required the integration of a very large number of discrete design objects; each had to be conceived, created, documented, tested, and ultimately integrated and verified as part of the whole. The reader will see the diversity of these efforts and recognize the challenge of unifying a design from this breadth of technical advancement.

Central to our strategy was the creation of a unified design tool suite operating in a seamless, homogeneous VMS computing environment. The first few years of the project were devoted to construction of this environment in parallel with top-level design formulation. The recognition that rigorous design methods were crucial to our success was possibly one of the team's most powerful fundamental notions. Papers included in this journal illustrate some of the legacy of powerful CAD tools and structured design approaches created by the VAX 9000 team.

As we have seen for the product, the methodologies were not immune to change as the project progressed. Working with rapidly evolving technologies, design process experts continually adapted to evolving user needs. Concurrent design permeated every aspect of the project and dominated the way people worked together, with many aspects of the technology and product design converging and adapting as we learned from our own processes. When the manufacturing process needed some help, designs could be reprocessed with the new rules and rereleased to keep things moving ahead.

And, move ahead they did! Today, the VAX 9000 system is installed at many customer sites where the systems are exceeding our original goals in both performance and dependability. It has been accepted by experienced, high-end computer users as a bona fide mainframe—a mainframe with the unique advantage of full integration with Digital's rich distributed processing architecture.

The VAX 9000 system was created by engineers working in many disciplines and collaborating worldwide to invent hardware, software, and processes that have significantly advanced the state of the art of computer design, manufacture, and service. The papers in this journal describe but a few representative examples of the creativity and determination of this large and dedicated team of professionals.

David B. Fite Jr.
Tryggve Fossum
Dwight Manley

Design Strategy for the VAX 9000 System

The VAX 9000 system is Digital's newest high-end processor in the VAX family. This paper describes the design strategy used to achieve high performance and shows how RISC concepts were applied to a CISC architecture. New opportunities for parallelism in VAX program execution were found by breaking the VAX instructions into simple tasks which could be pipelined efficiently. By using independent, dedicated pipeline stages, execution rates approach one instruction per cycle.

The task confronting the VAX 9000 design team was to develop a VAX system that outperformed any previous VAX system and that was competitive with similarly sized processors from other vendors. Although the VAX system is based on one of the world's most popular computer architectures, the VAX architecture's instruction complexities preclude efficient macroinstruction pipelining, such as that found in reduced instruction set computers (RISC). RISC processors can be built with low gate counts to handle simple, fixed-length instructions sets, load/store architectures, and delayed branching.

To compete with machines based on such architectures and still remain compatible with the VAX architecture, the design team chose to implement the VAX architecture on the VAX 9000 system by applying techniques that were similar to those used in RISC processors. We redesigned the VAX instructions into small, simple tasks, and designed dedicated hardware that was optimized for each task. The result is a network of specialized processors, each of which has its own data paths and state machines, that operate in parallel and execute VAX instructions quickly. The most common, simple instructions are executed at the rate of one per cycle.

System Overview

The VAX 9000 system is a tightly coupled multiprocessor, which runs the symmetric multiprocessing (SMP) version of the VMS operating system and can have up to four processors sharing a central main memory. Figure 1 shows a simplified block diagram of the system. The major system components include four CPUs, two memory controllers, two I/O controllers, and a service processor, which is connected through the system control unit (SCU). Through a cross-bar switch, the SCU provides high-speed, simultaneous transfers among the central processors, I/O devices, and memory banks. System cache consistency is maintained with duplicate tag directories located in the SCU. As references are made to memory, the addresses are checked against the tag directories. If a cache hit occurs, the cache in question is requested to invalidate or write back to main memory. The SCU supplies a bandwidth that allows near linear performance improvement as new processors are added to the system. The memory is interleaved on cache block boundaries to provide bandwidth for multiple CPUs and vector processors.

Four XMI backplane buses provide high bandwidth paths to I/O devices. Although the XMI is used as the system bus in VAX 6000 systems, the XMI is used exclusively for I/O in the VAX 9000 system. Several new adapters were designed to increase throughput and reduce latency for I/O transactions. These adapters include connections to the CI, the NI, the BI, and local disk controllers. Although high-performance I/O features, such as disk striping, solid-state disk, and load balancing have been added to all VAX systems, the VAX 9000 system benefits the most from these features because it has the I/O backplane bandwidth to take advantage of them. A block diagram of a single VAX 9000 CPU connected to the SCU and the major data paths between the two units is shown in Figure 2.[1]

Technology Contributions to Improved Performance

The central processor cycle time has been reduced to 16 nanoseconds (ns) mainly by the use of fast emitter-coupled logic (ECL) semiconductors and

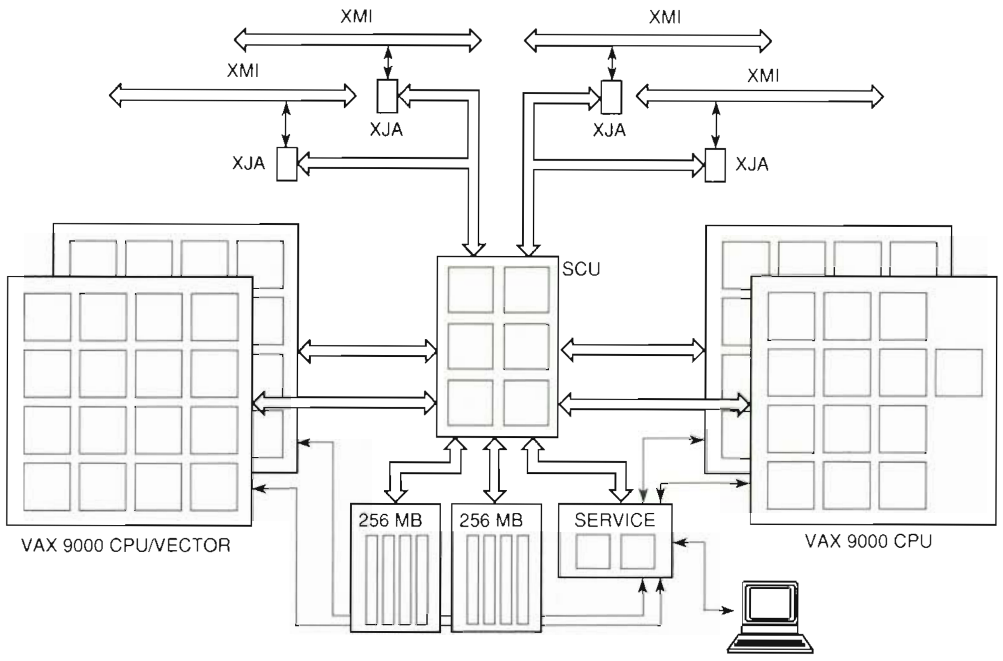


Figure 1 VAX 9000 System Diagram

fast self-timed random-access memories (RAMs) for registers and caches, and by decreasing the interconnect wire length between components.

Motorola's Macrocell Array III (MCA3) technology provided both macrocell array and standard cell capabilities. The entire system is composed of 77 unique MCA3 options and 5 custom chip types. A single MCA3 contains 838 cells (414 major, 224 input, and 200 output), which yield 10,000 equivalent gates, and 256 I/O pins. Maximum power dissipation is 30.0 watts, with unloaded gate propagation delays of 120 picoseconds (ps). Performance-critical operations, such as multiplication, division, integer and vector register accesses, and system clocking, were further aided by employing custom chips.[2]

Caches for instruction stream and memory data, scratch pad registers, and control stores all require high-speed local storage. Two versions of a proprietary self-timed RAM were designed for these specific applications. A 4 kilobit (Kb) self-timed RAM, at 5.5 ns, and a 16Kb self-timed RAM, at 11.5 ns, provide internal input and output latches and write pulse generation circuitry. Multiple access modes allow highly pipelined operations to take advantage of shorter access times.

Each new semiconductor generation reduces cycle time, which increases the relative importance of interconnect delay. High density signal carriers (HDSC), tape automated bonding, and a single planar module all reduce the interconnect delay between active components in the VAX 9000 system. Strict impedance control is maintained throughout the system. Clock skew is minimized by employing fixed-length, differential transmission and dedicated routing layers.

CAD Contributions to Improved Performance

Hundreds of computer-aided design (CAD) tools were used during the design and construction of the VAX 9000 system. However, none of these tools was more important in improving performance than the physical layout and timing analysis tools. Once the design team had placed large functional sections, placement tools refined individual macrocell selection and pin placements. Over 33,000 pins were selected to minimize overall wire length and maximize critical interconnections.

Routing presented several challenges. All levels of interconnect included critical signals, differential pairs, and fixed-length requirements. The HDSC contains large cutouts that enable die attachment and allow cooling through the back panel. These large routing restrictions and special routing characteristics could not be handled by existing CAD tools. Therefore, we developed Chameleon, a general-purpose router. With Chameleon, crosstalk is minimized, and crossing counts are maintained and used to increase signal integrity, which improves performance.

To model the timing relationships within the system, we used sophisticated CAD tools to generate an accurate representation of the VAX 9000 system. Detailed timing models of each macrocell device were created using the SPICE simulator program.[3] Chameleon and signal integrity tools provided delay values for each signal within the MCA3, HDSC, and planar modules. CPUDLY, using the AUTODLY timing tool, tied the various pieces together and gave the design engineers a powerful view of the timing domain.

Instruction Processing

VAX systems exist in a variety of environments and run thousands of applications. With any new, high-performance VAX system, it is important to increase the speed of all applications and to continue to provide general-purpose computer power. Given the size of the installed VAX base and the nature of the applications, performance gains should not require code modifications. Digital has gathered substantial information on how VAX processors are used. This data formed the basis for design decisions and trade-offs we made in the development of the VAX 9000 system.

Simple Instructions

In many VAX programs, only a few opcodes are responsible for a large percentage of the instructions issued. Most of these opcodes are simple and limited to a single arithmetic or logical operation. Often, one of the operands is in memory. A typical example is

```
ADDL3 (R0),R1,R2
```

Because of the high frequency of these instructions, speeding up these instructions is a top priority. Most of the high performance achieved on RISC processors is derived because these instructions are pipelined. In a complex instruction set computer (CISC), such as a VAX system, pipelining macroinstructions is more complex. Therefore, previous VAX implementations have pipelined operations at the microinstruction level.[4]

Processing simple instructions in a VAX system involves obtaining and decoding the instruction, fetching source operands, performing an operation, and storing the result. The most important

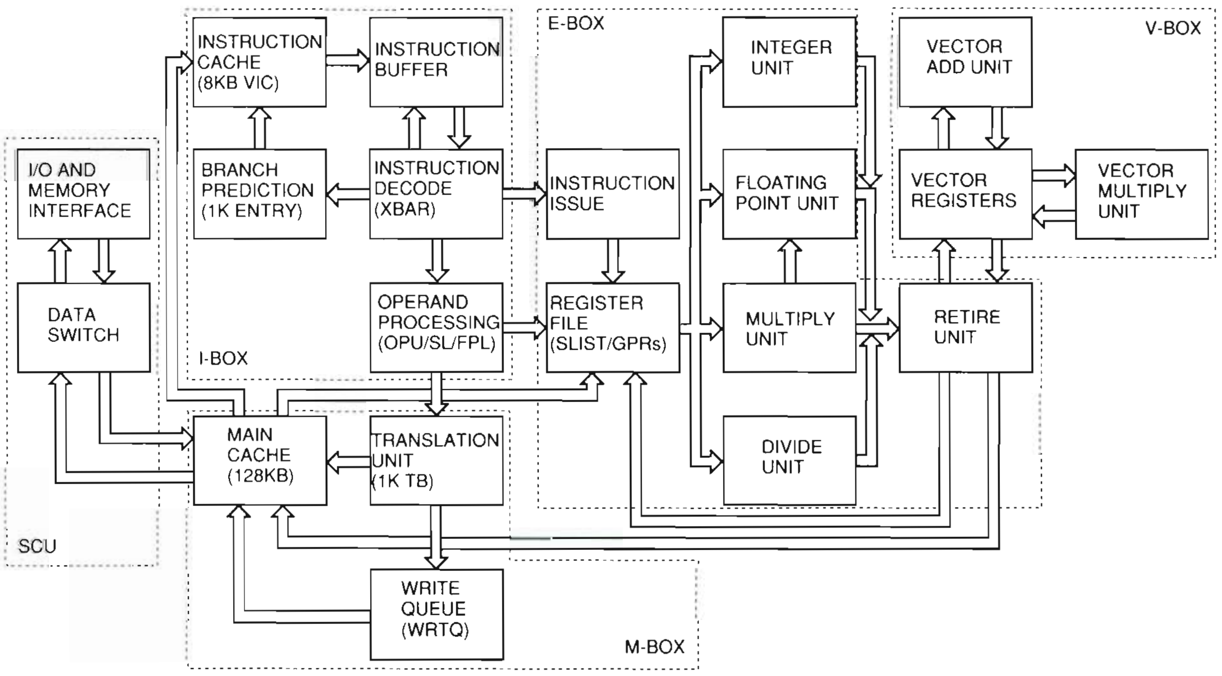


Figure 2 VAX 9000 CPU/Vector Block Diagram

difference between the way a VAX processor and a RISC processor process simple instructions is how the variable length instructions and memory specifiers are handled. VAX operands may reside in general-purpose registers (similar to RISC operands), in memory, or may be embedded in the instruction stream. The VAX architecture provides a rich selection of memory operand specifiers, which often require computations to create the address. In a RISC processor, only load and store instructions access main memory.

The instruction preprocessing stage (I-box) decodes instructions and fetches operands in the VAX 9000 system. In the execution stage (E-box), simple VAX instructions resemble RISC instructions. A simple opcode describes the operation, a single register file provides source operands, and a destination queue supplies a result descriptor. The I-box operates in parallel as with the E-box, which functions as a RISC processor by executing one instruction each cycle. Execution occurs without the need to identify the operand's source or addressing complexity. Figure 3 illustrates how simple instructions flow through the VAX 9000 pipeline. Although all VAX implementations perform these tasks, the VAX 9000 implementation uses separate, independent hardware units to overlap the work because concurrent operation is a prerequisite for single-cycle instruction execution.

Instruction Cache

We used an instruction cache in the I-box to decrease instruction stream fetch latency and reduce the bandwidth requirements on the main cache. Choosing a virtually addressed cache further reduced latency and simplified the design by removing the need for duplicate translation buffers. The virtual instruction cache is an 8 kilobyte (KB) cache with a quadword line size, 32-byte blocks, and a single-cycle access time. Line valid bits are maintained to allow variable size fills from the main data cache. Because the average VAX code block size is 16 to 20 bytes, the block size of the virtual instruction cache provides a good balance between the instruction decode stage and the main cache.

Context switches, translation buffer changes, and instruction stream modifications all require that the virtual instruction cache be invalidated. Two complete sets of block valid bits reduce cache sweeps to a single cycle, if consecutive sweeps do not occur within 256 cycles of each other. Block size and frequent sweeping reduce the virtual instruction cache's hit rate to approximately 96 percent, but by filling through the main cache, the miss penalty is minimized.

Instruction Decode

Because the majority of instructions executed require only a single cycle to execute, the instruction decode's task of keeping ahead of the E-box is not simple. Most instructions must be decoded in a single cycle to keep the VAX 9000 system's ticks-per-instruction (tpi) low.

For example, VAX instructions may contain up to six operand specifiers. With 59 different specifier addressing modes, instruction lengths can vary from a single byte to more than 50 bytes. However, the overall average VAX instruction length is 3.8 bytes, and 98 percent of instructions require only 8 or less bytes.[5] Furthermore, 96 percent of VAX instructions executed use only 3 or less specifiers.

In each machine cycle, a 9-byte instruction buffer is presented to the decode stage (XBAR). The instruction buffer contains instruction stream data prefetched from the virtual instruction cache. Instruction decoding consists of generating an initial microaddress, determining the number of specifiers for the instruction, including each specifier access mode and data type, and forwarding the appropriate specifier data to the operand processing stages. The XBAR can handle up to three specifiers. Instructions that contain more than three specifiers require additional decode cycles. Since general-purpose register specifiers occur approximately 41 percent of the time, three register specifiers can be processed concurrently.[6] Short literals comprise nearly 16 percent of the specifiers. However, the XBAR can only decode a single short literal per cycle. The remaining specifiers must all be processed by the operand processing unit, which

Table 1 Decode Cycles Required

	Instruction	VAX-11/780	VAX 8650	VAX 9000
MULF3	R3,R5,R7	3	2	1
ADDL3	S^#48,R4,@(R2) + [R3]	5	4	1
AOBLEQ	S^#63,R10,10$	3	3	1

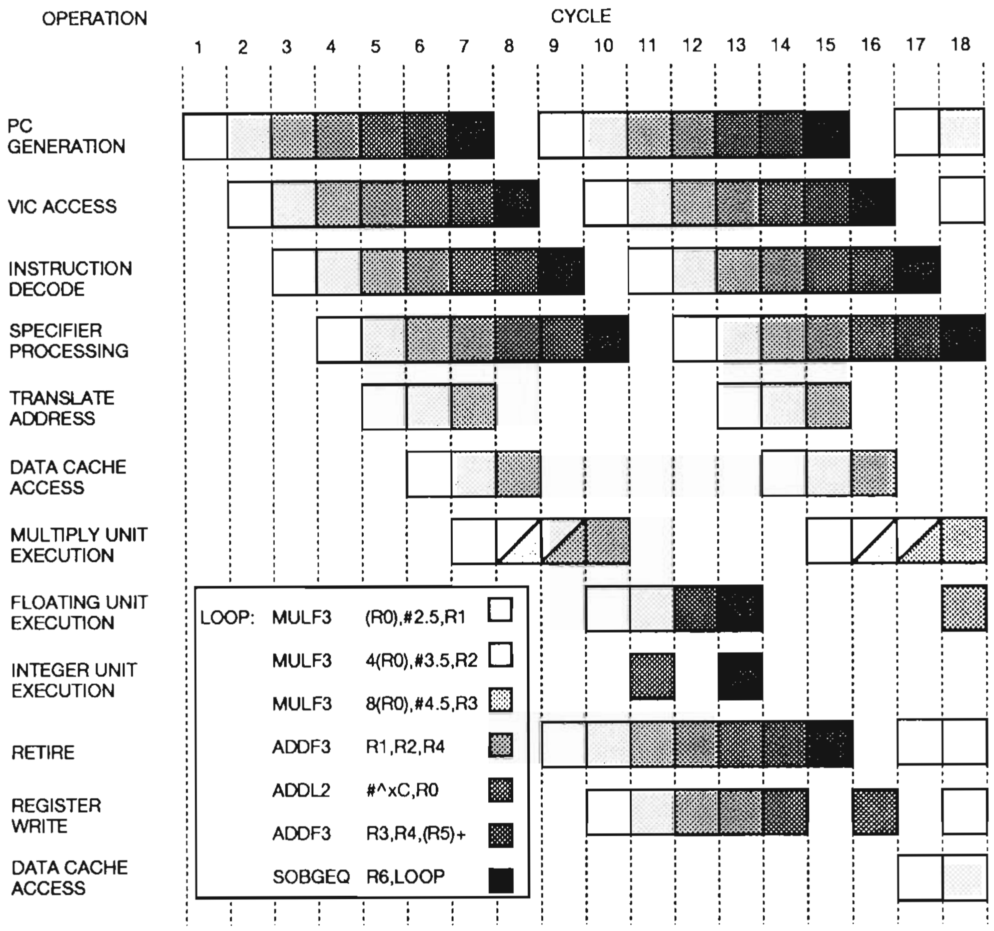


Figure 3 The VAX 9000 Instruction Pipeline

decodes a single complex specifier per cycle. Unlike preceding processors, the XBAR handles multiple specifiers in any order. Table 1 shows the number of decode cycles required for several VAX processors.

Operand Prefetching

Because most simple instructions are decoded and executed in a single cycle by various pipeline stages, instruction operands also must be handled in a single cycle. Multiple, specialized operand units increase operand processing throughput. From one to three register operands may be forwarded to the E-box by one register unit per cycle. A dedicated short literal unit expands all VAX data formats. The operand processing unit performs complex address calculations and requests memory operand data from the cache unit (M-box). Both the operand processing and short literal units can perform multiple cycle operations.

Load/Store Architecture

Load/store architectures separate memory accesses from computation. Loads can be scheduled to place arriving memory data at a functional unit just as an operation begins. To achieve this effect with VAX instructions, memory specifiers are treated as load/store instructions. VAX memory specifiers describe the effective addresses of memory operands. VAX memory specifiers do not contain the source and destination registers that are specified in RISC load/store instructions. Rather, the VAX 9000 system assigns temporary register file locations to buffer memory data. By processing specifiers early in the pipeline, data can be scheduled to arrive at the appropriate time.

Memory specifiers act as independent instructions executed in the operand processing unit. This unit creates the operand's effective address and forwards it to the M-box. For loads, the actual memory data is returned to the E-box register file. The translated physical address is saved in a queue of write addresses for store/destination specifiers. When execution results arrive from the E-box, the previously saved address is used to write the data into the cache.

Conflict Detection and Resolution

Macropipelining in the VAX 9000 system relies on autonomous units operating in parallel. Each independent unit is optimized for an individual task. However, macropipelining does require that mechanisms be added to resolve data dependencies among instruction processing units. Data conflicts occur when an instruction's results are required by an earlier pipeline stage. An addressing data conflict appears in the following example:

```
MOVL RO,R1
MOVB TABLE(R1),R2
```

Any dedicated address calculating hardware must wait for the MOVL instruction results before performing the MOVB instruction's effective address computation. A memory conflict is another form of data dependency.

In the following example,

```
MOVB RO,(R1)
MOVB (R2),R3
```

a prefetch unit could read the second instruction's source operand while the E-box writes the first instruction's results, if the values of registers R1 and R2 are different. However, when the registers contain identical values, the read must be delayed until the write occurs. The VAX 9000 system uses several different mechanisms to detect and resolve data dependencies. Passing pointers, scoreboard masks within the I-box, the write queue in the M-box, and architectural restrictions are all used to handle various conflicts.

Register Conflicts The simplest hardware mechanism employed in the VAX 9000 system is the use of pointers to reference data. The operand processing unit oversees a 16-entry source queue, an 8-entry destination queue, and a 16-entry source list. A single pointer is inserted into the source queue for each source specifier. The pointer represents either a register number, in the case of general-purpose register operands, or a tag that indicates an entry in the source list where the operand data is located. A pointer is added to the destination queue for each destination. This pointer represents a register number or a flag which indicates that the result should be written to memory.

The instruction issue unit removes source pointers from the source queue. These pointers are used to address either the general-purpose registers or source list for the actual source data. Destination pointers from the destination queue determine where results should be written. Register conflicts can be detected by comparing the source pointers needed to issue an instruction with all issued destination pointers in the destination queue. For example, in Figure 4, the MULL3's RO source queue entry would match the ADDL3's RO destination queue entry. A write to the general-purpose registers by the E-box removes the destination queue entry, and the instruction issue can resume.

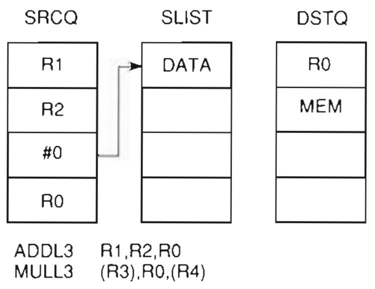


Figure 4 Register Conflict Detection

Addressing Conflicts To resolve addressing data conflicts, the I-box maintains a read/write register scoreboard. Two register masks are created for each instruction decoded. The first register mask denotes the general-purpose registers that the E-box will read for the instruction, and the second register mask specifies the general-purpose register writes. Each bit in these register masks refers to a single VAX general-purpose register. Specifiers that are being processed in the operand processing unit are checked against up to six previous instruction masks. From the first example above, the specifier [TABLE(R1)] requires that the operand processing unit read R1. If the R1 bit is asserted in any preceding instruction's scoreboard write masks, this effective address calculation must be deferred.

The VAX architecture presents a unique addressing conflict problem because some specifiers, such as −(Rn) and (Rn)+, modify general-purpose registers.

In the following example,

```
SUBL2 RO,R1
ADDL2 (RO)+,R2
```

the (R0)+ specifier modifies the contents of R0. Therefore, the operand processing unit cannot update the general-purpose register without affecting the prior instruction. The read masks are used to detect this type of conflict. All specifiers that modify general-purpose registers must check the scoreboard read masks before proceeding with the instruction. Thus, when a conflict occurs, the general-purpose register modification stalls.

When an instruction completes execution, the instruction's read/write mask is removed from the scoreboard. In all addressing conflicts, specifier processing continues once the blocking mask is removed.

Memory Conflicts The write queue is used to resolve memory conflicts. Physical addresses, received from the translation buffer, are inserted into an eight-entry FIFO. These addresses are later paired with the proper write data from the E-box and written into the M-box. To avoid prefetching stale data, all memory addresses for source memory operands are translated and compared with the addresses in the write queue. When no address conflict occurs, the data from memory is forwarded to the source list. Operand requests that conflict with a pending write address are stalled until the conflict is resolved. The conflict is resolved when the appropriate write data is received. The conflicting address is then removed from the write queue.

Miscellaneous Conflicts The VAX architecture includes instructions with operands that either are not known when the instruction is decoded (e.g., INSQUE, MTPR), or modify large portions of memory (e.g., MOVC5). To avoid conflicts from these instructions, the I-box suspends processing memory specifiers until the instruction execution is completed. Self-modifying code presents another form of conflict, which is solved by an REI instruction that notifies the hardware of this condition.

Branch Instructions

Branch instructions have a substantial influence on the overall performance of a VAX processor. On average, a VAX processor executes 3.9 instructions, including the branch, before a branch starts a new instruction sequence. Instructions that modify the program counter represent nearly 40 percent of the total instructions executed. The VAX 9000 system uses a 1024-entry branch cache and a two-tiered prediction scheme to increase the average code block size and reduce the branch-taken latency.

Unlike its predecessors, the VAX 9000 system commits all its resources to a single branch path. The prediction hardware selects the path of execution to resolve memory conflicts for those branch instructions that are decoded before results are available. This path selection is based on prior history, if the branch hits in the branch cache. If the branch does not hit in the branch cache, the path is predicted staticly, based on the instruction's opcode. When the branch executes, the prediction is compared to the actual results. The pipeline is flushed back to the correct code path if the branch prediction was incorrect.

The entries in the branch cache store the branch results of the previous execution of the branch and the target address, if the branch was taken. Because the branch cache is a one-way associative cache that can store only 1024 entries, the results have an average hit rate of approximately 80 percent. However, correct predictions occur 85 percent of the time from the cache, as opposed to an average hit rate of 56 percent, when the predictions are based solely on opcode. Loop branches are always predicted as taken, which increases the overall correct prediction rate to close to 89 percent. By caching branch targets, the calculation may be avoided and a latency factor of one-cycle branch taken is achieved. The branch cache can store a sufficient amount of branch context to eliminate the need to sweep the cache.

The I-box can process instructions with up to two conditional branches outstanding. Unconditional branches (e.g., BSBW, BRB) are processed as ordinary instructions by simply changing the instruction flow. To reduce the penalty for a bad prediction, which results in a four-cycle penalty, operand specifiers that modify general-purpose registers are not processed under a branch prediction and cause the operand processing unit to stall. Also, branch instruction execution is overlapped with the previous instruction to provide the actual branch results earlier.

Compute-intensive Instructions

Compute-intensive instructions require multiple execution stage cycles. Common examples of these instructions are multiplication, division, and floating point operations. All VAX implementations employ dedicated logic for compute-intensive instructions that occur frequently. Less frequently used instructions depend on microcode-controlled arithmetic and logical data paths. The VAX 9000 system contains four independent execution processors. The integer, floating point, multiply, and divide units execute the VAX instruction set. The I-box preprocesses instructions, which allows instruction execution to overlap in these units. In each cycle, a new instruction can be initiated in the appropriate unit prior to the completion of previous instructions. The floating point and multiply units are pipelined and can accept one instruction each cycle. The integer unit is pipelined for simple instructions. However, complex instructions must use microcode control to perform multicycle operations.

Pipelined instructions are issued in order and proceed through the data path without further microcode control. Upon completion, instruction results are retired in the same instruction order. The instructions must be processed in order because the result of one operation is often needed in a subsequent operation. Therefore, the pipelines must be short and contain data bypasses to make results available quickly. The multiply, float, and divide units' internal data paths are 64-bits wide. To understand how the pipelined and overlapped operations apply to the following operation,

$$y(i) = y(i) + C(i)$$

consider the program:

```
LOOP:  MULG3 R6,(R0)+,R4
       MULG3 R6,(R0)+,R2
       ADDG2 R4,(R1)+
       ADDG2 R2,(R1)+
```

The two MULG3/ADDG2 instruction pairs prevent a pipeline stall that could occur because of data dependencies. The instructions further reduce the loop overhead, which is already fairly small because the loop control instruction was predicted correctly. Instructions and source operands are prefetched. The multiply and add units accept the instructions as they become available. The memory references are made as the operand processing unit processes memory specifiers. The majority of specifier processing is performed independently of the instruction execution.

Memory-intensive Instructions

Some VAX instruction classes are primarily memory operations that require only minor computation. Typical examples of these instructions are character string, decimal, and privileged operating system. Pipelined execution offers little advantage to memory-intensive instructions because the number of memory references is not reduced as the number of cycles required for execution is reduced by new implementations. Because memory bandwidth is critical, the VAX 9000 system provides features to benefit these instructions.

For example, the virtual instruction cache services most instruction stream references, which frees the main cache to service prefetched operand references. Both the virtual instruction cache and the main cache have 64-bit data paths, important for character string operations and extended precision arithmetic. The caches are fully pipelined and allow one read per cycle. The main cache block size is 64 bytes, exploiting spatial locality. When cache references do miss, data is wrapped and the most critical data is returned first. A write back, write allocation algorithm further reduces main memory and cache bandwidth requirements and reduces latency.

The VAX system is a virtual memory architecture. Virtual addresses need to be translated to physical addresses through page tables in memory. A translation buffer caches the most recently used page tables entries. VAX systems, such as the VAX-11/780 system, process translation buffer misses in microcode, which can be time-consuming. However, the VAX 9000 system uses a memory management processor to process translation buffer misses as part of instruction preprocessing. This operation is performed early in the pipeline and is faster than microcode.

The CALL and RETURN instructions push and pop registers on the stack, and these instructions can be memory-bound. The VAX 9000 system contains both the control logic and the bandwidth to process these registers at a rate of one per cycle.

Unconventional Instructions

Special, dedicated hardware was added to the VAX 9000 system to process those VAX instructions that did not fit into the categories listed above. The additional hardware operates within the pipeline architecture and cycle time, and the cost of adding the hardware was minimal.

In the following example,

```
MOVL R0,-(SP) <----------> PUSHL R0
```

the MOVL and PUSHL instructions perform identical operations, but the PUSHL instruction does not explicitly specify a destination address. On previous VAX systems, the instruction prefetching would stall until the current instruction execution was completed. However, the VAX 9000 modifies such instructions during the decode stage by adding the implied specifiers. The benefits of this enhancement are more evident in the following instructions.

```
BSBW 10$ <----------> MOVAL Return_PC,-(SP)
RSB      <----------> JMP @(SP)+
```

Similarly, instructions such as LOCC and CMPC3 implicitly reference the general-purpose registers. The instruction decode stage creates a read/write mask with these references, which allows instruction prefetching to continue.

To aid handling instructions like PUSHR and CALL, the integer execution unit contains special bit mask manipulation hardware, which optimizes general-purpose register saves and restores. The VAX instruction set contains variable-length, bit-field instructions that handle non-byte data. These instructions can reference memory within a 512 megabyte (MB) range. The field referenced is within the first 8 bytes of the base address more than 95 percent of the time. Therefore, to allow instruction prefetching to continue, the operand processing unit assumes that the field is within the initial quadword and requests that data. If, during execution, the field destination actually resides outside the prefetched quadword, the correct data is fetched and the pipeline is flushed to avoid potential memory conflicts.

Integrating Vector Processing

The VAX 9000 project team was instrumental in integrating vector operations and data types into the VAX architecture. For many scientific applications, the use of vectors improves performance in three ways:

- Vector instructions specify many operations in a single opcode, which eliminates instruction stream decode as a processing bottleneck.
- Vector registers increase available local storage.
- Vector registers support high peak performance through high bandwidth and short access latency.

The VAX vector architecture implements a load/store architecture, which permits the hardware to deal with large pieces of memory in a uniform manner and increases the use of parallelism.

We added the vector instructions and data types to the VAX architecture in an integrated fashion. Scalar and vector instructions are mixed throughout the pipelines. Systems that do not include vector processors emulate vector instructions with software, a technique especially useful for program development.[7,8]

Logical Integration

The VAX 9000 vector processor connects to the scalar CPU as an additional functional execution unit. Vector instructions are processed, and operands are stored, in queues, the same as are scalar instructions. As instructions are issued, a control word is sent with instruction operands to the vector processor. The processor contains vector registers and arithmetic units. Addresses for load, store, gather, and scatter operations are also generated by the vector processor. Vector data is stored in the main cache, and both the scalar and vector processors have fast, shared access to that data.

Physical Integration

The VAX 9000 scalar and vector processors reside on a single planar board. Three multichip unit slots are reserved for the optional vector processor, which is field-installable. The integration of the vector processor directly with the scalar processor keeps critical interconnects short and reduces vector instruction overhead.

Error Handling

Reliability, availability, and integrity are critical factors in a high-performance computer system. These factors are affected by the quality of the physical design (i.e., worst-case design), effective cooling, redundant power supplies, and quality controls during manufacture. Still, failures are possible, and the VAX 9000 design had to deal effectively with errors.

Error handling in the VAX 9000 system has two main goals:

- Minimize system service disruption from individual failures
- Maximize the failure information collected for use in preventive and corrective maintenance

A large percentage of hardware failures are intermittent, and many solid hardware failures start as intermittent. The VAX 9000 system was designed to recover from these failures and to use the failure data to predict (and prevent) future problems.

To gather information effectively, VAX 9000 storage elements (i.e., latches, flip flops, and RAM cells) are visible to the service processor unit through a serial diagnostic bus. Most state information that is relevant to isolate the failing component is available for error analysis programs that can be run at a convenient time. The result of this processing is then used to isolate the failing components for quick repair.

To access the storage elements through the visibility chain, the system clocks must be disabled, which disrupts the system operation for a period of time. The error may also have affected the execution of the instructions in the pipeline. Error handling minimizes these disruptions by making them invisible to the users almost all the time.

The macroinstruction is the unit of execution in a program that is visible to the user. Between instructions, the program state is clearly defined in terms of memory contents and register values. Interrupts and exceptions are handled between instructions to save this state in an orderly fashion. It is important to handle errors the same way.

Two problems arose in trying to provide the same method of error handling. First, instructions go through many stages in a pipelined computer, and several instructions will be in progress at the same time. It is difficult to identify a beginning and end for each instruction. Second, even when boundaries are established, errors can occur at any time and the errors do not automatically line up with instruction boundaries.

To solve this, we made the E-box the point of synchronization between error handling and instruction execution. In the instruction execution model, the E-box accepts operands, then computes and delivers results for storage. If an error occurs that directly affects one of these steps, the error is synchronous to the execution of that instruction. Asynchronous errors do not directly affect any of these steps and are treated as interrupts, i.e., processed after the E-box completes an instruction but before it starts another instruction.

A synchronous error causes a trap to occur in the E-box when the E-box requests data from the subsystem with the error. Since such data can be unavailable as a result of virtual access problems, the E-box is ready to deal with exceptions at that time, and errors can use the same pipelined mechanism.

We do not differentiate between those synchronous errors that affect computation in the E-box and those that do not. Instead, if the program visible state of the machine has not been modified, the instruction is backed up to the beginning and restarted. Performing this task is not a problem, since the state is normally not changed until the result is stored at the end of the instruction. Errors occurring in early pipeline stages are easily recoverable. In a few cases, memory and registers could have been modified early and, as a result, be affected by the error. Status flags indicate if this has happened.

By getting to an instruction boundary, the clocks can be stopped in an orderly fashion, and the state can be read out, including temporary data to be used for failure analysis. The machine can be reset to start processing at the instruction boundary once the clocks are started again.

While the clock is stopped, the CPU cannot interact with other subsystems or I/O processors. To keep these functions from being blocked and possibly timing out, we only stop the clock to the CPU in error, not all the clocks in the system. We also sweep the cache of written data before the clock is stopped, and I/O interrupts are directed to other CPUs in a symmetric multiprocessing system.

Performance Modeling

When multiple features are added to a CPU design to individually enhance performance, some of those features can interact negatively with each other to decrease performance. Therefore, we designed a performance model to help us evaluate the performance of the design and make trade-offs where necessary. Although instructions were not executed on the model, it is an accurate cycle-by-cycle model of the system for most instruction operations. Equally important, the model was written at a high level, which made it easy to modify and use to experiment with different features before they were added to the design.

Cycle Time

A perennial CPU design issue is the trade-off between cycle time and cycles per instructions. In a VAX system, the cycle time is often limited by the RAM speed in the control store and cache. We modeled a machine at 8 ns and one at 16 ns for the VAX 9000 system. At 8 ns, the pipelines became longer. Although the peak throughput almost doubled, the model showed that the net performance gain did not offset the risks associated with the shorter cycle time.

I-stream Synchronization

The VAX architecture requires that changes to the instruction stream be synchronized with an REI instruction. This synchronization makes it easier to implement an instruction cache that is separate from the main cache. To synchronize, either all memory writes can be watched or the I-cache can be cleared on every REI. The first alternative entails high hardware costs, and the second can affect performance. However, the model showed us that the performance impact would be minimal if the

I-cache was refilled from the main cache rather than from main memory because the critical parameters were the main cache bandwidth and the I-cache invalidation time, rather than the refill latency.

Branch Prediction

The branch prediction scheme used in the VAX 9000 system was analyzed in great detail. We investigated the use of multiple history bits to improve the effectiveness of branch prediction. In all cases, the use of extra bits provided less than a 1 percent improvement in system performance. Furthermore, no multiple bit scheme could be implemented without increasing cycle time because multiple history bit branch prediction schemes update status each time a branch is encountered. Therefore, we chose to use a single-bit technique in the VAX 9000 design. Unlike multiple bit schemes that read and write history bits for each branch instruction encountered, the single-bit technique updates the history bit only when the prediction is wrong. The single-bit scheme is both faster and simpler.

Cache Parameters

The main data cache was accurately modeled. The VAX 9000 system uses a first-in first-out (FIFO) block replacement scheme. The performance model predicted that a true least recently used replacement policy would provide an insignificant improvement in performance over the FIFO method. Also, a true least recently used policy requires that status be read and written for each cache access. In contrast, the FIFO replacement policy updates status only when a cache miss has occurred. Further, the update can be done in parallel with the writing of data into the cache block. Although the 128-byte cache block provided a better cache hit, we chose the 64-byte block because it produced better system level performance.

We chose two-set associativity because the model clearly indicated that performance would degrade with a direct-mapped scheme. The model also predicted that a four-way set associative cache would not improve performance enough to justify the extra hardware, design complexity, and cycle time penalty.

The data bypass mechanism, the write queue, and the parallel translation buffer fix-up mechanisms were implemented after the performance model indicated significant performance gains would be achieved from these features.

We also used the performance model as a verification tool. The model provided us with early warnings when a feature did not function in the model, or when the cycle count differed from the count in the gate-level simulation. For example, from the model, we became aware of problems in the design of how conflicts between instructions in specifier processing were handled. Periodically, we compared the performance model to the logical model. Both models were subjected to the same instruction sequences. Deviations of more than ±5.0 percent were investigated. Some design bugs were found that did not affect the results of the program but which did keep performance features from working properly. The average deviation was on the order of ±1.0 percent.

Performance tests are among the first programs run on a functional prototype. The VAX 9000 system performed almost as expected. Table 2 compares the actual performance of a VAX 9000 system to its predicted performance for a small sample of modeled programs. The accuracy of the predictions highlights the increasing importance of models in the modern engineering process.

Table 2 Performance Measurements of a VAX 9000 System

Program Name	Predicted (VUPs*)	Measured (VUPs*)
HANOI	28.54	25.53
FFT45	36.87	37.85
GAUSS	32.72	32.57
WHETS	27.78	27.17
WHETD	34.48	34.89

* Performance measured in VAX units of performance (VUP), where the performance of the VAX-11/780 system = 1.0 VUP.

Vector Performance

Vector processing was modeled using graphical descriptions of the pipeline. The graphical descriptions were essentially critical path method scheduling charts. This approach is reasonable because vector processing makes regular demands on system resources. In fact, the regularity of resource demand patterns was a major reason that vector processing techniques were developed. By using the pipeline schedules, we realized that data should be prefetched to ensure good vector performance.

Performance Measurement

Table 3 compares the VAX 9000 scalar and vector processors performance to other members of the VAX family of processors.

Table 3 Performance of the VAX 9000 Scalar and Vector Processors

Program Name	VAX 8550 System (VUPs*)	VAX 9000 Scalar Processor (VUPs*)	VAX 9000 Vector Processor (VUPs*)
A3D	6.55	65.54	77.45
DYFESM	5.12	31.88	40.49
EMIT	5.86	41.65	79.86
CFFT2D	5.52	25.76	64.18
BMK8A1	5.45	30.65	83.84
MXM	5.93	40.81	269.32

* Performance measured in VAX units of performance (VUP), where the performance of the VAX-11/780 system = 1.0 VUP.

The variations in these performance numbers indicate that significant performance improvements can be achieved by using applications that take advantage of machine resources. The numbers also highlight opportunities. By modifying applications to capitalize on machine features, large performance gains may be realized. Performance gains of 100 to 200 percent are often realized and may substantially extend the lives of older programs.

Vector applications tend to fall into three categories. The first category generally does not contain much parallel content. This category is represented by A3D and DYFESM in Table 3. Vectorizing such programs improves performance by a modest 0 to 50 percent. Programs EMIT and CFFT2D in Table 3 represent the second category, which are applications of moderate parallel content. Applications in this category realize a 50 to 150 percent performance gain when vectorized. Applications in the third category, highest parallel content, demonstrate performance improvements of more than 150 percent when vectorized. Programs BMK8A1 and MXM in Table 3 are examples of this class of application.

Often, modest code changes can realize dramatic performance improvements. By simply redefining array dimensions or loop specifications, an application can move from the first category to the third category.

Acknowledgments

Many people contributed to reaching the VAX 9000 performance goals. The authors would especially like to thank David Orbits, whose advanced development work on high-performance VAX designs became the basis for the performance model; and Bill Grundmann, Rick Hetherington, John Murray, Bill Smith, and David Webb, who comprised, with the authors, the original VAX 9000 architecture team.

References

1. J. Murray et al., "VAX Instructions That Illustrate the Architectural Features of the VAX 9000 CPU," *Digital Technical Journal,* vol. 2, no. 4 (Fall 1990, this issue): 25–42.
2. M. Adiletta et al., "Semiconductor Technology in a High-performance VAX System," *Digital Technical Journal,* vol. 2, no. 4 (Fall 1990, this issue): 43–60.
3. SPICE is a general-purpose circuit simulator program developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.
4. D. Clark, "Pipelining and Performance in the VAX 8800 Processor," *Architectural Support for Programming Languages and Operating Systems* (ACM, October 1987).
5. C. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (ACM, March 1982): 177–184.
6. J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the 11th Annual Symposium on Computer Architecture* (Ann Arbor: June 1984): 301–310.
7. *VAX Vector Processing Handbook* (Maynard: Digital Equipment Corporation, Order No. EC-H0419-46, 1989).
8. R. Brunner and D. Bhandarkar, "Vector Extensions to the VAX Architecture," *Proceedings of COMPCON '90* (San Francisco: Spring 1990).

John E. Murray
Ricky C. Hetherington
Ronald M. Salett

VAX Instructions That Illustrate the Architectural Features of the VAX 9000 CPU

The VAX 9000 system is Digital's largest and most powerful VAX system. As such, it offers many unique features that required the use of advanced technology and innovative architecture in the design of the system. Overall, the VAX 9000 microarchitecture produces a high level of system performance and the lowest cycle time of any VAX processor, i.e., less than five cycles per instruction. Three sections of the VAX 9000 CPU—the instruction fetch and decode unit (I-box), the execution unit (E-box), and the data cache and main memory interface unit (M-box)—are illustrated in this paper through descriptions of a small sample of VAX instructions. These instructions are discussed in relation to their flow through the pipeline, how their architectural features combine to work on a single macro instruction, and how various stages of the pipeline interact.

In October 1989, Digital introduced its VAX 9000 family of high-performance scalar, vector, and parallel processors. The VAX 9000 system is designed to be expandable from one to four processors, with an optional integrated vector facility available on each processor. The design team obtained high levels of performance with advanced technology and innovative architectural features.[1,2] The technology provided a platform that has the shortest cycle time for any VAX processor. Most VAX processors average ten or more cycles per instruction, whereas the architectural features of the VAX 9000 system reduce that average below five.

The VAX architecture is a complex instruction set architecture. VAX instructions vary in length and number of operand specifiers. The opcode may be one or two bytes long. The number of specifiers is implied by the opcode. Each specifier's length is determined by the specifier type, and the length can vary by up to 17 bytes.[3] Although the VAX 9000 implements a large number of instructions in a single cycle, some instructions need to be implemented in tens of cycles. In these cases, microcode assistance is required. To increase performance, many features were included in the VAX 9000 system that have not been implemented in previous VAX systems. The system contains a virtual instruction cache, a branch prediction cache, multiple specifier evaluation units, deep instruction prefetch, hardware translation buffer fix-up unit, write address buffer and conflict checker, multiported write-back cache, independent arithmetic units, and separate issue and retire queues. These features are pipelined and do not interact in a straightforward way. Many stages are not directly linked to the subsequent stage but feed a queue or first-in first-out (FIFO) buffer. The subsequent stage works on the output of the FIFO buffer. The pipeline is not a fixed-length and many operations are done in parallel.

The architectural features do not function totally independent of one another. In fact, the highest level of performance is achieved when all the units function in harmony. This paper highlights the implementation of the macropipeline found in the three major subsystems of the VAX 9000. These subsystems are the instruction fetch and decode unit (I-box), the execution unit (E-box), and the data cache and main memory interface (M-box).

The design team for the VAX 9000 system's I-box evolved a cost-effective subsystem that outperforms all previous VAX systems. As shown in Figure 1, the I-box processes the majority of instructions in just one cycle. It combines a single cycle access virtual instruction cache with a 25-byte instruction buffer and an instruction decode cross bar that can decode three specifiers per cycle. To minimize cycle-wasting stalls, a branch prediction unit handles transitions from one code block to another. In addition, the operand processing unit receives and processes specifiers from the decode unit. The specifiers are passed either to the E-box as pointers, literal data or addresses, or to the M-box as virtual addresses.

Figure 2 illustrates how the front end of the M-box translates addresses by using either a translation buffer or an autonomous virtual-to-physical address translation unit. Physical addresses for reads are used to access a two-way associative write-back cache and to fetch data from memory through the system control unit (SCU), if the data is missing from the cache. Read data is returned to the E-box. Write addresses from the operand processing unit are translated and queued by the M-box until the E-box provides the data for the write.

The E-box of the VAX 9000 CPU performs all scalar operations. As shown in Figure 3, the E-box is a pipelined design that incorporates a microsequencer to control functional unit operation. Other dedicated control logic directs the flow through the pipe stages.

A multiported register file provides general-purpose registers and temporarily holds memory data. The data is processed by one of the four arithmetic functional units. Results pass through a retirement multiplexer to the register file or the M-box data cache, as shown in Figure 4. Multiple VAX instructions are executed concurrently in the E-box pipeline. The primary goal of the E-box is to produce a 32-bit result each cycle, which allows the majority of the simple, but most frequent, VAX instructions to be executed in one cycle. This goal is achieved when four requirements are met. First, the I-box must have commands available for the E-box. Second, operand data, often from the M-box data cache, must be available. Third, pipelined or single-cycle latency functional units are required for single-cycle throughput. Finally, results must be transferred from the functional units. E-box features, such as queues, data bypass paths, and powerful arithmetic units, help the system attain a high-performance level. Stalls are avoided and each instruction is executed in a minimal amount of time.

The M-box of the VAX 9000 CPU is the primary source of memory data. Therefore, it contains the virtual address translation buffer and the data cache. The M-box is multiported and pipelined with two autonomous pipeline segments. Each segment occupies one machine cycle, and the cache access latency is, therefore, two cycles long. During the first cycle, the M-box receives and prioritizes virtually (or physically) addressed memory requests. The M-box then indexes the translation buffer to produce a 33-bit physical address and to perform protection and validity checks. The second pipelined cycle involves data cache access, data alignment, if required, and port response. There are numerous architectural features within both segments that are targeted at high bandwidth for prefetching and storing scalar and vector operands.

To illustrate the various features of the VAX 9000 microarchitecture, we have selected the code sequence shown in Figure 5.[4] In the following sections, we discuss each instruction as it progresses through the pipeline as if it were the only instruction in the pipeline. We then summarize by considering the same instructions as a block of code.

VAX Instruction ADDL2

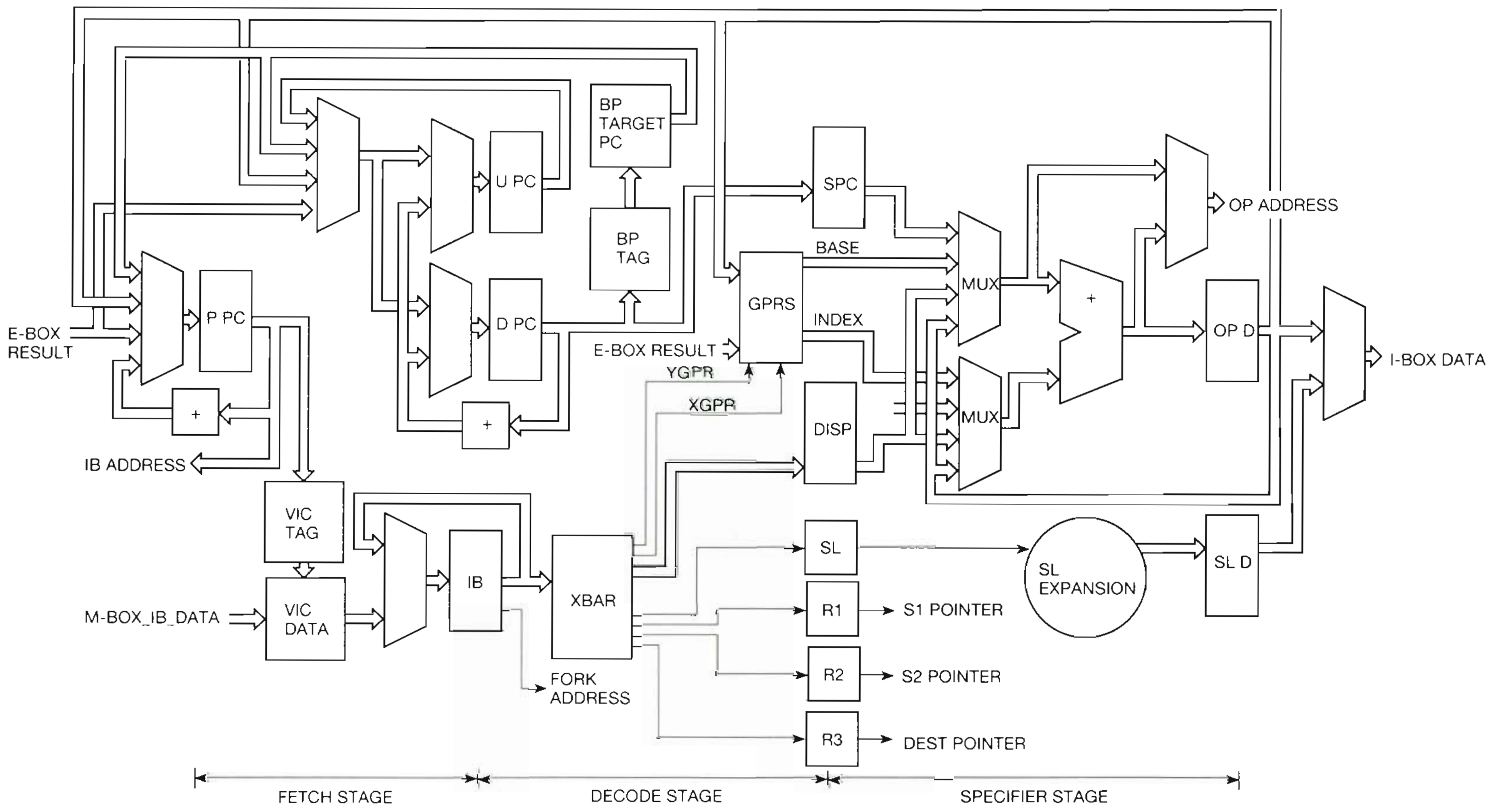
The ADDL2 instruction uses general-purpose register R8 as an address to memory. The contents of that location are added to general-purpose register R7, and the result is written back to the same location in memory. The instruction is encoded in three bytes: opcode, register, and base register.

Cycles One through Three

If we assume that the ADDL2 instruction is the first instruction either in an interrupt routine or following a context switch, the program counter is generated by the E-box and passed to the I-box on a 32-bit bus. The program counter is latched and used to access the virtual instruction cache during cycle one. The virtual instruction cache contains up to 8 kilobytes (KB) in 32-byte blocks and 8-byte lines of instruction stream data.

Bits <12:3> of the program counter's prefetch buffer are used to access an 8-byte line from the virtual instruction cache. Bits <12:5> are used to access a tag, a valid block, and four quadword valid bits. The tag is compared with bits <31:13> of the program counter's prefetch buffer. If the tag and the bits match, the block and the quadword within the block are valid, and the instruction is in the virtual instruction cache (i.e., a hit). Bits <2:0> of the prefetch buffer are used to rotate the quadword for the opcode byte to be loaded into byte 0 of the I-buffer at the end of cycle one. Similar to the VAX 8650 system, the first byte of the I-buffer is the operation code (opcode) of the instruction.[5]

The ADDL2 is three bytes long and normally fits in one line of the virtual instruction cache. If the ADDL2 instruction crosses a line boundary, a



KEY:

VIR	VIRTUAL INSTRUCTION CACHE
SI	SOURCE 1
S2	SOURCE 2
DEST	DESTINATION
IB	I-BUFFER
P PC	PREFETCH PROGRAM COUNTER
U PC	UNWIND PROGRAM COUNTER
D PC	DECODE PROGRAM COUNTER
S PC	SPECIFIER PROGRAM COUNTER
BP	BRANCH PREDICTION
PC	PROGRAM COUNTER
OPU	OPERAND PROCESSING UNIT
SL	SHORT LITERAL
GPR	GENERAL PURPOSE REGISTER
GPRS	GENERAL PURPOSE REGISTERS
XGPR	X GENERAL PURPOSE REGISTER
YGPR	Y GENERAL PURPOSE REGISTER
OP D	OP DECODE
SL D	SHORT LITERAL DECODE
R1	REGISTER 1
R2	REGISTER 2
R3	REGISTER 3
DISP	DISPENSER

Figure 1 Block Diagram of the VAX 9000 System I-box

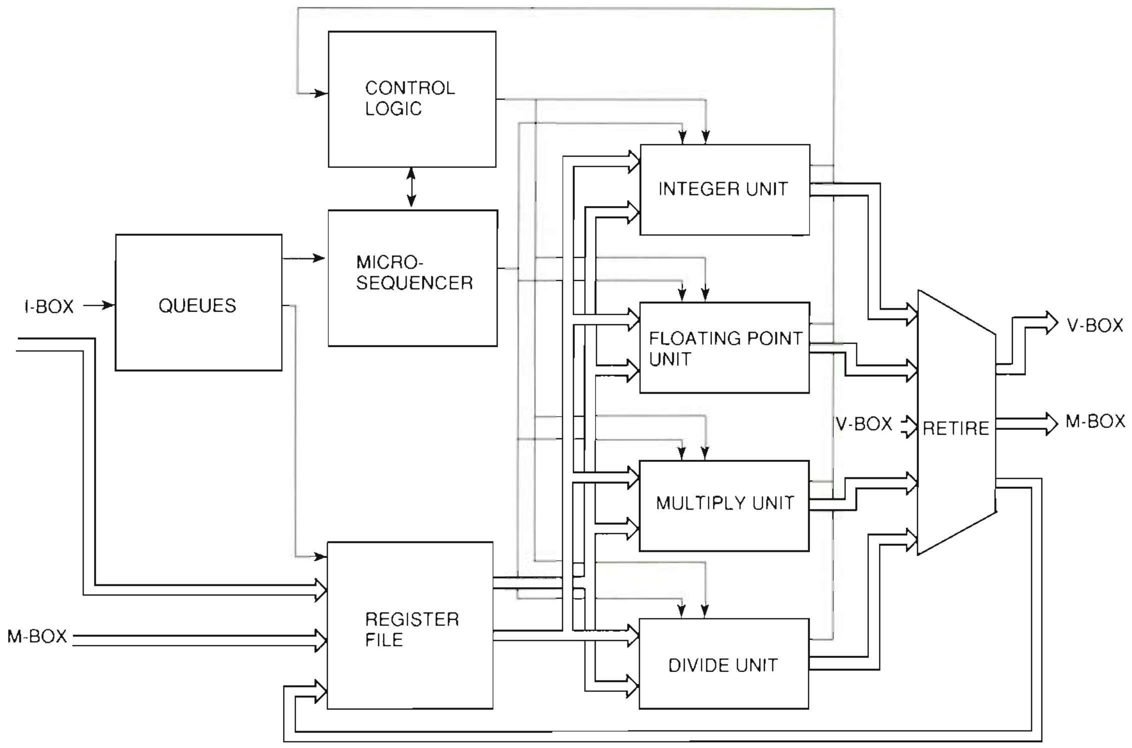


Figure 2 Front End of the VAX 9000 System M-box

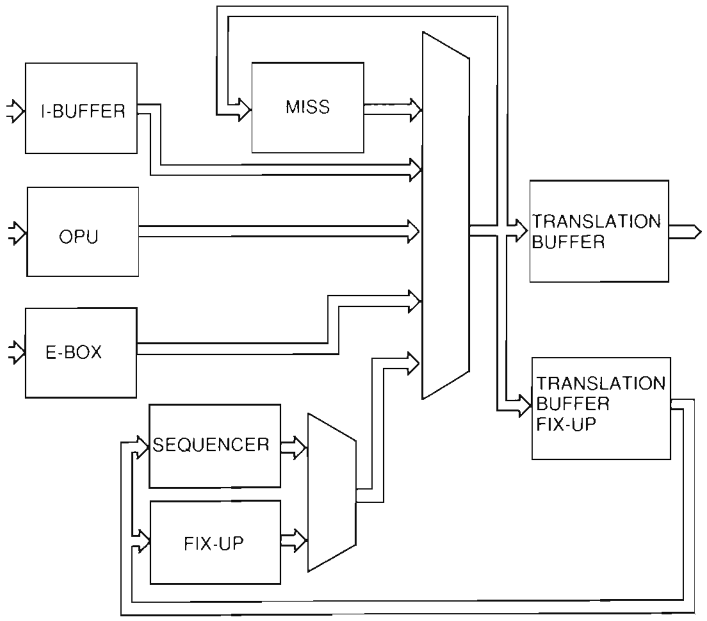


Figure 3 Block Diagram of the VAX 9000 System E-box

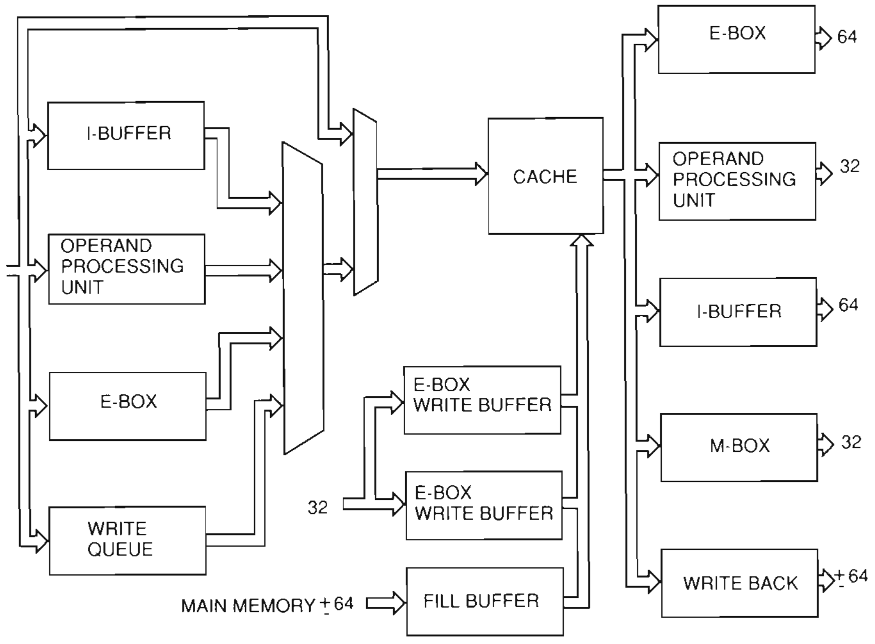


Figure 4 Cache Unit of the VAX 9000 System M-box

subsequent cycle is required to access the second line. The average VAX instruction is 3.8 bytes long. Therefore, a virtual instruction cache hit delivers about two instructions to the I-buffer.[6]

Other VAX processors generally require a cycle to decode the opcode and one or more cycles to decode each subsequent specifier.[7,8] However, the VAX 9000 CPU's instruction decode cross bar can decode the vast majority of common instructions in a single cycle.

If the three bytes of the ADDL2 instruction were loaded into the I-buffer at the end of cycle one, the bytes would be decoded during cycle two. The decode unit (XBAR) passes data from the I-buffer to a short literal unit, a register/pointer unit or an operand processing unit. As the opcode and specifier bytes are decoded in parallel, the XBAR determines in less than a cycle that both specifier bytes should be routed to the register/pointer unit and that the memory specifier should be routed to the operand processing unit.

In parallel with the XBAR decode process during cycle two, the program counter is passed to the E-box from the I-box. The opcode is used to address the fork random-access memories (RAMs) in the E-box that provide a fork address to the microsequencer. At the end of cycle two, the decoded bytes are shifted out of the I-buffer, and the subsequent instruction is presented to the XBAR in cycle three.

The fork address from the I-box is then used to address a fork RAM in the E-box. For each opcode, the fork RAM provides an entry address into the control store, indicates which functional unit should begin the execution, and specifies how many source operands are needed in the first cycle. The fork address is modified when an instruction

```
                  68  57      C0  0080  22   1$:  ADDL2       R7,  (R8)
       53       6044  00      41  0083  23        SUBF3     #0,5,  (R0)[R4],  R3
59 85 9999A999  535940C2  8F  45FD  0088  24        MULG3  #2345.5,  (R5)+,    R9
       E3   00000121'  EF 0D  E4  0095  25        BBSC       #13,  BDATA,     1$
```

Figure 5 VAX Instructions That Illustrate the Major Features of the VAX 9000 System

is restarted after it was interrupted before completion. Memory management faults on the instruction stream also modify the fork. At the end of cycle two, the fork RAM data is latched in a fork queue, and the instruction program counter is latched in the program counter queue.

The register/pointer unit accepts the register and specifier byte at the end of cycle two. During cycle three, the register/pointer unit passes two source pointers (general-purpose register R7 and memory data) and the destination pointer (memory destination) to the E-box. The source one pointer points to general-purpose register R7. The memory data will be returned eventually to a 16-bit deep circular queue, called the source list, in the E-box. The register/pointer unit tracks the source list pointers and allocates a source list entry to the memory data. The source list address is passed to the E-box and the operand processing unit. The destination pointer simply indicates that the result of the instruction goes to memory.

Further, during cycle three, the operand processing unit generates the memory address and passes it to the M-box. For the register deferred specifier (R8), the operand processing unit accesses its local copy of R8 and passes it to the M-box, together with the source list tag received from the register/pointer unit and a control function that indicates the memory location is to be read and then written.

The fork queue is a cyclical, eight-entry FIFO buffer that is flushed for interrupts, exceptions, or incorrect branch predictions. For the ADDL2 instruction, the queue passes part of the fork RAM data to the microsequencer, which is idle and awaiting a valid fork, early in the third cycle. Fork RAM data is used to generate the appropriate control store address for all control store RAMs. The remaining fork RAM data is passed to the issue control by the end of the third cycle.

Cycle Four

At the start of cycle four, the M-box receives a command from the operand processing unit to perform a read with a write-check. The M-box must read a longword from memory, send the longword to the E-box, and check for write access. The command is accompanied by a 32-bit virtual address, a tag field, context (size of operand), and the request signal.

Arbitration for access to the translation buffer occurs every cycle. If the operand processing unit wins arbitration, the command is decoded and the context is checked against the starting address to determine if additional virtual addresses are required. The M-box includes a feature that adds four addresses to E-box or operand processing unit addresses, if the size and alignment of the request crosses a quadword boundary. Other VAX systems trap on unaligned accesses using E-box cycles and require using microcode to generate the incremented address and subsequent fetch.

In parallel to the arbitration process, virtual bits 31, <17:09> index the 1024-entry translation buffer. The translation buffer is a direct-mapped, associative memory that contains the results of the most recent 1024 translations. Bits <30:18> are compared, validated, and protection-checked against the tag field. The physical frame number is a 24-bit field that is appended to the virtual address bits <9:0> to create the 33-bit physical address. The self-timed RAM used for the translation buffer is a 1024 by 4 self-timed RAM with a 4.5 nanosecond (ns) access time.

Protection checking occurs during the latter portion of cycle four. The example we are discussing is a request for a read and write check. Therefore, both read and write access are checked. Fault indication is forwarded with the request to the data cache and subsequently, with the data, to the E-box. If the request has a valid entry in the translation buffer and no protection violations exist (i.e., translation buffer hit), a data cache access is required in cycle five.

The two source pointers and the destination pointer from the I-box are latched in the source and destination queues, respectively, at the start of cycle four. The source queue holds 16 entries and can receive 2 entries per cycle. The destination queue holds eight entries. Both queues are circular FIFO queues that can be flushed with the fork queue. The two source pointers are also latched in the source operand logic at the start of cycle four. The source operand logic determines which two source pointers to use each cycle. The pointers can come from the source queue, the I-box, the microword, the register log, and several special functions. In this example, the two pointers are selected directly from the latched I-box pointers because using the source queue would have required an extra cycle.

The selected pointers address the register file and are passed to the issue logic early in the fourth cycle. The register file contains the 15 general-purpose registers, R0 through R14. These registers can be written by either the E-box or the I-box for autoincrement or autodecrement specifiers. The first pointer accesses general-purpose register R7. The contents of general-purpose register R7 are passed to the data distribution logic by the end of cycle four. The second pointer accesses one of the 16 locations in the source list. The source list is a queue for source operand data that is written by the I-box, with immediate or short literal data, or by the M-box, with memory data. The pointer is used to access the appropriate source list register, and the data is passed to the data distribution logic.

The issue control uses the fork RAM data and the source pointers to determine if the instruction can be executed. The issue control checks that the target functional unit is ready and that all the required source operands are available. In this example, the integer unit is ready, the first operand (i.e., the general-purpose register R7) is available, but the second operand (i.e., memory data) is not. Because normal issue cannot occur without the second operand, we created a special issue control to handle this case. When issue is prevented only by the lack of a single memory operand, the instruction is "issued with bypass." To save an operational cycle, when the M-box delivers the missing operand, that operand bypasses the source list and passes immediately to the waiting functional unit. The issue control signals the fork and source queues that entries were used and can now be removed.

Cycle Five

Cycle five begins with the cache data and tag self-timed RAMs latching the physical address. The priority request was selected by the cache control in the latter portion of cycle four. The default priority request selection is the write queue. However, if the default is used and a translation buffer hit occurs, the current address from the translation buffer is used. The first stage of the M-box, or the translation buffer stage, is referred to as the front end. It provides the cache with a 4-bit cycle-identification field that identifies the command type and port. In addition, a context field provides the cache with the data size. The request for the second specifier of the ADDL2 is a cache read and a write check.

The write queue is a key feature of the M-box and the VAX 9000 system. The write queue is an 8-entry FIFO buffer that holds pretranslated operand memory destination addresses and allows the operand processing unit to continue prefetching operands after memory destination operands. The write queue is designed to be a content-addressable memory that checks for memory conflicts as subsequent memory source operands are accessed. Entries in the write queue are discarded when the E-box completes execution and a successful cache write occurs.

A write check inserts the physical address and a number of status bits into the write queue. The status bits are termed valid, fault, twice (asserted for the final entry when writing unaligned operands that require more than one address), last, PCD (page crossing danger alert), and blocked. The write check illustrated in this example has a valid and last flag asserted.

The cache tag is accessed in the middle of cycle five. The cache tag store is two-way set associative, with 1024 entries per set. Each entry represents a 64-byte block of memory data. The tag store also contains 16 valid bits (i.e., one per longword) and a written bit because memory update requires a write back. The cache tag is used to determine if the requested data resides in the data cache and, if not, whether the cache data held there needs to be written back to memory.

If a cache hit occurs, the data, with an asserted response line and tag, is sent to the E-box. The tag field tells the E-box where to place the field in the source list. In cycle five, the issue logic asserts a microword hold signal throughout the E-box. Because execution did not occur in cycle five and the latched microword was not used, the microword latches must be held until execution can occur.

Cycles Six through Ten

At the start of cycle six, the M-box data is latched to the data distribution logic. The data is immediately passed to the integer unit, where the add operation is performed. The results of that operation are sent to the retire logic by the end of cycle six. The issue logic deasserts the microword hold signal to allow subsequent microwords to become latched. The issue logic also makes an entry in the eight-location result queue. The result queue is used to maintain write ordering when multiple functional units are operating in parallel and also acts as a scoreboard for register conflicts. A general-purpose register is not a valid source operand if the register is in the result queue waiting to be updated by a functional unit. When the functional unit specified by the top entry of the result queue completes the operation, the results are retired and the queue entry is discarded. The integer unit always completes in a single cycle. Therefore, the ADDL2 instruction is discarded from the result queue by the end of cycle six.

In cycle seven, the retire multiplexer selects the integer unit result data and sends it to the M-box to be written. A request signal for an op write also is sent for the M-box to initiate the write. The M-box indicates that the address translation was successful to ensure that a memory management fault does not occur. A signal that the instruction is done removes the instruction program counter from the program counter queue and indicates successful address translation completion to the I-box.

At the start of cycle eight, the E-box sends the results of the ADDL2 instruction to the M-box to be written into the data cache. The write data interface from the E-box to the M-box is 32 bits wide. The M-box has two 32-bit data buffers that receive the write data and hold the data until the tag status is appropriate for the write to occur. The E-box signals the M-box to perform the write. This signal does not affect the front end of the M-box because address translation has already occurred. The signal is a request to the cache control and arbitration for an op write. The top entry of the write queue holds the status and complete physical address for the write destination of the ADDL2 instruction. Because this is a modify operand, write access is checked and reported to the E-box when the operand is read from memory.

A two-cycle cache write starts in cycle nine. The first cycle is the lookup cycle. The cache control selects the write queue address to address the cache. Before the write can occur, the cache block must be dedicated to this particular physical address.

The cache tag store and data cache are read in parallel. If the operand is unaligned or is less than a longword, the cache data line read during the lookup cycle is captured and merged in the next cycle with the data from the E-box. To ensure data consistency, data is allowed to exist in the cache in one of two states, read only or written. The SCU controls the data in up to four individual CPU caches. Read-only data may be valid in multiple caches, but written data may only exist in one cache. Thus, before an M-box can change data, it must ask permission from the SCU.

In this example, the operand is longword-aligned, and the write occurs in cycle ten if the block in the cache tag store has a valid and written status. The write also occurs for aligned longwords and quadwords in cycle ten if the cache block is completely invalid. If this cache block has a read status (i.e., valid and not written), a command is sent to the SCU to request permission to write. The write is delayed until the SCU responds with approval to write.

Cycle ten is the cache data write cycle. Write enables to the self-timed RAMs in the data cache are asserted in this cycle as a result of the tag store lookup cycle in cycle nine. The tag store is updated only if a valid bit had to be asserted. However, a partially valid and written cache block that may require setting the appropriate valid bit can be written. Each line of the cache is protected with byte parity. However, because it is a write-back cache, the reliability of the machine is significantly enhanced using an error-correcting code. The check-bit pattern for the error-correcting code storage is generated and stored separately.

VAX Instruction SUBF3

The SUBF3 instruction subtracts one half of the F_float format number addressed by general-purpose register R8 and indexed by general-purpose register R9. The resultant number is placed into general-purpose register R3. The instruction is encoded in five bytes: opcode, short literal, index register, base register, and destination register.

Cycle One

As with all VAX instructions, the first cycle of the SUBF3 may be either a virtual instruction cache access or a simple shift to the low-order bytes of the I-buffer. In a few cases, the instruction not in the cache. Consider an example where the SUBF3 instruction is preceded by a long instruction that is gradually decoded for several cycles. As a result, the SUBF3 instructions cross a virtual instruction cache block boundary. The virtual instruction cache is accessed for instruction stream data every cycle, but the address is incremented only when the data is latched in one of three instruction stream buffers, I-buffer (9 bytes), I-bex (8 bytes), or I-bex2 (8 bytes).

If only one byte or if no valid bytes exist in the I-buffer, the virtual instruction cache data is loaded directly into the I-buffer. If the I-bex is empty and the I-buffer contains between two and eight valid bytes, the virtual instruction cache data is merged with the the I-buffer data and the remaining bytes are loaded into the I-bex.

If there is a virtual instruction cache miss (i.e., the data is not in the virtual instruction cache) and there are no valid bytes in the I-bex, the I-box passes a request to the M-box for data. Generally, the request will be for a virtual instruction cache block, i.e., 32 bytes, because the I-box decodes instructions sequentially across a block boundary. However, a branch or interrupt may direct the decode into the middle of a block. In such a case, the I-box requests the remainder of the target instruction stream in that block. The first cycle of the SUBF3 accesses the virtual instruction cache only to find what data is not in the cache. The address and the request then are passed to the M-box.

In the next cycle, the result of the virtual instruction cache tag match is signaled to the M-box through an I-box abort signal. If a virtual instruction cache miss has occurred, the block in the virtual instruction cache is cleared and the I-box awaits the data from the M-box. The M-box passes eight bytes of data to the I-box. The third cycle after a virtual instruction cache miss is found in the I-box when hits occur in both the translation buffer and the cache. The data sent to the I-box is written and read through the virtual instruction cache self-timed RAMs to the I-buffer or the I-bex. The virtual instruction cache's tag, valid block, and the relevant valid quadword are written next. If the I-buffer is empty, the eight bytes of SUBF3 instruction are written into the I-buffer. In subsequent cycles, the following instruction stream quadwords are written into the virtual instruction cache, with a new quadword valid bit, until the end of the block is reached. The data is also available to the I-buffer.

Cycle one of the SUBF3 instruction could be fetching the instruction stream from the virtual instruction cache, as described for the ADDL2 instruction, or it could be already in the I-buffer (e.g., bytes <8:3>) following the ADDL2 instruction (i.e., bytes <2:0>). In the latter case, the SUBF3 instruction would be shifted into the lower bytes as the ADDL2 instruction is shifted out.

Cycles Two through Eight

In cycle two, the SUBF3 instruction is completely decoded and shifted out of the I-buffer. As a result, the following actions occur:

- The fork address is passed to the E-box.
- The short literal is passed to the short literal expansion unit.
- The base and index registers are passed to the operand processing unit.
- The destination general-purpose register R3 and the two sources are passed to the register/pointer unit.

During cycle three, the register/pointer unit allocates the next available entry in the source list to the short literal and the subsequent entry in the indexed memory reference. The E-box is informed of these allocations as pointers to the relevant entries are passed to the pointer queues in the source one and source two pointers. The register/pointer unit also passes the destination register to the destination queue in the E-box.

The operand processing unit passes the tag, with the address for the indexed memory specifier request, from the register/pointer unit to the M-box. The address is generated by the adder in the operand processing unit. In parallel with the operand processing unit and register/pointer unit, the short literal expansion unit takes the 6-bit field and expands it to a 32-bit F_floating number.

During cycle four, the short literal is written through the I-box data bus to the relevant entry in source list. Issue control can issue with bypass because only the memory data for operand two is missing.

The E-box stalls until the memory data arrives. Because the I-box and the M-box generally are functioning ahead of the E-box, memory stalls are short or nonexistent. In this example, the memory data arrives at the end of cycle five, as was the case with the ADDL2 instruction.

In cycle four, the M-box operates for the SUBF3 instruction in a similar manner to its cycle four activity for the ADDL2 instruction. At the start of the cycle, a command, address, context, and tag field are sent from the operand processing unit to the M-box. The command is a simple operand read. Arbitration occurs early in the cycle. The translation buffer is then accessed, and the physical address is sent to the cache.

Cycle five begins when the data cache receives the physical address for the operand processing unit to read. The tag store lookup and address matching are performed simultaneously with the data read, and the data is available to the E-box at the end of the cycle. If the operand read results in a cache miss, the M-box must assemble a command and an address, which are sent to the SCU to enable the SCU to access a 64-byte block of memory data. In addition, the data cache tells the SCU which set the cache will replace with the new cache block. If the current cache block contains valid and written data, the block must be written back to main memory before the new cache block arrives.

The SCU sends a command and an address back to the M-box when the memory data is ready. The send takes approximately 26 cycles and is followed, within a short period of time, by eight cycles of data transfer. Each cycle is 8 bytes long. The requested quadword is returned first to respond to the requesting port during the first cycle of the cache refill. On the eighth cycle of cache refill, the tag store is updated.

The floating point functional unit is started in cycle six, as specified by the fork RAM data. Both source operands are delivered, and the microword indicates a SUBF operation. The floating point unit requires two cycles to perform the SUBF operation. Unpacking and alignment occur in the first cycle. The floating point unit signals the issue control that the result will be available at the end of the following cycle. The issue control enters the general-purpose register R3 destination but must wait another cycle before beginning retirement. If the next instruction requires that the floating point unit and the operands be available, the instruction would be issued in this cycle because the floating point unit is fully pipelined.

The second execution cycle occurs in cycle seven. The floating point unit adds, normalizes, rounds, and packs. The result is latched in the floating point unit at the end of the cycle, and the issue control discards the top entry from the result queue to retire the data.

In cycle eight, the retire multiplexer selects the floating point unit result data and sends that data to the data distribution logic. The data distribution logic holds the result, which will be written into general-purpose register R3 in the register file during the next cycle. The write is purposely delayed to permit it to be aborted if an arithmetic fault occurs. By holding the result in the data distribution logic, result bypassing into the data path can act as a source operand. The result is written into the register file at the beginning of cycle nine.

VAX Instruction MULG3

The MULG3 instruction takes the G_format floating number, addressed by general-purpose register R5, from the instruction stream, multiplies it by the immediate constant 2345.675, which is also a G_format number, and puts the result in general-purpose registers R9 and R10. General-purpose register R5 also is incremented by eight as a side effect of the specifier evaluation. The opcode is 2 bytes long, the constant is a nine-byte immediate specifier, and the autoincrement and register specifiers are each a single byte. Thus, the instruction is encoded in 13 bytes.

Cycles One through Five

As in cycle one of the SUBF3 instruction, the MULG3 instruction can either be a virtual instruction cache access cycle or part of the instruction already can be in the I-buffer and shifted to the least significant byte as the previous instruction is shifted out. For example, if the previous instruction is the SUBF3 #0.5 (R0)[R4] R3 in bytes <4:0> of the I-buffer, the first four bytes of the MULG3 instruction could be in bytes <8:5>. The four remaining bytes of the immediate specifier could be valid in the I-bex and the rest of the instruction could be contained in the I-bex2. At the end of cycle one, the first four bytes are shifted to the low four bytes of the I-buffer. The next four bytes are merged from the I-bex to the high four bytes of the I-buffer. The I-bex is now empty, and the bytes in the I-bex2 can be loaded into the I-bex.

Because the MULG3 instruction has a 2-byte-long opcode, the only decoding necessary in cycle two is to note the 2-byte length and shift out the first byte so as to align the specifiers to be the same as a single byte opcode instruction. The specifiers are then in bytes <1:8> of the I-buffer. As the first opcode byte (in this case, #FD) is shifted out, the next valid byte in the I-bex is merged into byte 9 of the I-buffer, which leaves seven valid bytes in the I-bex.

Decoding really begins in cycle three. The fork's address is sent to the E-box, and bit <8> is set to indicate a 2-byte-long opcode. The first five bytes of the immediate specifier are passed to the operand processing unit. The first byte also is passed to the register/pointer unit for source list allocation. The five bytes shifted out of the I-buffer are replenished from the I-bex, which leaves two valid bytes in the I-bex.

In cycle four, the register/pointer unit allocates the two entries in the source list for the immediate G_floating number by passing a source one pointer to the E-box and the tag to the operand processing unit. The operand processing unit passes the first longword of the immediate G_floating number to the unit's output buffer.

The next four bytes of the immediate are passed from the I-buffer to the operand processing unit. The remaining two valid bytes from the I-bex are merged into the I-buffer. The I-bex is then loaded with eight bytes from the virtual instruction cache.

In cycle five, the autoincrement and register specifiers are decoded and the remaining bytes of the instruction are shifted out. Five bytes from the I-bex are merged with the four valid bytes in the I-buffer. The autoincrement general-purpose register R5 is passed to the operand processing unit and the register/pointer unit, which also receives general-purpose register R9. The first longword of the immediate specifier is passed from the operand processing unit output buffer, through the I-box, to the source list entry allocated by the register/pointer unit. The second longword is passed to the operand processing unit output buffer.

The first microword is accessed and distributed throughout the E-box. The microsequencer uses the fast fields of the microword to generate the final control store address for this instruction. The microinstruction is not issued because it requires two source operands and the second source pointer is not yet available.

Cycle Six

In cycle six, the register/pointer unit allocates two source list entries for the autoincrement specifier, passes this information to the E-box in the source one pointer, and passes a tag to the operand processing unit. The general-purpose register R9 is passed to the E-box as the destination pointer.

The operand processing unit accesses general-purpose register R5 and passes it, with a tag and a quadword read request, as an address to the M-box. In parallel, the operand processing unit writes general-purpose register R5, incremented by 8-byte lengths in the unit's output buffer. The second longword of the immediate specifier is written to the source list at the relevant entry.

The operand processing unit sends the M-box a read request quadword for the double-precision floating point operand. If the address is on a quadword boundary, the front end of the M-box will not produce any additional virtual addresses because the operand will not cross a page boundary or a cache line boundary. If there is a miss in the translation buffer for this reference, all other arbitration stops and control are given to the state machine of the translation buffer fix-up unit.

Bits <31:09> of the request are captured by the translation buffer's fix-up unit in parallel with the translation buffer RAM's access to achieve an early start on miss processing. The fork to the state machine is sensitive to bits <31:30> of the virtual address. Therefore, when a translation buffer miss occurs, a constrained control word flow begins based on the values of bits <31:30>. Because this is a user mode, the value is zero. Therefore, on the first cycle following the translation buffer miss, the virtual page number is compared against the P0 length register, P0LR. On the next machine cycle, the P0BR (i.e., base register) is added to the virtual page number to create the system virtual address of the process page table entry. The fix-up unit acts the same as any other port into the translation buffer, and makes a virtual read request with an aligned longword context. The state machine is controlled by a microword that branches to itself until one of three events occurs: a miss in the translation buffer (the fix-up unit processes double misses), a memory management fault, or a cache response. The cache response, which is the event most likely to occur, signals the state machine to return to idle and prepare for the next miss. Hardware control external to the fix-up unit writes the entry into the translation buffer, and the original request is retried. This time there is a translation buffer hit, and the physical address is sent to the cache. Single misses in the translation buffer require seven cycles to process. A double miss requires 13 cycles, assuming data cache hits occur.

The issue control asserts the microword hold signal to force the microword latches to hold the first microword until it can be executed. The microsequencer regenerates the control store address of the second microword each cycle until the execution stall ends.

Cycles Seven through Thirteen

Cycle seven is the data cache read cycle for the quadword operand processing unit request that was translated in the previous cycle. The VAX 9000 system has a 128KB data cache, with a block size of 64 bytes and access width of 8 bytes. The 64-bit access width matches the 64-bit data path to the E-box, which was constructed to provide high bandwidth for double-precision operand transfers. When a cache hit results for the read of an aligned quadword, both the normal response line and the quadword response signal are asserted to alert the E-box that the M-box is sending a quadword of data.

In cycle seven, general-purpose register R5 of both the E-box and I-box is written with the incremented value. In addition, both source pointers and the first source operand are available to the issue control. Because only the second operand is missing, the microinstruction can be issued with bypass awaiting memory data.

The quadword operand is available to the M-box at the end of cycle eight. The low longword is latched in the data distribution logic of the E-box, and the high longword is held in the M-box.

In cycle nine, the quadword operand is written into the register file at the two source list locations allocated by the operand processing unit. However, the low longword is available as a source immediately. The low longword of the short literal operand and the low longword of the memory operand are passed to the multiply functional unit at the start of cycle nine. The multiply unit performs the first cycle of execution, which includes unpacking and multiplying the most significant bits of the two operands. Issue control drops the microword hold signal to allow the second microword to be latched. An entry, which specifies general-purpose register R9 as the destination for the low longword of the result, is made to the result queue. The second microword is issued because the multiplier requires the next half of each source operand and both are available from the register file.

The microsequencer then attempts to generate a new control store address from the next entry in the fork queue. If no new forks are available, the microsequencer remains idle.

In the tenth cycle, the multiply unit receives the high longword of both source operands. The second execution cycle is performed, which includes unpacking and three simultaneous multiplications of the appropriate combinations of the most and least significant bits of the two operands. The multiplier signals the issue control that the result will be available in the following cycle. The issue control makes an entry, which specifies general-purpose register R10 as the destination for the high longword of the result, in the result queue. The multiply functional unit is fully pipelined and could be issued in this cycle to start subsequent operations.

Cycle eleven is the third and final execution cycle. The multiplier accumulates the four products it produced in the two previous cycles, rounds, and packs the final double-precision result. The issue control discards the top entry from the result queue to retire the low longword of the result.

In cycle twelve, the retire multiplexer selects the multiply unit result data and sends it to the data distribution logic. The issue control discards another entry from the result queue to retire the high longword of the result. The low longword of the result is written into the register file's general-purpose register R9 in cycle thirteen. The high longword of the result is written into general-purpose register R10 in the next cycle as the instruction is completed.

VAX Instruction BBSC

The BBSC instruction tests a bit in memory, branches if the bit is set, and clears the bit. The BDATA is the base address in memory with the number 13 position-bit offset. The majority of VAX field instructions have a position offset of less than 64 bits. Therefore, the VAX 9000 system's I-box prefetches the quadword addressed by the base. As with all conditional branches, the result of the test is predicted and the VAX 9000 system's I-box continues to fetch instructions along the predicted path. The BBSC is encoded in eight bytes: one opcode, one short literal position, five for the base address (a 4-byte displacement off the program counter), and one displacement.

Cycles One and Two

Cycle one for the BBSC can be fetching the instruction stream from the virtual instruction cache, as described for cycle one of the ADDL2 instruction, or it already can be in the I-buffer (e.g., bytes <8:3>) and the I-bex (i.e., bytes <7:6>) following the MULG3 (i.e., bytes <2:0>). In the latter case, the BBSC instruction is shifted into the lower bytes as the MULG3 instruction is shifted out.

The decode of the BBSC begins with passing the short literal, number 13, to the short literal expansion unit and the program counter/relative base address to the operand processing unit. Information on both specifiers is passed to the register/pointer unit. In this cycle, the fork address is also passed to the E-box. The fork address is modified for field instructions if the base is a register. Therefore, passing the fork address is delayed until the base specifier is decoded. In this example, the base is decoded in the cycle after the opcode is received. If the base is a register, the field instruction takes a different microcode flow.

During cycle two, the decoder passes the program counter decoder for the program count of the instruction to be decoded to the operand processing unit. The program counter is passed to the operand processing unit and the E-box in the first decode cycle. Whenever a specifier is passed to the operand processing unit, the XBAR also sends a specifier offset delta. When the delta is added to the program counter's decoder, the address of the last byte of the specifier plus one is produced.

As the short literal and program counter/relative specifiers are decoded, they are discarded from the I-buffer. The BBSC displacement is shifted to the first byte of the I-buffer. The data arriving from the cache is merged into bytes <8:2>, and the other byte is placed in the I-bex.

The branch prediction unit begins operating during the first decode cycle. A prediction for the branch must accompany the fork address sent to the E-box. The prediction is made by using the program counter to access a branch prediction cache and determine how the branch behaved the last time it was decoded (i.e., one history bit). If the branch is in the cache, the prediction is that the branch will behave the same as the last time. If the branch is not in the cache, a prediction is made based on the normal behavior of this conditional branch. For example, a BEQL (58 percent) and a BBSC (73 percent) normally do not branch, whereas a BNEQ (62 percent) normally branches. If the BBSC instruction is in the cache and branched last time, this information is indicated to the E-box, with the I-box prediction given as true.

Cycle Three

In this cycle, the register/pointer unit allocates one entry in the source list for the position specifier and three entries for the base specifier. The unit then passes the source one, source two, and destination pointers to the E-box.

In the operand processing unit, the address of the last byte of the specifier plus one is first calculated using the program counter of the instruction and the delta provided by the XBAR. The displacement from the instruction is then added to this calculation. The result is latched in the operand processing unit's output buffer and passed to the M-box. The operand processing unit also passes a quadword, field modify function, and the source list tag.

The short literal expansion unit extends the size of the position specifier to a longword and latches it in the unit's output buffer. In this example, the extension is done with zeros. The XBAR passes the branch displacement byte and an updated value of the program counter's delta to the operand processing unit. The delta of the program counter and the branch displacement are also sent to the branch prediction unit as instruction lengths. The BBSC instruction is completely decoded, and the opcode and displacement are discarded from the I-buffer. The branch prediction unit does most of its work during the last decode cycle of a branch. For the majority of conditional branches, the last decode cycle is also the first.

The branch prediction cache contains 1024 entries. Each entry has a history bit, a 32-bit target program counter, a 6-bit instruction length, and a 16-bit branch displacement and its tag. The entries are addressed by bits 9 through 0 of the program counter's decoder. If the tag matches bits <31:10> of the program counter's decoder, the entry is assumed to be the entry, or a hit, for this branch.

If a hit occurs and the history bit shows that the branch was not taken last time, the branch prediction unit latches this state information and allows the subsequent instruction stream to be decoded. The operand processing unit produces the target address as soon as it is not busy. The target address must be stored in the program counter's unwind buffer in case the prediction is incorrect. The E-box indicates the correctness of the prediction as soon as possible. For simple branches, the E-box could indicate that the prediction is incorrect before the branch is fully decoded.

If a hit occurs but the history bit shows that the branch was taken last time, the branch prediction unit latches this state information and stops the decoding of the subsequent instruction stream by clearing the I-buffer and the I-bex. The program counter of the subsequent instruction is stored in the program counter's unwind buffer. The program counter's target address, which is received from the branch prediction unit cache, is passed to the program counter's prefetch buffer. The target address that is later provided by the operand processing unit may be discarded. The branch displacement and instruction length from the branch prediction cache are latched. For the following discussion on the remaining cycles in the BBSC instruction, we have assumed that the BBSC instruction is a branch prediction hit and that the branch was taken the last time decoding occurred.

Cycle Four

In cycle four, both the operand processing and short literal expansion units contain data to be passed to the source list. The operand processing unit normally has the higher priority of the two. Therefore, the short literal expansion unit will stall. The operand processing unit passes the base address to the source list through the I-box. In the operand processing unit, the new delta of the program counter is added to the program counter, the sign of the branch's displacement is extended from a byte to 32 bits, and the two are added to produce the new target address. The result is latched in the operand processing unit output buffer.

The virtual instruction cache is accessed for the target instruction. If the instruction is in the virtual instruction cache, it is passed to the I-buffer. However, there is a gap in the pipeline because no instruction can be decoded this cycle.

The displacement and instruction length from the branch cache are compared with the actual displacement and instruction length. Normally, these lengths match. However, if they are different, the target address from the branch prediction unit cache is probably incorrect. The fetching and decoding of instructions must wait until the operand processing unit provides the correct address.

At the start of cycle four, the M-box receives a request from the operand processing unit. This request differs from all requests previously described in that it contains a command that gets special treatment in the M-box. The command is an "opu read with write check no block."

The command is used because the VAX 9000 CPU contains an optimization that enhances the performance of bit field instructions. With this command, the operand processing unit prefetches a quadword of data, starting from the address pointed to by the base, without looking at the value of the position operand. Hopefully, the majority of bit fields are within 64 bits of the base. The special command tells the M-box that if a fault should occur, it should pass the fault, with an operand, to the E-box and not close down the operand processing unit port or put a lock on the fault parameters. The command is an unaligned quadword operand and, as such, requires that the M-box produce additional virtual addresses to correctly access the cache. A quadword is unaligned when bits <2:0> are nonzero. For this example, we have assumed that the starting address is xxxxxxx1.

Specialized hardware in the front end of the M-box detects if the starting address requires sequencing (i.e., the addition of a constant of 4 to the current address) and how many sequenced addresses are necessary. In this case, three addresses are required. The first is the starting address (i.e., addr = xxxxxxx1), which is received from the operand processing unit. As the starting address is accessing the translation buffer, a constant of 4 is added and the sequence port requests a virtual address (i.e., addr = xxxxxxx5) from the translation buffer at the start of cycle five.

The issue control uses the fork RAM data to determine that the integer unit and two source operands are required. Because only the first operand is missing from the source list, the instruction is issued with bypass. The microsequencer generates the second control store address based on the fast access fields of the first microword.

Cycle Five

Decoding the target instruction stream begins in cycle five. The operand processing unit sends the target address to the branch prediction unit through the program counter's target address. However, as noted earlier, the target address sent is discarded. Because the operand processing unit does not use the I-box data register, the short literal expansion unit can pass the short literal to the source list.

The branch prediction unit now waits either for the E-box to indicate the correctness of the prediction or for subsequent branches to be decoded. The unit predicts a maximum of three branches before it stalls decoding to resolve the first branch.

As the address xxxxxxx5 is accessing the translation buffer, the final address is produced by adding 4, which makes a translation buffer request (i.e., addr = xxxxxxx9) through the sequencer port in cycle six. The three translation buffer accesses are contiguous and interruptible. Data alignment is performed by the M-box, but the alignment is constrained to longwords. When an unaligned quadword is detected, the front end of the M-box alters the context field that it passes to the data cache unit. The quadword request is effectively broken into two unaligned longwords, which are properly rotated into the low longword of the quadword interface and sent to the E-box independently.

Cycle five is the data cache read cycle for the first unaligned longword. Because the starting address is xxxxxxx1, the entire longword is contained in the cache line. Therefore, one additional rotation cycle is all that is required before the data is sent to the E-box. The M-box pipe is effectively lengthened by a cycle when it is performing unaligned operations. Because cycle five is a data cache read cycle, no response is issued to the E-box. In addition to the data cache read, the physical address is placed in the write queue. A memory write is required after the bit is tested. A status bit for a new quadword is set in the write queue. The new quadword indicates that this is the starting address of an operand and writes should not take place until an entry appears in the write queue with a last bit assertion.

Because the first operand is written into the source list, the operand is available to the integer unit at the start of cycle six. The microword hold signal is asserted to hold the first microword during the stall. The microsequencer regenerates the control store address of the second microword.

Cycles Six through Nine

In cycle six, the data cache is read again with address xxxxxxx5, which is the same cache line read in cycle five. However, because the context is a longword, one additional byte of data must be read from the cache to satisfy the request. Also, in cycle six, rotation of the data read in cycle five is completed, and the M-box responds to the E-box. Finally, address xxxxxxx5 is placed in the write queue.

By using source pointers from the source queue, the position and base address operands are selected by the fork RAM and passed to the integer unit. If the base address operand page faults, the base specifier was an indirect specifier. The M-box returns the data to the E-box as faulty, rather than returning the indirect address to the I-box. The return to the E-box results in a memory management page fault. Both operands are saved in registers within the integer unit. Also, the position is divided by eight by shifting, and is saved. The source pointer used to get the base address as source two is incremented and used to select the next source list entry, which is the low longword of the prefetched quadword field. Issue control determines that only the source two operand is missing and issues the second microword with bypass. The microword hold signal is deasserted and the microsequencer generates the control store address of the third microword.

Cycle seven begins with a data cache read of address xxxxxxx9. The rotator is spinning the three bytes <7:5> of interest from the cache read in cycle seven to the correct position. No response is issued to the E-box because this unaligned reference requires two data cache reads to fulfill. The address xxxxxxx9 and the last bit are inserted into the write queue. The M-box delivers the required longword, and execution begins immediately. The second execution cycle calculates the target byte address. The position, divided by eight, is added to the base address. The microsequencer generates the fourth control store address by using the next address field of the microword. No operands are selected for the next cycle, and the next instruction is issued normally.

Cycle eight is a rotation-only cycle. The one byte <8> of interest, read from the cache in the previous cycle, is rotated into the correct position (i.e., byte <0:3>), and the M-box sends the data to the E-box by issuing a response.

The third execution cycle uses the bit position to set up the special encoder in the integer unit and clear the appropriate bit. The source two register file pointer is incremented again to select the high longword from the source list. This microword branches on three conditions determined by hardware functions. The first condition indicates if the low longword of the prefetched field has a page fault. If a fault does exist, the microword flow checks whether the longword is needed or not. As noted earlier, the longword was prefetched in the hope that the bit position was within the first 64 bits of the base. If the bit is not within the first longword, the page fault can be disregarded. The second branch checks whether the position is greater than 63 bits. If it is greater, the microcode must ignore the prefetched quadword and initiate a byte-read directly to the M-box for the appropriate byte. The third branch checks whether the position is greater than 31 bits. This check is used to determine which prefetched longword to use when the position is in the 64-bit range. In this example, the bit position of 13 means that the bit is in the low longword and no page fault is assumed.

Issue control determines that only the source two operand, which is the high longword, is missing. The fourth microword is issued with bypass, and the microsequencer generates the control store address of the fifth microword.

In cycle nine, the M-box delivers the high longword and execution begins immediately. The encoder in the integer unit clears the correct bit in the low longword of the field. The microsequencer generates the sixth control store address, and the next cycle is issued normally.

Cycles Ten through Fifteen

In cycle ten, the E-box initiates a byte write to the M-box. Data is passed to the M-box, and the appropriate byte is shifted to the low byte location. The sixth and final microinstruction is issued normally.

In cycle eleven, the M-box receives an explicit E-box write request to retire the BBSC instruction with a memory write. Explicit writes differ from writes initiated by the I-box in that the E-box supplies a virtual address with the data, whereas the I-box provides a virtual address and the E-box subsequently provides the data for I-box writes. However, three entries exist in the write queue for the prefetched quadword. These entries were placed in the queue for memory conflict-checking purposes and cannot be used for writing purposes because only a byte of data is being written and not a quadword. The write field command from the E-box forces the write queue control to discard the three entries. The front end of the E-box accesses the translation buffer and checks for write success during this cycle. If the write is successful, the physical address and the context of the byte are sent to the data cache.

The final execution cycle determines if the branch prediction was correct. The bit specified by the correct position is shifted to the least significant position in the shifter, where it can be used for a macrobranch comparison. The macrobranch result is compared to the I-box branch prediction in cycle twelve. The microword also indicates that the microsequencer should start forking for new macroinstructions.

Cycle twelve is the data cache lookup cycle for the byte-write operation. The data size is less than a longword. Therefore, the byte that is to be written must be merged with the seven unaffected bytes of the cache line.

Two signals are sent to inform the I-box of the branch prediction status. The branch valid signal indicates that a branch prediction validation has occurred, and the branch signal indicates if the validation was correct.

The branch prediction logic receives the branch valid signal. If the prediction was correct, the program counter's unwind buffer is discarded and the branch logic state returns to idle. If the prediction was incorrect, any data for subsequent instructions is flushed from the pointer, source list, fork, and program counter queues and the program counter is restored from the unwind buffer.

The byte of E-box data is rotated and merged with the cache line that was read during the lookup cycle in cycle thirteen. In cycle fourteen, if the prediction was incorrect, the branch prediction cache is written by using the program counter's unwind buffer as the target address. The prediction also is amended; the branch logic state returns to idle; and the virtual instruction cache is accessed using the program counter's prefetch buffer for the subsequent instruction. The data cache is then written.

Interactions between Instructions

A short cycle time and features, such as a virtual instruction cache, multiple specifier decode, multiple operand and instruction prefetch, queues for decoded instructions, and multiple functional units, combine to produce a system with several variable length overlapping pipelines interconnected with various buffers and queues. There are more than twenty different functions that could be counted as single pipeline stages but many operate in parallel such that the pipeline is considered to vary between eight or nine stages.

We have described each instruction as a single entity moving through the various VAX 9000 pipeline stages. However, many interactions exist between instructions that can decrease the speed of the system. Bypasses are required in several stages in case the previous instruction generates results that the current instruction needs. In some cases, the pipeline must stall as it runs dry, or an upstream stage must wait for a result that is in a stage several cycles downstream. To maximize performance, the I-box decodes up to five instructions ahead of the E-box. This process evens the flow through the pipeline and keeps the E-box busy. Figure 6 illustrates the code block as it moves down the pipe.

The first stage is the virtual instruction cache access, or fetch, stage as the instruction is read from the virtual instruction cache. Some instructions do not need an actual virtual instruction cache access but are in the I-buffer from a previous virtual instruction cache fetch. The instruction decode takes place in the decode, or XBAR, stage. The I-buffer is shifted and the fork RAM is accessed in this stage as well.

The specifier, or operand processing unit, stage has several parallel functional units: the operand processing unit, the short literal expansion unit, and the register/pointer unit. The microaddress generation occurs in this stage. Together with the translation buffer cache lookup, the I-box can pass data to the E-box, and the microword access can occur in the translation buffer stage. The cache stage includes issue and source list access. The execute stage can be executed in either the integer unit, float unit, multiply/divide unit, or all three. The E-box can retire only one issued microinstruction each cycle, but not all issued instructions need to be retired. The final E-box stage is the write general-purpose registers stage, where the registers are updated. However, the M-box can access cache or queues for writes at the same time. The last stage is the cache data write stage.

The ADDL2 instruction flows through nine stages without problems, if there are no previous instructions in the pipeline and all the caches hit. The SUBF3 takes two cycles to execute and ends in the write general-purpose registers stage.

The MULG3 needs four cycles for decoding. The operand processing unit is busy for three cycles. The E-box issues two microinstructions, the second of which requires two execute cycles. The MULG3 includes two retire and write general-purpose registers cycles. The BBSC uses two decode cycles, and the translation buffer is accessed three times for the unaligned quadword. The first four bytes of data from the cache need an extra cycle to pass through the rotator, and the second four bytes need two cache accesses and the rotator cycles. Six microinstructions are issued in the BBSC instruction, and the E-box write needs a translation buffer lookup before the cache lookup can occur. Figure 6 also illustrates the E-box stall that occurs because the two MULG3 retire cycles delay the first BBSC retire cycle and second issue cycle.

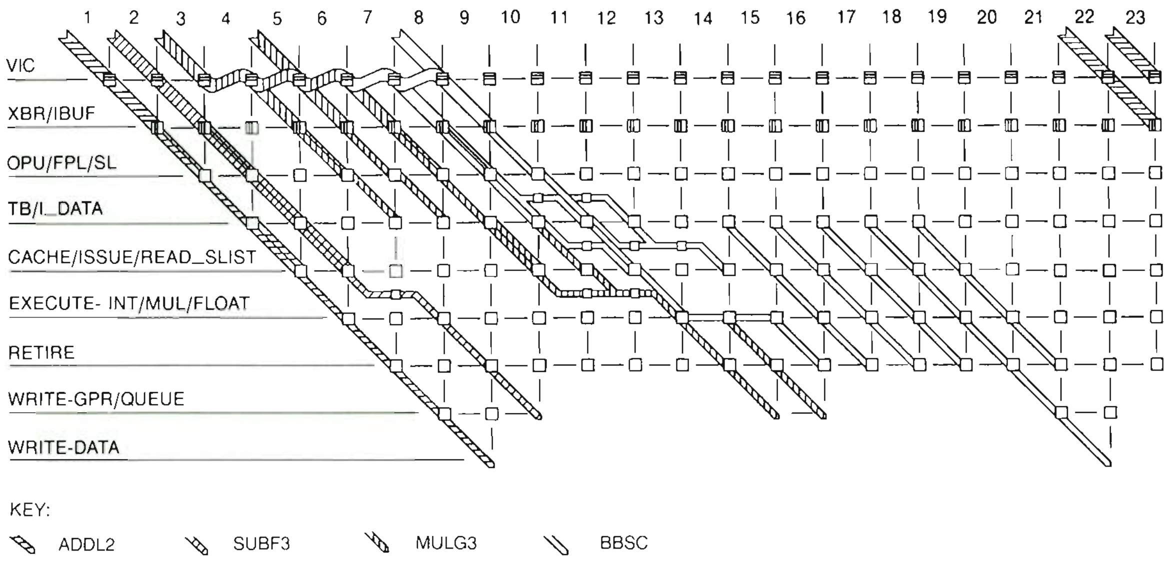


Figure 6 VAX 9000 Instruction Pipeline

Overall some part of this set of instructions is being worked on for 22 cycles. After cycle nine the I-box can be prefetching and decoding the instructions after the branch, either the instructions directly following the branch or instructions that are branched to. The number of retire cycles used or wasted by a sequence of instructions is a good measure of the time taken to execute those instructions. If the prediction is correct, these four instructions execute in 15 cycles; but if the prediction is incorrect, these instructions take 21 cycles.

Conclusion

The various advanced architectural features contributed to the low number of cycles required for an average VAX instruction. The virtual instruction cache provides a high bandwidth of instruction stream to the I-buffer (8 bytes per cycle) and requires a much lower bandwidth from the M-box (8 bytes every 12 cycles). The large I-buffer presents 9 bytes of instruction stream for decoding. The instruction decoder (XBAR) delivers up to three specifiers per cycle. The operand processing unit calculates operand addresses and branch target addresses. The branch prediction unit accurately predicts the majority of branches, and instruction decoding continues down the predicted path so that no time is lost waiting for results from compares. The I-box prefetches and decodes up to five instructions ahead of the E-box. The translation buffer contains up to 1024 virtual-to-physical address translations. However, if the required translation is not contained in the translation buffer, the fix-up unit autonomously creates an entry, which eliminates the usual latency involved when an E-box is used to translate addresses. The M-box also pretranslates write addresses and stores them in the write queue for subsequent access and conflict checking. The 128KB, two-way associative, write-back cache provides a very low miss rate, high bandwidth, and low latency. The queues of decoded instructions allow the I-box pipeline to be less tightly coupled to the E-box. The multiple functional units in the E-box allow multiple VAX instructions to be executed in parallel. The architectural features of the VAX 9000 interact to produce a CPU that executes VAX instructions in the least number of cycles or ticks. The low number of ticks per instruction, combined with the short cycle time, produce the highest performance VAX system now available.

Acknowledgments

The design and implementation of the VAX 9000 was a team effort with contributions from many people, including: Dave Fite, Mark Firstenberg, and Mike McKeon, who were responsible for the I-box; Dave Webb, Maurice Steinman, Joe Macri, Brad Hollister, and Basheer Ahmed, who designed the M-box; Bill Grundmann, Larry Herman, Ginny Lamere, Elaine Fite, Dan Stirling, Eileen Samberg, Mark Haq, and Matt Adiletta, who were members of the E-box design team.

References

1. D. Marshall and J. McElroy, "VAX 9000 Packaging—The Multichip Unit," *Proceedings of COMPCON '90* (San Francisco: Spring 1990): 54–57.
2. T. Fossum and D. Fite, "Designing a VAX for High Performance," *Proceedings of COMPCON '90* (San Francisco: Spring 1990): 36–43.
3. T. Leonard, *VAX Architecture Reference Manual* (Bedford: Digital Press, Digital Equipment Corporation, 1987).
4. J. Murray et al., "Microarchitecture of the VAX 9000," *Proceedings of COMPCON '90* (San Francisco: Spring 1990): 44–53.
5. M. Troiani et al., "The VAX 8600 I-box, A Pipelined Implementation of the VAX Architecture," *Digital Technical Journal,* vol. 1, no. 1 (August 1985): 24–42.
6. J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proceedings of the 11th Annual Symposium on Computer Architecture* (Ann Arbor: June 1984).
7. T. Fossum, J. McElroy, and W. English, "An Overview of the VAX 8600 System," *Digital Technical Journal,* vol. 1, no. 1 (August 1985): 8–23.
8. S. Mishra, "The VAX 8800 Microarchitecture," *Digital Technical Journal,* vol. 1, no. 4 (February 1987): 20–33.

Matthew J. Adiletta
Richard L. Doucette
John H. Hackenberg
Dale H. Leuthold
Dennis M. Litwinetz

Semiconductor Technology in a High-performance VAX System

The VAX 9000 system is the newest member of Digital's VAX family of computer systems. The 9000 is a high-performance ECL processor, with a very fast, 16-nanosecond cycle time. To achieve this high level of performance, a new generation of semicustom and custom integrated circuits was required for the scalar CPU and the vector processing option. Goals for circuit density, performance, and skew maintenance were fulfilled with the development of a high-speed gate array, special custom chips used in key applications, and a high-speed RAM employing a new architecture.

The semiconductor requirements for the VAX 9000 system posed a number of challenges for Digital's Integrated Circuits Development Group. Those requirements included a tremendous number of equivalent logic gates (1,037,400 gates) and a large amount of RAM in the processor (3,280,000 bits). Moreover, the project's performance goal of over 30 VAX-11/780 units of performance (VUPs) required the development of state-of-the-art semiconductors and the use of innovative techniques to design them.

Given the project's goals, the IC technologists evaluated several competing semiconductor technologies and decided to implement most of the logic within the 9000 system in a high-speed, high-density, 10,000-gate array. The gate array provides a broad range of speed and power-dissipation options. Working with Motorola, the IC Group first engineered the base 10,000-gate macrocell array (MCA), which is implemented in Motorola's MOSAIC III process. Logic engineers then designed the 77 different gate array chips (options) on the base array, using a rich library of logic functions and a set of automated place and route tools. Additionally, they designed five custom chips, invented a fast cycle time, self-timed random access memory (STRAM) architecture, and designed a multichip unit to interconnect all these high-performance ICs.[1]

Four different design methods were used to implement the chips. The MCAx chips employ a gate array design technique. The CDxx, the VRGx, and the STRAM chips required a full custom approach. The STGx chip was implemented using a silicon compiler technique. The MULx and DIVx chips mwere implemented using a standard cell design approach. Statistics on 9000 system chip design are given in Table 1.

This paper describes the VAX 9000 MCA III gate array, the development of each of the five custom chips, and the STRAM architecture. Before our discussion of the gate array, we present a brief overview of the semiconductor technology used to fabricate the array and the custom chips.

Semiconductor Technology

In 1985, the VAX 8800 series was Digital's largest and most powerful system, offering single-CPU performance of eight VUPs. The 8800 CPU logic was Motorola's Macrocell Array I (MCA I) gate array, which was fabricated in MOSAIC I bipolar technology. In comparison, the VAX 9000 goal of 30 VUPs was aggressive, and the IC Group realized a new semiconductor technology was required.

At the start of the project, the technologists evaluated semiconductor vendors to determine what was the "best" technology available to implement the new system. CMOS, BiCMOS, bipolar, and GaAs IC technologies were evaluated. Among the factors considered were logic density, gate delays, on- and off-chip interconnect delays, manufacturing risks, and product delivery.

Although very high gate densities were available with CMOS technology, the logic gate delays proved

Table 1 VAX 9000 Chip Statistics

Chip	Description	Die Size (Millimeters)	Signal Pins	Transistor Count	RAM Bits	Power (Watts)
MCAx	MCA III gate array chip	9.8 × 9.8	256	40.1K	–	30
CDxx	Clock distribution chip	6.2 × 6.2	170	7.2K	–	13.9
STGx	Self-timed register file chip	9.8 × 9.8	152	29.3K	–	17.8
MULx	Multiplication chip	9.8 × 9.8	182	48.4K	–	30.9
DIVx	Division chip	9.8 × 9.8	112	29.2K	–	23.9
VRGx	Vector register file chip	9.8 × 9.8	198	76.0K	9216	24.9
1KSR	1K × 4 self-timed RAM	4.9 × 3.6	33	28.0K	4096	2.4
4KSR	4K × 4 self-timed RAM	6.4 × 4.2	35	103.0K	16384	2.4

to be too slow to meet the cycle time requirement. Also, the CMOS output circuits could not drive signals off-chip into a 50-ohm transmission line as quickly as a bipolar transistor, which limited the speed of signal between ICs.

BiCMOS offers the advantage of highly dense CMOS coupled with bipolar drive capability. However, the technologies available at the time were optimized for the best CMOS transistors with a compromised bipolar device. This approach limited the overall performance of the circuit to a level roughly equivalent to that of previous generation bipolar devices, which would not be aggressive enough to meet the CPU performance needs.

Gallium arsenide (GaAs) ICs offer a theoretical performance advantage of between two and three to one over silicon implementations. The group found IC densities were lower than those of bipolar devices, however; and the on-chip speed advantage was countered by the need for more off-chip signals in the critical paths of the CPU. Also, because the manufacturing technology of GaAs ICs was immature, very few companies had attempted to sell GaAs into the commercial marketplace. So while this technology was considered for a time in some applications where alternatives also existed, GaAs were eventually dropped from consideration because of the uncertainty of availability.

The IC Group also studied Motorola's third generation of their oxide-isolated self-aligned implanted circuits (MOSAIC III) bipolar technology.[2] It offered a factor of six in speed advantage over the previously used MOSAIC I technology and had the potential of providing eight to ten times the logic density. Although not as dense as CMOS or BiCMOS, MOSAIC III was much faster than either of those technologies and much denser than any available GaAs technology. In addition, although many of the manufacturing steps were new, most of them were based on previously proven techniques. The group therefore concluded that MOSAIC III was best suited to meet the challenges of the VAX 9000 system.

The MOSAIC III process is an advanced silicon bipolar process which yields a transistor structure with a polysilicon base, emitter and collector electrodes, polysilicon resistors, and three layers of metalization. Compared to the MOSAIC I device used in the 8800, the critical collector-base junction of this transistor structure takes up approximately 50 percent less area, as shown in Figure 1. Combined with shallower junctions and reduced base resistance, the intrinsic device performance was improved by a factor of three. Further, the polysilicon resistor produced with this process has far lower parasitic capacitance than the MOSAIC I monosilicon resistor. Some key performance modeling parameters and density metrics are provided with the figure.

The VAX 9000 packaging imposed other requirements on the semiconductor technology. Power dissipation increased from 5 watts for the MCA I to 30 watts for the MCA III because of the increase in gate density from 1,200 to 10,000 gates. Therefore it was determined that all chips should be mounted directly to the multichip unit cold plate for optimum cooling. For manufacturing economy, it was desirable to bond the multiple leads of the chip directly to the pads on the high-density signal carrier (HDSC). Consequently, all CPU chips must be provided to the multichip unit assembly site in a tape automated bond (TAB) package. As shown in Figure 2, chips are mounted in a plastic carrier suitable for automated handling, and the surface of the die is protected from mechanical damage with an epoxy encapsulent.

MCA 10K Gate Array

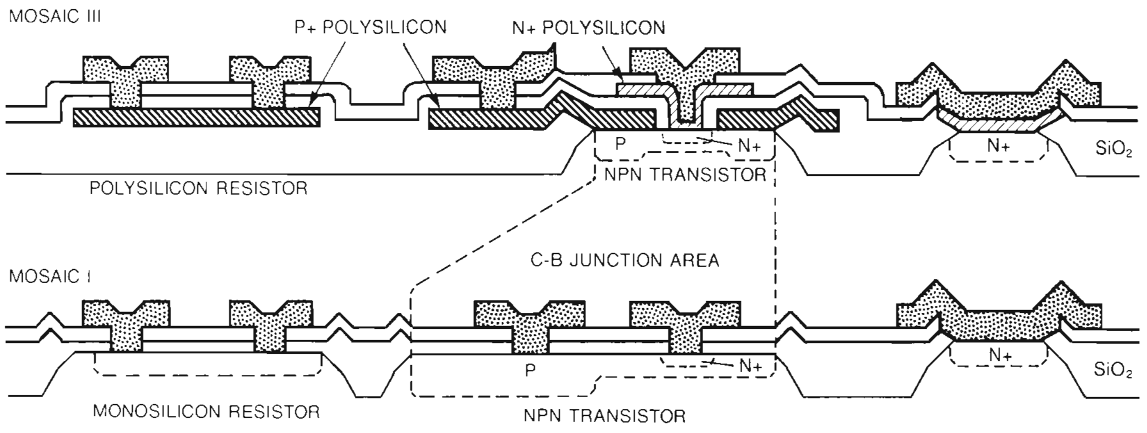
A high-performance emitter coupled logic (ECL) gate array with 10,000 equivalent gates and 256 inputs/outputs has been developed for the VAX 9000 system. The gate array design approach used in the VAX 9000 system ensures the shortest possible turnaround time from option mask to hardware, thereby reducing the system design time. In this approach, cell boundaries are defined with all transistors and resistors fixed within the cells. When a cell function is selected from a predefined cell library, the cell customization occurs at the metal between the transistors and resistors. Then, to define the function of the gate array option, the metalization between cells is customized. This approach allows the semiconductor foundry to build many wafers up to the customization level; when a gate array is to be built, only the custom metal is required. As noted above, 77 different 10K ECL gate array options are used in the VAX 9000 system. This gate array has a rich selection of logic cells with different power settings for the logicians to use to meet performance and power requirements.

Using Rent's Rule, technologists maintained a balance between the number of gates and the package I/O count. This balance ensures that a maximum number of logic cells for a given signal pin count are available for the logic designers. Technologists evaluated several key factors to determine the gate array physical layout and to ensure its success:

- Area of the silicon chip versus yield
- I/O pad pitch
- Maximum power dissipation
- Speed of the gates
- Maximum number of logic cells

Successful trial layouts of the 10K ECL gate array floor plan were completed before any VAX 9000 options were started.

The gate array floor plan, shown in Figure 3, comprises a central core area of 414 major (M) cells, divisible into quarter cell functions, arranged in an array of 20 rows and 21 columns, less 6 sites for the master bias generators and special clock generator circuits. The number of transistors used in a quarter cell is based on the logic cell most frequently used in the 10K ECL gate array, the scan latch. A ring of 200 output (O) cells is interspersed with 224 interface (I) cells. The ring surrounds the internal cells and interfaces the pad drivers with the internal



	MOSAIC I	MOSAIC III
NPN F_T:	5 GHz	16 GHz
R_b:	1475 ohms	400 ohms
C_{JC}:	50 ff	20 ff
C_{JE}:	45 ff	24 ff
C_{JS}:	185 ff	54 ff
DRAWN EMITTER SIZE:	3μm × 4μm	1.75μm × 4μm
METAL 1 PITCH:	8μm	4.5μm
METAL 2 PITCH:	15μm	7μm
METAL 3 PITCH:	—	12μm

Figure 1 Comparison of MOSAIC III and MOSAIC I Devices

cells. The 256 I/O pad cells along with the 104 power pads are located around the perimeter of the 10K gate array. The metalization system uses three interconnect layers. The customized routing channels reside on the first and second metal layers with interconnecting vias between the two layers of metal. The top metal layer and parts of metal 1 and 2 provide power and ground distribution.

The 10K ECL gate array used in the VAX 9000 is approximately ten times more dense than the ECL gate array used in the VAX 8800 system. The gate delays in the 9000 are improved six times over gate delays in the VAX 8800. Table 2 compares the 10K ECL gate array used in the VAX 9000 to the ECL gate array used in the VAX 8800.

Previous gate array designs, in general, have provided only two levels of series gating, thereby limiting the complexity of functions that can be designed with one current switch. Within this gate array, three levels of series gating at both internal and output macrocells provide additional "AND" (product) gate functions at very high speed with one switch delay and at a lower power level. Figure 4 compares three-level series gating and two-level series gating for a "2-3-4-4 AND/OR" logic function (internal gate). Table 3 lists the differences in typical gate performance for a low power gate. The table also compares low power gate and high power gate. Notice the power difference between the two-level and three-level high power gate.

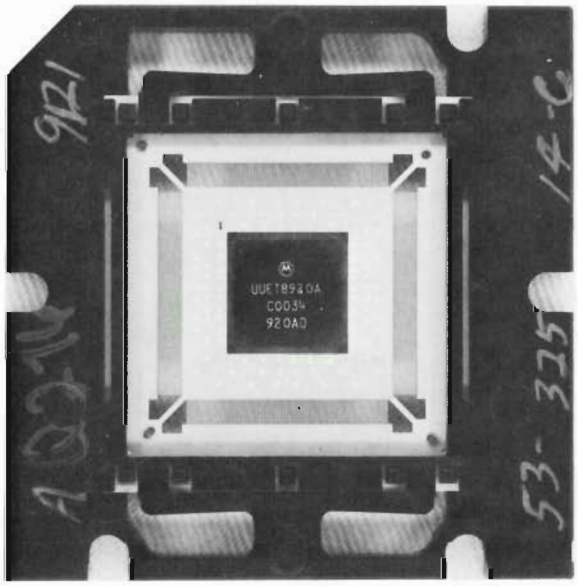


Figure 2 Chip in TAB Package Mounted on Plastic Carrier and Encapsulated

Table 2 Comparison of Number of Cells and Delays in the VAX 8800 and VAX 9000 Gate Arrays

	VAX 8800 Gate Array	VAX 9000 Gate Array
Internal major cells	48	414
Output cells	26	200
Input cells	25	224
Input cells gate delay	1.05 nanoseconds	175 picoseconds (high power)
Metal delay (fall delays)	2.6 picoseconds per mil	1.3 picoseconds per mil

All current switches within the array are powered from the main supply voltage VEE1. Three-level-series gated functions are implemented in the VAX 9000 gate array option, which requires VEE1 to be set to −5.2 V. Input cells are powered from a second, lower supply voltage VEE2 (−3.4 V) to save power. The output emitter followers of M, I, and O cells as well as series-terminated ECL (STECL) output followers employ constant current source pulldowns to VEE2 to save power. The constant current source pulldowns minimize the sensitivity of AC performance to variations in power supply. This same termination scheme was used in VAX 9000 custom chips.

One of the technologists' main goals was to minimize power consumption of each macrocell while obtaining the highest possible performance from the 10K ECL gate array. The overall 10K ECL Gate Array power is limited to 30 watts because of the cooling requirements, the internal power distribution, and the current density limits on power pins.

A unique feature included in the 10K ECL gate array that previous gate arrays do not have is series-terminated ECL (STECL) outputs. STECL outputs

Table 3 Comparison of Two-level and Three-level Series Gating

	Two Levels of Gating	Three Levels of Gating
Gate delay from input pin A to output pin YA (low power)	300 picoseconds	250 picoseconds
Low power gate	9.88 milliwatts	8.84 milliwatts
High power gate	18.20 milliwatts	13.00 milliwatts

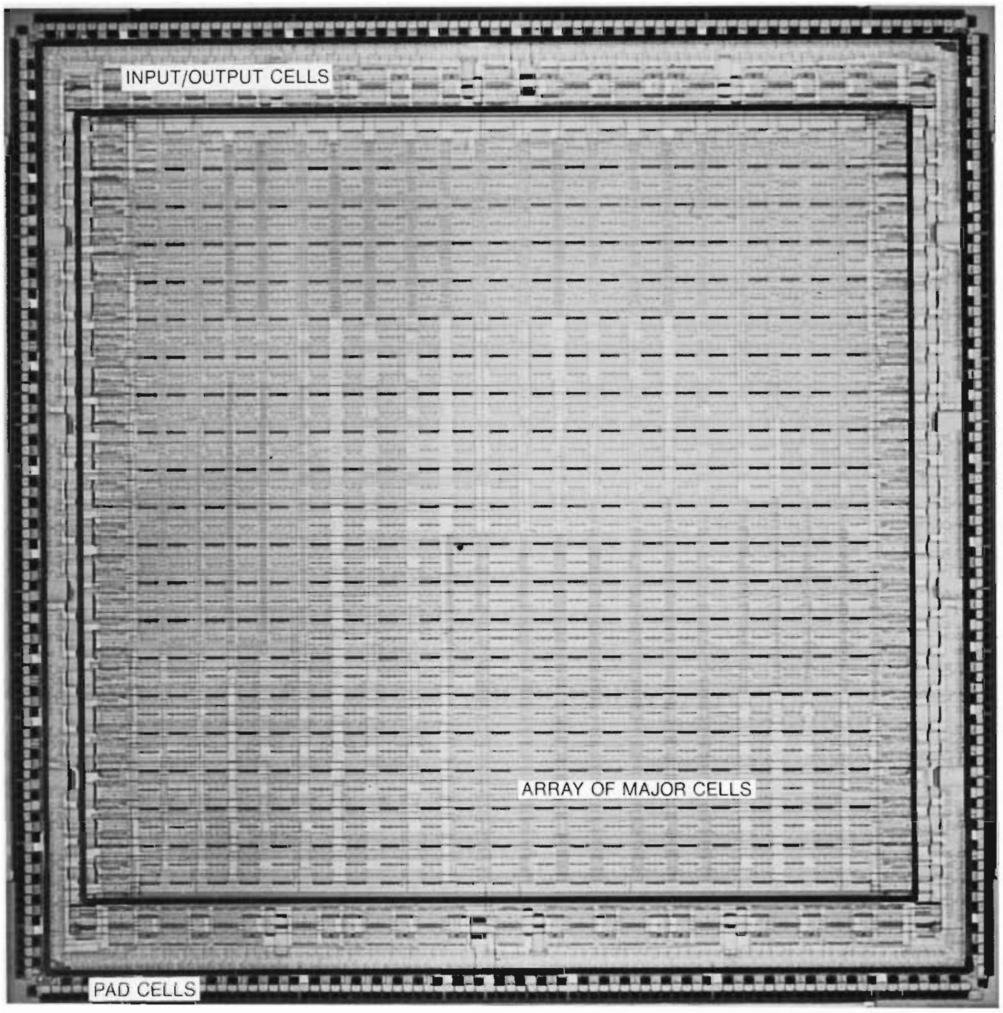


Figure 3 Photomicrograph of the Gate Array

include a constant current source pulldown and a series terminating resistor. This feature allows the elimination of off-chip termination resistors used in conventional 50-ohm ECL outputs. STECL outputs allow shorter interconnections between chips on the multichip unit because the chips can be placed closer to each other, thus improving performance. Another advantage of using STECL outputs over 50-ohm outputs is that less than half of the simultaneous switching output noise is coupled to unswitched outputs. All custom chips used in the VAX 9000 employ STECL termination.

Clock Distribution Chip—CDxx

The major function of the clock distribution chip (CDxx), shown in Figure 5, is to distribute master and reference clocks to each MCA on a multichip unit. There are eight pairs of differential master and reference clocks. The chip also supplies clocks to all STRAMs on the unit. Each of the STRAM's four groups of six clocks can be programmed to one of eight possible clock phases. This flexibility in programming allows the system designer to select the appropriate clocks for STRAMs in order to meet system timing requirements.

In addition to providing the functions above, the design goals for the CDxx project included the following:

- Minimize the space occupied by the chip on the multichip unit
- Provide scan control and scan distribution
- Include a wideband amplifier
- Ensure low clock skew
- Provide a temperature-detecting circuit

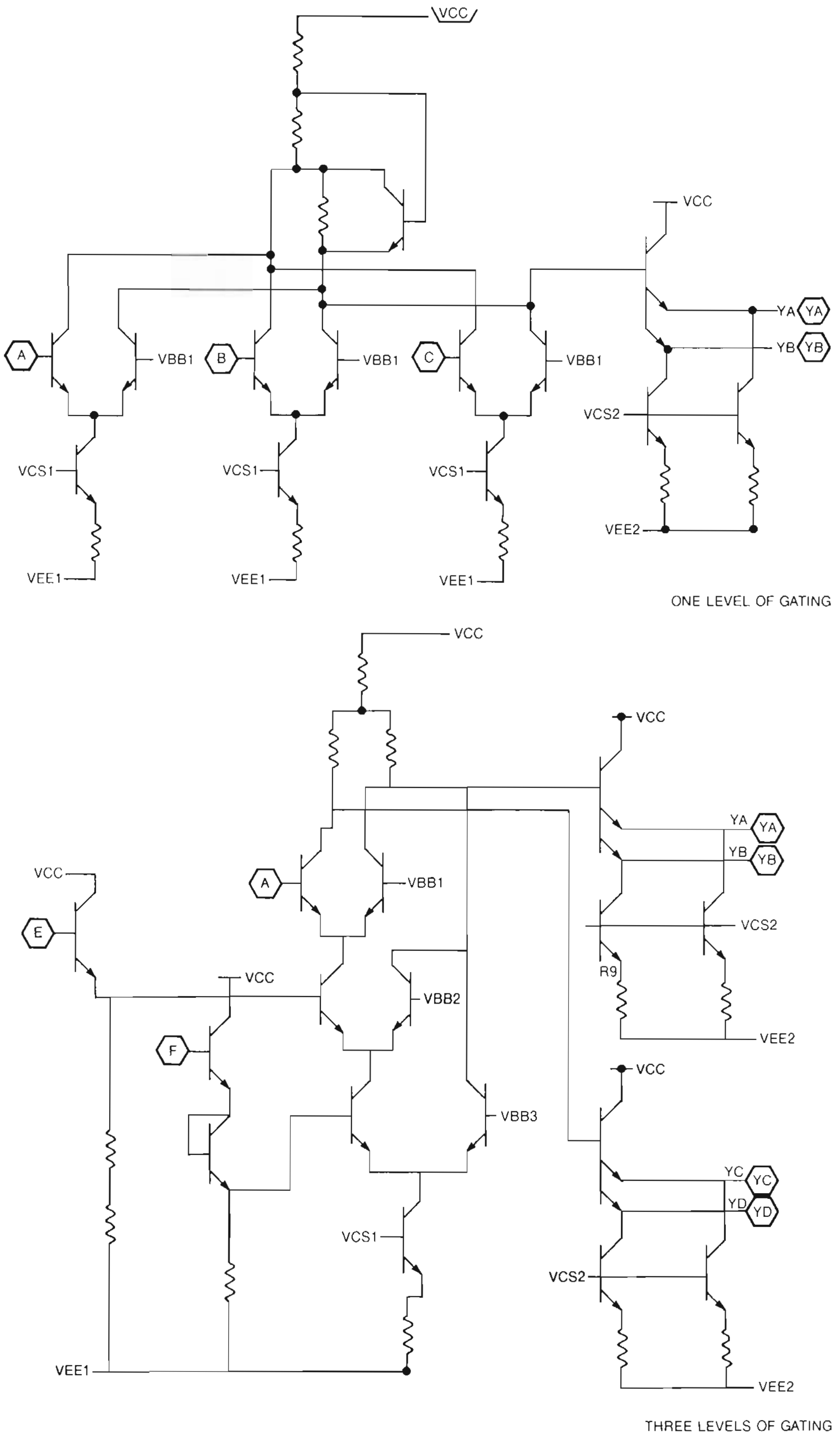


Figure 4 Two-level Functions versus Three-level Functions

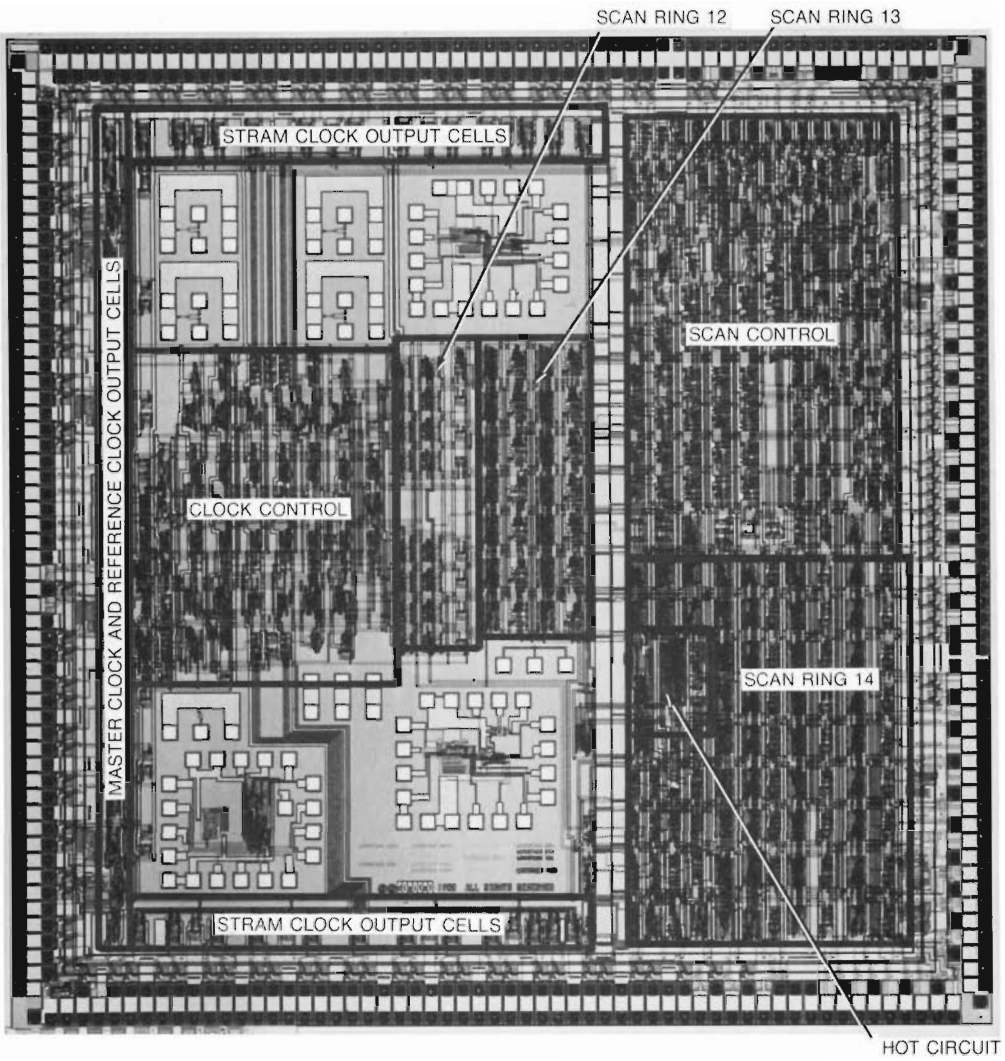


Figure 5   Photomicrograph of CDxx Chip

Minimizing the real estate occupied by the chip was complicated by additional functions located on the CDxx, such as scan and the temperature detecting circuits. The minimization was accomplished by employing a custom chip design approach in which each element (cell) is optimized and then manually placed and routed to achieve a compact design. As it turned out, the size of the chip was not determined by the amount of real estate needed to implement the circuits, but rather by the number of pins required to communicate to the rest of the multichip unit.

Since a CDxx is mounted on every multichip unit in the CPU, the scan distribution and control logic are located on this chip. The CDxx chips in the system are chained together on the system scan bus. Each CDxx receives its scan control signals from the previous CDxx in the chain or from the service processor. As shown in Figure 5, there are three scan rings located on the CDxx. Ring 12 is a 16-bit ring reserved for the CDxx STRAM clock generation control ring. This ring controls the STRAM clock phase selection and enable for each of the four STRAM clock groups. Ring 13 is a 14-bit ring reserved for the CDxx scan control. Data is shifted into this ring and then loaded into CDxx control registers. Ring 14 is a 47-bit ring reserved for the CDxx information scan ring. Data is loaded into this ring from CDxx data registers and shifted out to the service processor.

The design of the wideband amplifier was prompted by the need for the clock distribution chip to receive two differential sinusoidal master and reference clock signals as inputs. These signals are transformer coupled from the clock source. The master clock runs at one eighth the system cycle time, and the reference clock runs at the system cycle time. The wideband amplifier receives differential sinusoidal signals of relatively small amplitude — less than 125 millivolts peak to peak — and transforms them to 100K ECL levels on output. The design of the input circuits meets these criteria and typically functions with inputs less than 65 millivolts.

All the clocks are distributed by the CDxx as pairs of differential signals. The distribution of these clocks is, of course, to be done with minimal clock skew. Clock skew is the difference in delay time between different clock outputs measured from a common point. The common point in this case is the number of master clock inputs to the chip. To maintain low clock skew, technologists designed fast gates and minimized the number of cascaded gates in the clock path. Also, all the metal that interconnects the cells in the clock path is controlled for equal delay. As a result, the measured clock skew is less than 100 picoseconds on a chip for master, reference, and STRAM clocks. The delay of master clock input to output is less than 1 nanosecond (ns).

The temperature-detecting circuit on the CDxx warns the system when a device junction temperature approaches the maximum allowed temperature on a multichip unit. As implemented, the circuit is controlled from the system console. The console loads the CDxx with a number that represents the temperature the circuit must use as a point of comparison. If the junction temperature of the CDxx is higher than the programmed value, the circuit trips and notifies the console of a temperature problem. The console then takes corrective action.

Self-timed Register File Chip — STGx

The self-timed register file chip (STGx) is employed in the VAX 9000 to provide four register banks accessible through multiple read and write ports. The four banks include a microcode scratch-pad register bank, the VAX general-purpose register set, a memory data register storage bank, and an instruction data register bank. The performance requirements for the STGx were quite rigid and guided several key design decisions, including density and layout. The read access time was to be less than 5 ns. The write access time was to be less than 6 ns. In other words, the chip must read or write any one of its 64 locations in 5 or 6 ns, respectively. Both goals have been met. In fact, the read access time is typically less than 4 ns, and the write time is typically less than 5 ns. Figure 6 is a photomicrograph of the STGx chip.

The STGx is a 64-word by 18-bit ECL register file containing three write ports and two read ports. The 64 words are separated into four 16-word by 18-bit storage array sections. Each of the four storage banks has dual read capability. Storage bank one has dual write capability; storage banks two and three have triple write capability; and storage bank four has single write capability. Simultaneous write access to the array is possible through all ports with correct results occurring; the only exception is in the case of writes to the same location from multiple ports, which is an undefined operation. A write followed by a read access to the array — even to the same address — is possible with correct results occurring. The chip has two clock inputs for controlling reads and writes.

One requirement for the design was to include a self-timed write capability so that the system need not provide properly timed write pulses to the chip. In the system, the chip is clocked with STRAM clocks for reading and writing. The design uses these clocks to latch read address information, to latch write address information, and to latch input data. In addition, the design takes the leading edge of the write clock to generate a delayed write pulse. The delayed write pulse is used to write the appropriate word in the 64-word by 18-bit array, taking into account the time needed to decode the write address.

The design style used to implement the self-timed register file chip is similar to a silicon compiler technique. The chip's storage area is made up of four arrays. The input address register for both read and write ports, the input data latches, and the data output drivers are arrangements of cells in strips. The placement and routing of these arrays and strips was procedurally performed using custom layout tools. Once the blocks were assembled and placed, interconnections among blocks, strips, and pins were then routed manually.

Multiplication Chip — MULx

The architecture of the scalar processor defined an integrated floating point processor. Unlike most RISC processors, which off-load all floating point operations to a separate floating point processor, the VAX 9000 system handles floating point operations within the E-box.[3] The multiplication unit therefore supports both integer and floating point formats. To achieve this support, a custom chip was

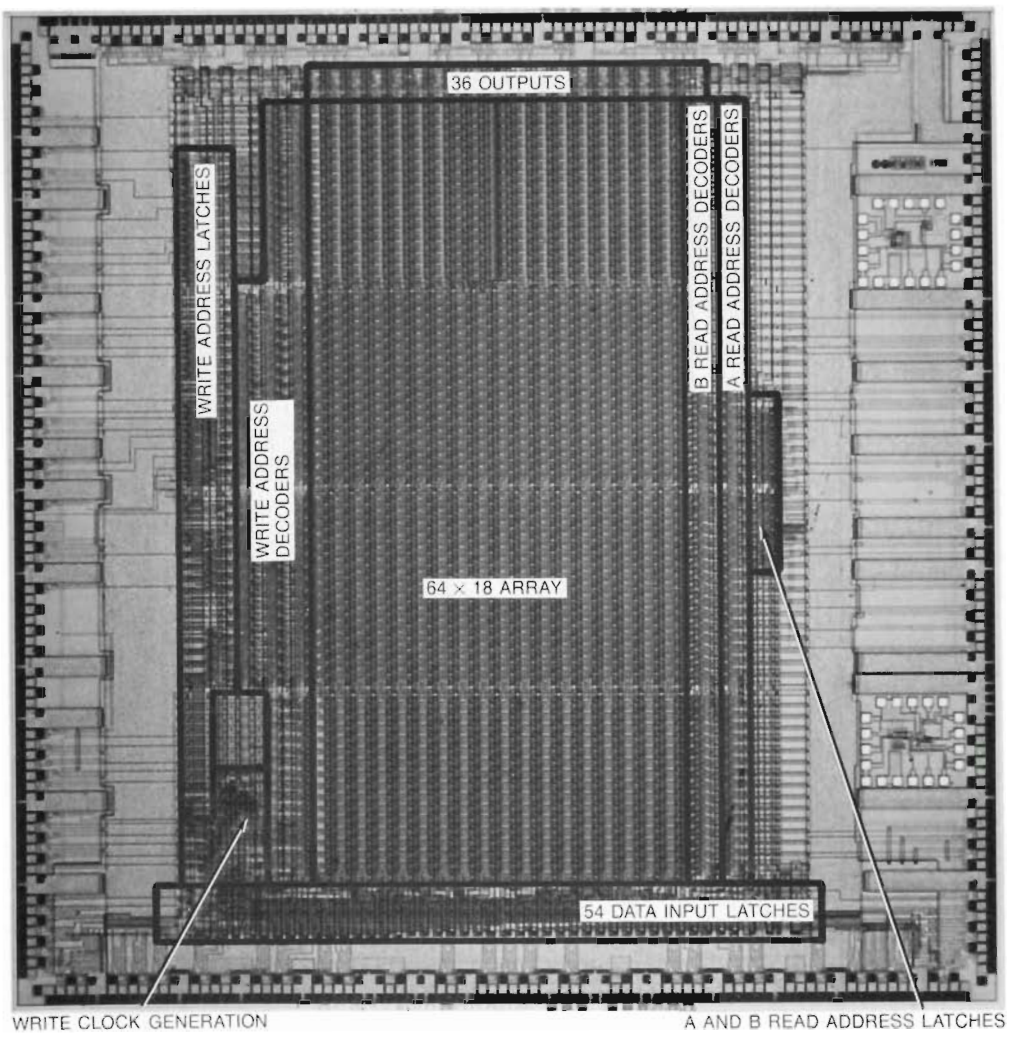


Figure 6 Photomicrograph of STGx Chip

required that provided superior performance, special logic gates, and improved density. Custom chip technology provided enough density to accommodate a 32-bit by 32-bit, eight-logic-level multiplication array in a single chip (MULx). To minimize the cost and time of custom design, designers employed standard cell design techniques in which the cell height was fixed and the width could vary to take advantage of packing density. By constraining the design in this fashion, the High Performance Systems Group's CAD suite could be employed to place and route the chip. Special logic gates eliminated three logic levels, and high-powered fast gates provided the performance to permit a 32-bit by 32-bit multiply operation in less than 9 ns. Figure 7 shows a photomicrograph of the MULx chip.

Three MULx chips were required in the scalar processor to achieve double-precision performance in which every 64 ns a 56-bit multiplication could complete. Each MULx chip has two 32-bit input data buses. The MULx chip is also employed to perform all integer multiply operations in a single 16-ns cycle.

The scalar processor, which has 32-bit-wide data paths, delivers double-precision input data in two cycles. In the first cycle, each MULx consumes the most significant high bits of each operand. All three MULx chips latch this data while also unpacking it, multiplying it, and then latching the product. One of the MULx chips' results is then saved. In the second cycle, the remaining double-precision data, the least significant low bits, is consumed, and each

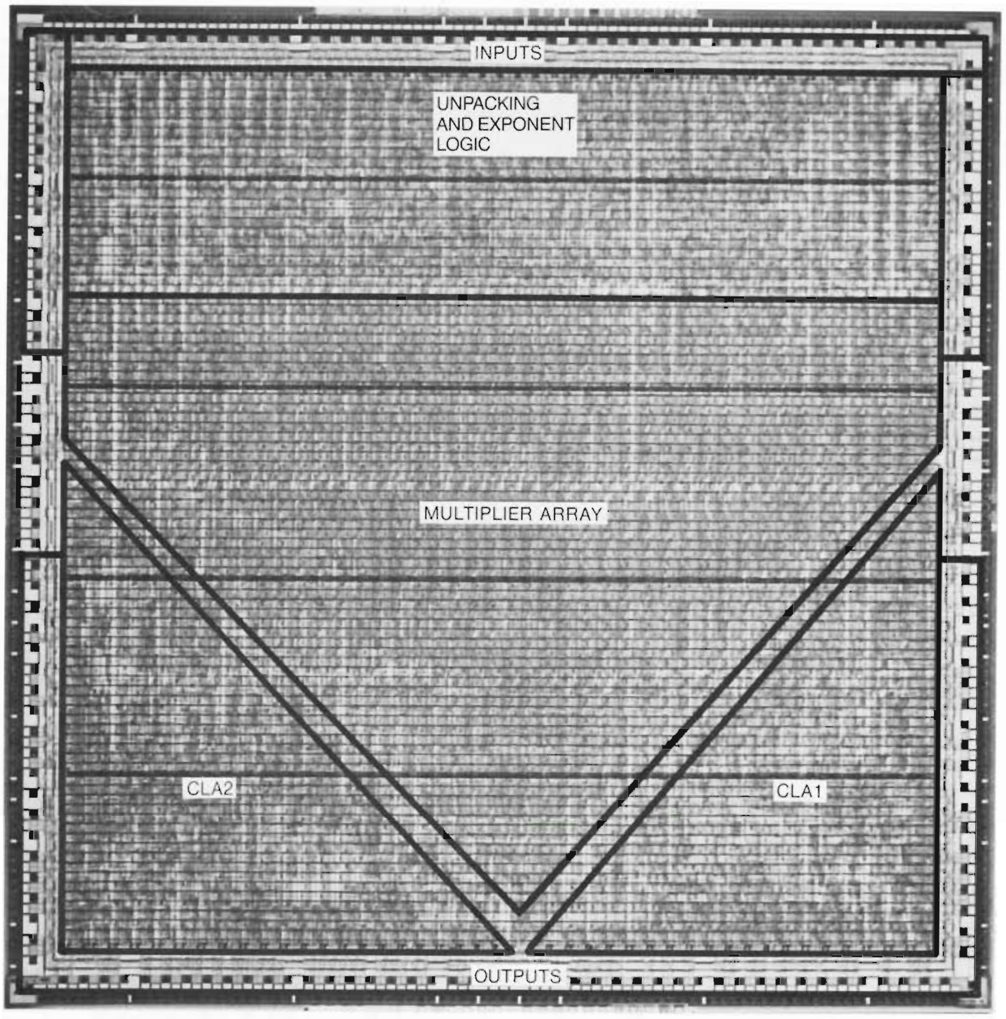


Figure 7 Photomicrograph of MULx Chip

MULx chip unpacks the data and performs a unique multiply: operand A high bits and operand B low bits; operand A low bits and operand B high bits; and operand A low bits and operand B low bits.

An MCA III gate array accumulates all these results, and another rounds and packs the bits into a VAX floating point product. Since each MULx needs to know which partial product it must compute in the second cycle, two personality bits are included that are loaded by means of the system scan chain.

MULx chips are also used in the vector processor. The vector processor (V-box) has 64-bit-wide data paths. Four MULx chips are employed to complete a double-precision multiply every 16 ns. Since the operand unpacking differs between the scalar and vector processors as a result of how fast operands are delivered, each MULx has an additional personality bit for indicating whether the MULx is in the V-box or E-box.

The MULx chip, as used in both the scalar and vector processors, is a 32-bit by 32-bit ECL parallel multiplier which is fully pipelined for a 16-ns cycle time. It performs both two's complement and sign/magnitude multiplication. In a single cycle, the chip unpacks VAX floating point formats F, D, and G, or integer formats long, word, and byte; performs exponent calculations and sign handling; and completes up to a 32-bit by 32-bit multiplication.

If the operation is double precision, the 64-bit result is a partial result. It must be accumulated with three other partial results to form the double-precision, correctly rounded, and normalized product.

If the operation is an integer type, then the 64-bit two's complement result is the VAX integer product. Along with producing this integer product, MULx also produces the correct condition codes. Integer operations require one machine cycle to complete. Operands are not latched at input. Instead they are immediately unpacked and sent to the multiplication array. This multipurpose array then produces a set of sum and carry product vectors. These vectors are then added in a full carry lookahead adder (CLA). This adder comprises a 31-bit adder and a 32-bit adder, cascaded. The produced sum is the 64-bit product, which is then latched. The output of the latch is used to compute integer-type condition codes.

The integer instructions supported include VAX MULB, MULW, and MULL. EMUL is also directly supported, along with the Z and N bit condition codes. Finally, to assist in H format-type multiplications, a true 32-bit by 32-bit magnitude multiplication is also supported, called EXTMUL (extended multiply). There is a 64-bit data path back into the E-box for EMUL- and EXTMUL-type operations.

Six features of the MULx design that improve performance and minimize logic should be noted. First, unlike traditional designs, the MULx design does not include Booth recoding of the multiplier operand. Booth recoding offers no logic savings either in timing or real estate when the multiplication array reduction scheme is optimal. Second, a Baugh-Wooley two's complement algorithm was used to implement integer multiplication.[4] Third, engineers designed special full adder logic gates to integrate multiplication summand generation into the full adder cell and to eliminate the need for an additional logic level. Fourth, a unique multiplication reduction algorithm was developed which provides the initial routing advantages of a Wallace tree, with the minimal logic of a Dadda tree.[5,6] Fifth, a ripple is formed in the reduction array. The ripple facilitates the start of the least significant 31-bit CLA addition at least one logic level sooner than the most significant 32 bits and does not require a carry-in input to the upper 32-bit adder. Finally, by developing a very fast 4-3-2-1 AND/OR gate, engineers were able to remove two additional logic levels in both CLA adder networks.

To avoid bugs in the array design, since bugs in an array consisting of 1000 full adders could have significantly affected the product shipment schedule, engineers developed a FORTRAN program to logically interconnect and physically place the array.

Any bugs would be algorithmic and not random, and algorithmic bugs should be obvious. In addition, by algorithmically placing the array, significant density improvements were realized. This program provides a Wallace-Dadda implementation that logically reduces 32 rows in 8 logic levels, and consumes as many initial summand bits. It also uses the least number of full adders as theoretically possible, while delivering the least significant 32 bits of sum and carries at least one full logic level sooner than the most significant bits.

Division Chip—DIVx

The iterative divide function performed by the division chip, DIVx, requires a significant amount of hardware, the density of which a standard cell chip affords. Two gate arrays would be required to perform the same function, in which case a timing-critical path crossing would occur between the two chips. Therefore, the IC designers implemented the DIVx chip as a standard cell design by building on the techniques developed for the MULx chip described above. Also, like the MULx design, the goals for the DIVx design project were to optimize performance and minimize real estate use by fitting the iterative divide function in a single chip.

The IC designers employed a standard cell technique in which four horizontal sections are defined, each section having a different number of columns. Reference cells are located in the center row of each section and provide ECL reference voltages to the cells above and below in that section's columns. Placement was driven for performance, with quotient selection logic being distributed to where it was required. This method made for an irregular structure, as can been seen in Figure 8.

The VAX 9000 system optimizes both multiplication and division by providing separate functional units. Each functional unit performs both integer and floating point operations. This approach differs from the one taken by most processor architects, who conceptually link multiplication and division. Usually, algorithms are chosen that can share hardware at the expense of the performance of either operation. The separate division unit in the 9000 provides superior performance for both integer and floating point operations. The DIVx chip is also used by the V-box to perform very fast vector division operations, as shown in Table 4.

Division is an iterative process. Unlike the case of multiplication, one cannot predict the summands and then reduce the summand matrix. The two approaches to division most commonly used are the Taylor Series convergence algorithm and a subtract and shift algorithm.[7] The algorithm employed in the 9000 is a variation on the subtract and shift

Table 4 Division Performance

Data Type		Cycles	Time (Nanoseconds)
Integer:	byte	3–4	48–64
	word	3–5	48–80
	long	3–8	48–128
Floating point:	F-format	7	112
	D-format	13	208
	G-format	12	192

method, which allows for savings in hardware as well as increased performance.

In this method, an imprecise quotient is selected based on a truncated estimated partial remainder and a truncated version of the exact divisor. This imprecise quotient digit is corrected when the next guess quotient digit is selected. The selected digits may be positive or negative. The positive digits are accumulated in a positive-value shift register. The negative digits are accumulated in a negative-value shift register. The final corrected binary quotient is then formed by subtracting the negative register from the positive register.

The algorithm is based on a signed digit notation scheme. To determine two quotient bits, the bits may be chosen from a digit set that includes $\{-2, -1, -0, +0, +1, +2\}$. The digit set is simply an expanded form of the common nonrestoring digit set that typically uses $\{-1, 0, +1\}$. In nonrestoring algorithms, the quotient is normally corrected as

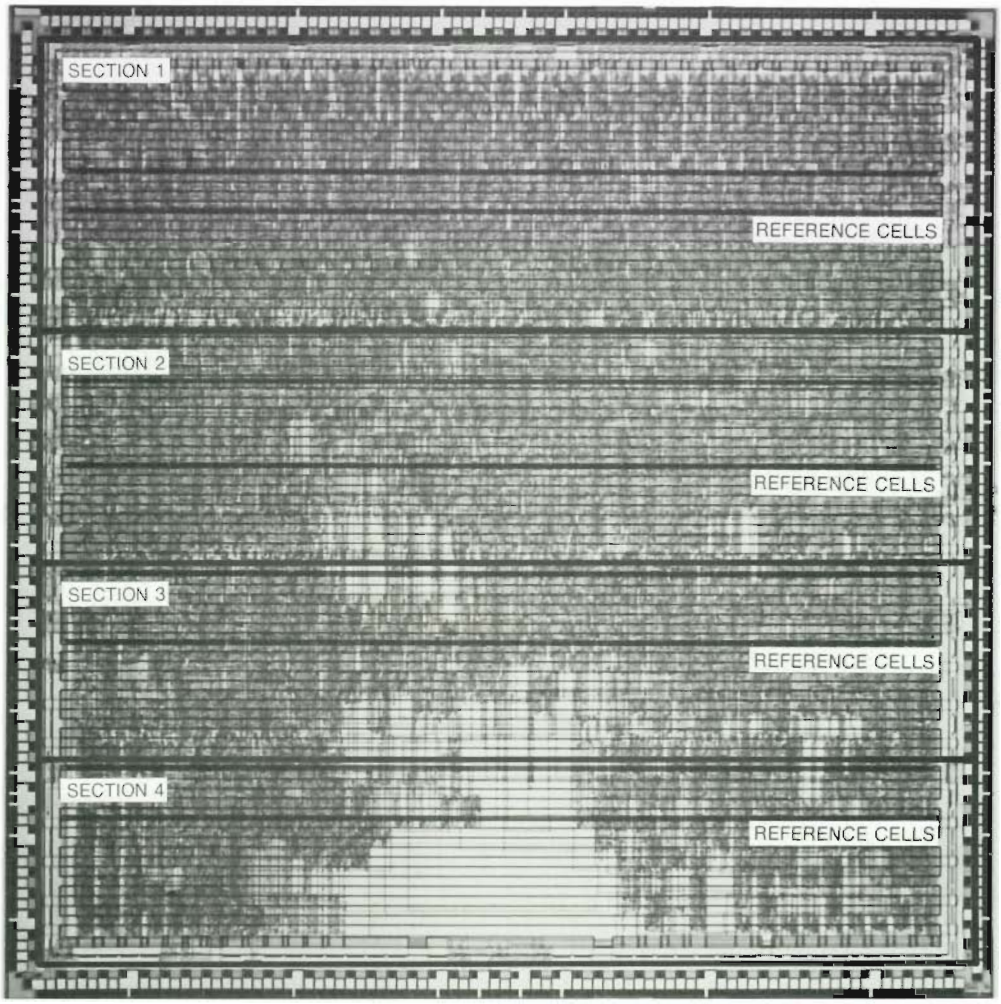


Figure 8 Photomicrograph of DIVx Chip

needed; whereas here, it is not corrected until the entire iterative process is completed. The next significant difference between this division technique and the nonrestoring method is that the quotient bits selected are based on an estimate of the partial remainder and divisor rather than the exact values. The first advantage of this method is that an estimate can be obtained faster than the exact value. Second, a truncated estimate is acceptable, rather than a full-width estimate. Consequently, this method saves a significant amount of hardware and increases the speed of the operation. If one were to complete each partial remainder, up to three additional chips would be required and the delay would more than double.

The trick to the method lies in the quotient selection. The selection is based on partial remainder range transformations which guarantee that a quotient digit selected in one iteration may be corrected to the exact quotient digit on the next iteration. Therefore, although six quotient digits are determined per major iteration, an additional minor iteration is required to guarantee the least significant digit of the major iteration. The major and minor iteration terms refer to the architecture of the divide iterative hardware. The DIVx produces six quotient bits per machine cycle. This is a radix 64 division technique. However, the high radix division is accomplished by overlapping lesser radix divisions. In particular, there are three sets of radix 4 division groups. The first two sets are overlapped, so that the critical path through the radix 64 division is actually the critical path through two radix 4 divisions. A minor iteration is the path through one radix 4 division group. A major iteration is the path through the overlapped set of two radix 4 division groups, followed by the final radix 4 group. It is important to note that extra iterations do not adversely affect the corrected quotient. Finally, to produce the corrected quotient, the set of negative quotient digits is subtracted from the set of positive quotient digits, where each digit is properly radix 2 weighted, based on the order of selection. (That is, the first quotient digit selected is the most significant bit of the correct quotient.)

Vector Register File Chip — VRGx

The VAX 9000 architecture adds vector instructions to the standard VAX environment, thus a vector register file was required. There were two primary design requirements for the vector register file. First, the register file and associated cross-bar logic had to fit in a single multichip unit; and second, the register file had to perform read and write at different addresses within a single 16-ns clock cycle. These requirements could not be met with available memory and logic chips, thus necessitating the development of a fully custom vector register chip.

The vector register file is 64 bits wide and consists of 16 vector registers with 64 elements each. The vector register chip, VRGx, was developed as an 8-bit slice of the 64-bit vector register file. The chip contains 9216 bits of RAM for data storage and the cross-bar logic (6000 equivalent gates) that allows access from the five read ports and three write ports. Integrating the register memory and the cross-bar logic on the same chip allowed timing to be optimized so that the system timing requirements were met.

VRGx Chip Physical Features and Organization

The VRGx chip is fabricated using the MOSAIC III ECL process, which was not designed as a memory process. Coordination with the vendor resulted in the addition of an implant step for the memory-cell-bit line emitters. Key features of the process are three metal interconnect layers, oxide isolation, and polysilicon emitters with a drawn width of 1.75 microns.

Figure 9 shows the locations of the major circuit blocks in the VRGx chip. The major blocks of the VRGx chip are five read ports, three write ports, and 16 vector registers in the RAM bank array. The block diagram, Figure 10, shows the main data paths. The 16 vector registers are implemented as 64-word by 9-bit single port RAMs. Eight bits are a slice of the 64-bit vector register file and the ninth bit is for byte parity.

Timing

A register RAM can be read from one address and written from a different address in one 16-ns clock cycle. This dual operation is made possible by a 2 to 1 multiplexer on the RAM address inputs. The read address is applied during the first portion of the cycle, and the write address is applied during the second portion of the cycle. Splitting the clock cycle into read and write portions eliminates conflict between read and write ports in the event that a single register RAM is selected for both read and write. Read data is held in a latch during the second portion of the cycle and is unaffected by the write operation.

A single clock cycle consists of nonoverlapping clock phases A and B. Latches on the read and write

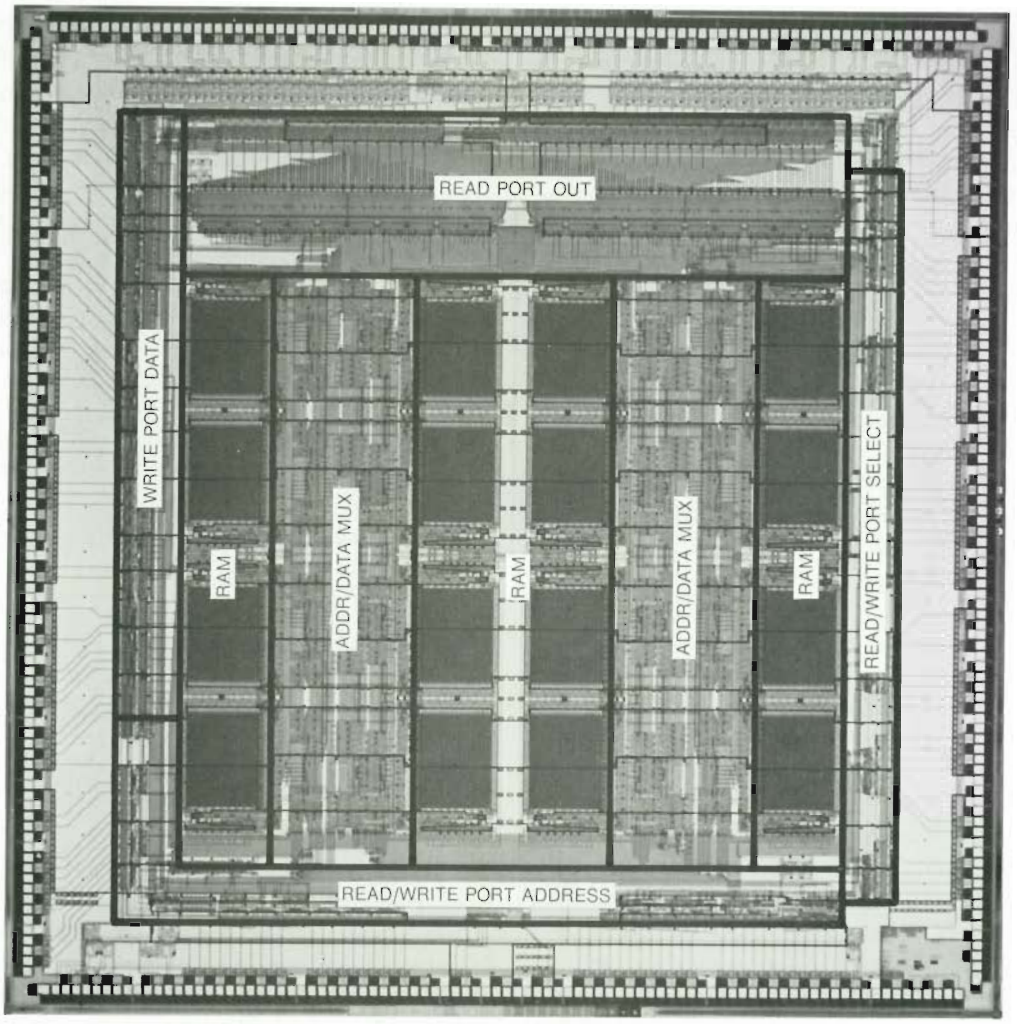


Figure 9 Photomicrograph of VRGx Chip

port inputs are clocked by phase A, and read port output latches are clocked by phase B. For a read operation initiated on phase A, the output read data becomes valid during phase B.

Cross-bar Logic

Cross-bar logic in the RAM bank array makes each of the 16 vector register RAMs independently accessible from the read and write ports. Enable inputs on the ports prevent invalid addresses from conflicting with intended addresses. Read and write ports may point to the same register RAM, but different write ports may not point to the same RAM. Also, different read ports may only point to the same RAM if the vector element address is the same. All conflicts must be resolved external to the chip.

A read port consists of an enable, a 4-bit register select, a 6-bit vector element address, and a 9-bit output. An enabled read port applies a register select code that points to a particular RAM bank. At that RAM bank, a 5 to 1 multiplexer selects the vector element address from the active read port and applies it to the read address of the RAM. Then the RAM output passes through a 16 to 1 multiplexer controlled by the register select code, so that the selected RAM output reaches the output of the active read port.

A write port consists of an enable, a 4-bit register select, a 6-bit vector element address, and a 9-bit write data input. An enabled write port applies a register select code that points to a particular RAM bank. At that RAM bank, a 3 to 1 multiplexer selects

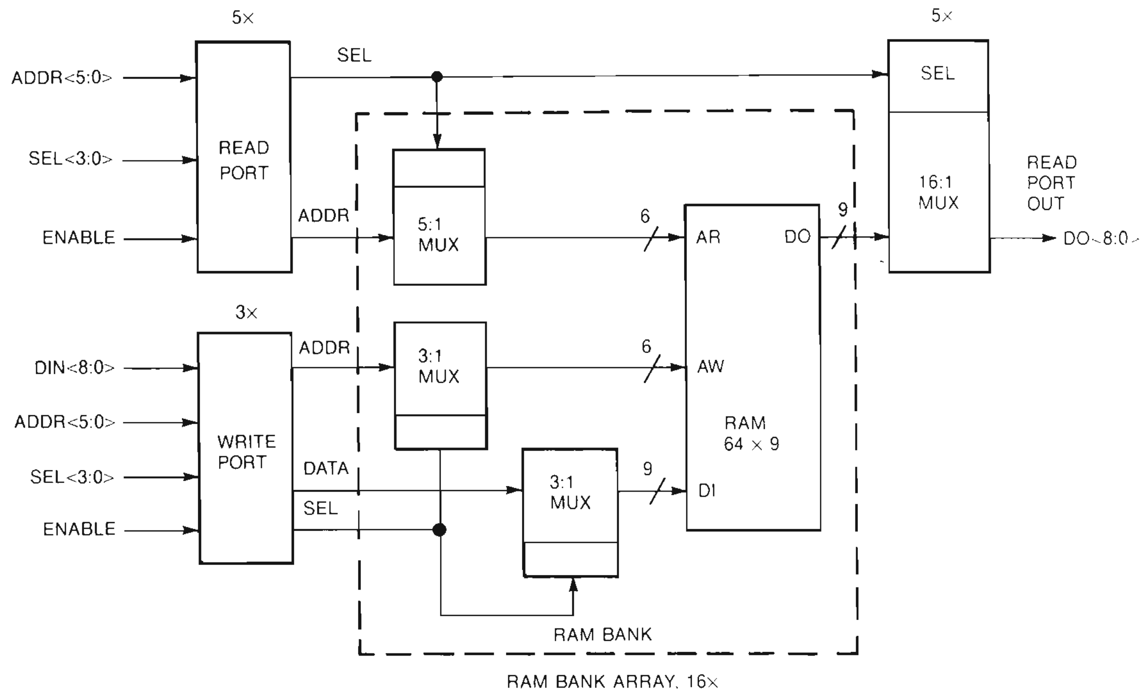


Figure 10 VRGx Chip Block Diagram

the vector element address from the active write port and applies it to the write address of the RAM. Also, a 3 to 1 multiplexer selects the write data from the active write port and applies it to the RAM data input.

RAM Technology

The normal transistors in an ECL process are of the NPN type, where the collector is a buried N-doped region. For memory cells, a lateral PNP transistor is placed in the same collector region, and the combined structure has the latching characteristics of a silicon controlled rectifier (SCR). The memory cell array in the 64 by 9 register RAMs is implemented with ECL SCR memory cells.

The SCR memory cell shown in Figure 11 consists of two cross-coupled SCR structures. Extra NPN emitters connect to the bit lines and provide a means of writing and sensing the cell. The "on" side of the cell saturates, allowing the bit line emitter to conduct in the inverse mode. Inverse gain of the bit line emitters must be limited to avoid excessive leakage into the unselected cells. An added process step applies a special base implant to the bit line emitters only to control their inverse gain.

Advantages of the SCR cell include good density, low standby power, large sense voltage differential, and low sensitivity to alpha-particle-induced soft errors. The cell has one limitation: excess charge storage due to write current can delay subsequent writing to the opposite state. This problem is eliminated with a special bit line current steering circuit that makes write current state dependent (Figure 11).

The SCR memory cell in Figure 11 is written by applying a high current (four times read current) to the "off" bit line emitter. The current steering transistors prevent this current from reaching a bit line emitter that is already "on." Thus, attempting to write a cell that is already in the desired state does not result in any additional cell current beyond the normal read current, and no additional charge storage occurs.

Other Chip Features

Other noteworthy chip features include scan logic, parity error detect logic, and a data pipeline for write port 0 data. Scan operation gives access to the register RAMs. In a single scan-in and scan-out operation, it is possible to read five registers and to write three registers.

Parity checking logic is used to detect input errors and set error flags. There is a parity check on the 9-bit write port data inputs. Another parity

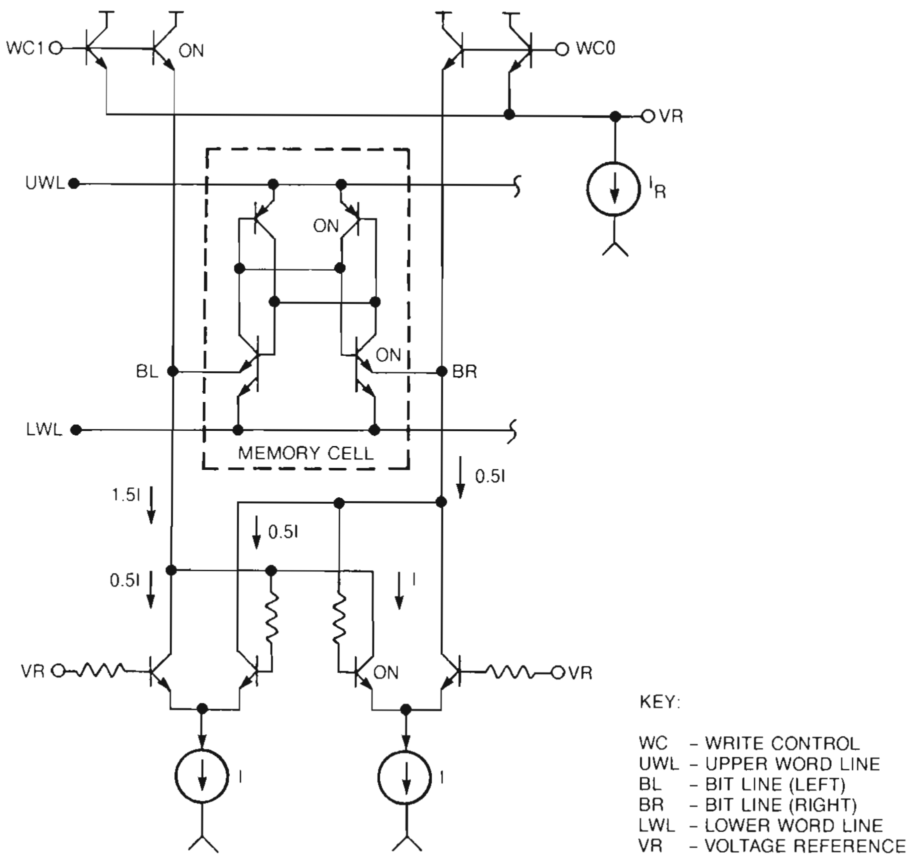


Figure 11 SCR Memory Cell with Bit Line Current Steering Circuit

checker is applied to address and control inputs. These are assigned to three parity groups, with a parity bit input for each group.

The write port 0 data pipeline allows a delay of one, two, and three clock cycles to be selected, delaying the write port data as necessary to resolve register access conflicts.

Self-timed RAM

In the VAX 9000 system—as in any high-performance CPU—fast memory is used for cache and control store applications. Engineers traditionally use very fast static RAMs within the CPU for memory. Logic designers, however, have long recognized that CPU performance is often limited as a result of the time needed to access data in these RAMs. This limitation is not only the result of the access time and write cycle performance of the devices themselves, but also of the off-chip circuitry and interconnect used for write pulse generation and distribution. The logic designers and technologists for the VAX 9000 knew that unless some architectural improvements were made to the traditional static RAM, much of the RAM performance improvements would be lost in the wiring interconnect. They also realized that Digital's memory suppliers would have to be convinced that a new RAM architecture would be marketable to their other customers. After several design iterations, the technologists submitted a set of specifications for a synchronous, self-timed RAM (STRAM) to several suppliers for their review. After extensive market surveys, our memory suppliers agreed that this new architecture could eventually become a new standard for high-speed static RAMs.

The VAX 9000 system requires two configurations of the basic STRAM device: 1K words by 4 bits, and 4K words by 4 bits. A block diagram of the STRAM is shown in Figure 12. The STRAM is similar to the traditional RAM in that it has chip select, input address and data, and output data. However, the STRAM also has several nontraditional inputs such as write, a differential clock, and a reference voltage (Vbb). Latches added to all inputs and outputs provide pipelined timing. An internal write pulse generator controls write operations and eliminates the need to generate and distribute the write pulse signal externally on the module. Also two optional output configurations are provided: a 50-ohm drive open emitter for standard parallel termination on the module, and a resistor and pulldown current source which is wired externally to implement STECL or on-chip source termination.

The clock buffer design allows inputs to be driven differentially from off-chip to minimize clock skew. The clock buffer is also designed to accommodate customers who are not greatly concerned about skew or who may be more concerned about conserving routing area. One input of the clock buffer may be tied to the output pin of the reference generator which provides the standard ECL threshold voltage (Vbb), allowing the other input of the clock buffer to be driven in a single-ended mode.

Input and output latches are clocked on opposite edges of the internal differential clock buffer. Timing diagrams are shown in Figure 13. On a falling edge of CLK H, data and address inputs flow into the RAM array.

If write is asserted during the next rising edge of CLK H, then a write cycle is initiated, and the input data is stored in the memory at the address presented at the ADR inputs. At the same time, the data is passed through the multiplexer and the output latch.

If write is deasserted on the rising edge of CLK H, then the STRAM is in a read cycle and input data is ignored. The data stored in the RAM at the address presented at the ADR inputs flows out to the multiplexer and output latch.

If chip select (CS) is deasserted prior to the rising edge of CLK H, then write and read operations are disabled and the output latches are reset low.

For proper operation of the STRAM, certain timing requirements must be fulfilled. The write operation is terminated by either the falling edge of

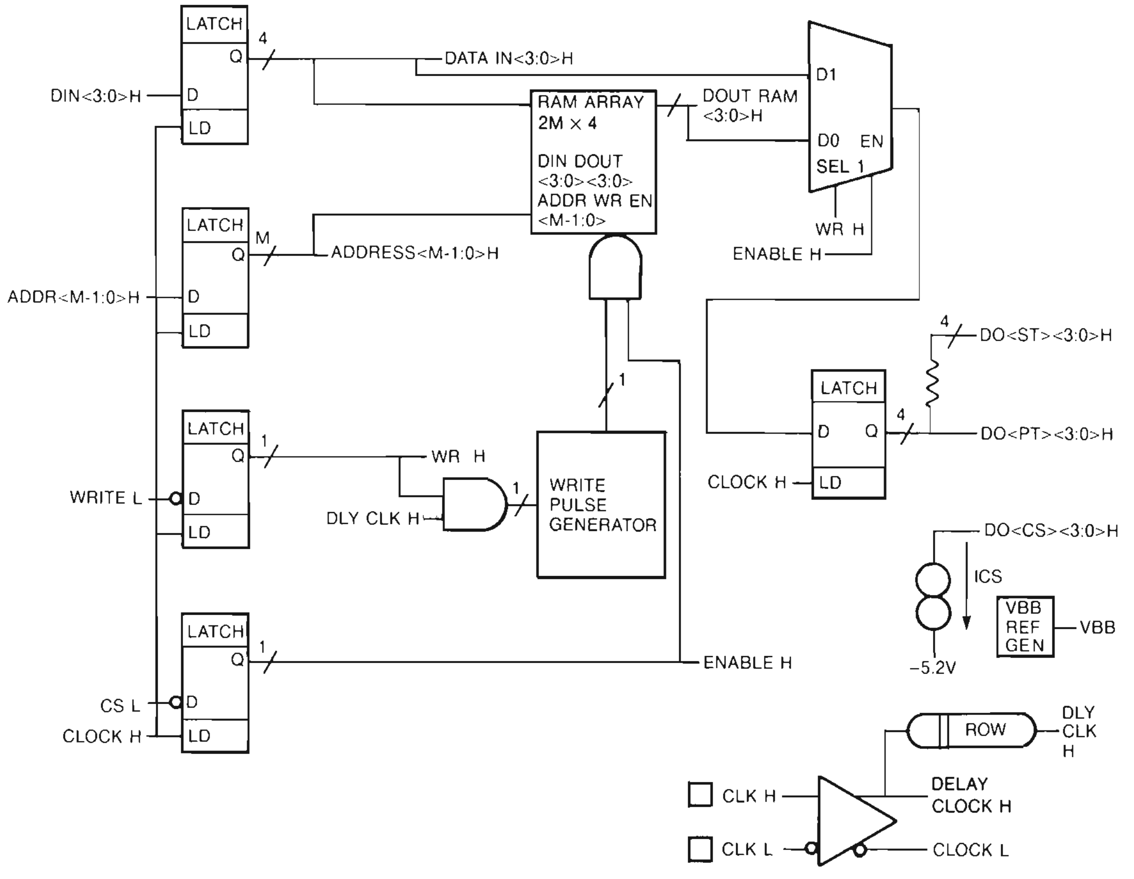


Figure 12 STRAM Block Diagram

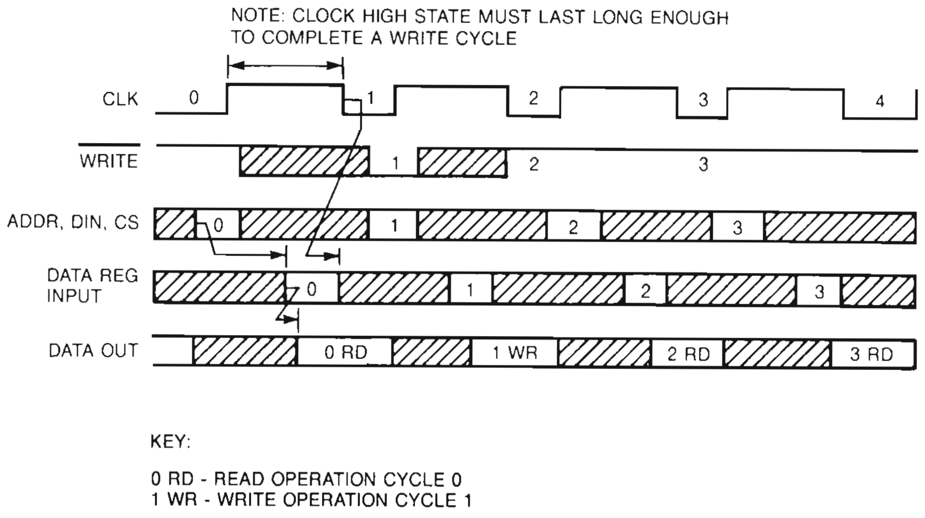

NOTE: CLOCK HIGH STATE MUST LAST LONG ENOUGH TO COMPLETE A WRITE CYCLE
CLK
WRITE
ADDR, DIN, CS
DATA REG INPUT
DATA OUT
0 RD
1 WR
2 RD
3 RD
KEY:
0 RD - READ OPERATION CYCLE 0
1 WR - WRITE OPERATION CYCLE 1

Figure 13 STRAM Timing Diagrams

CLK H or by the internal write pulse generator, whichever occurs first. Therefore CLK H must be asserted long enough to ensure that data is properly written into the memory array. The internal write pulse generator provides an output having the proper duration as determined by a string of gates.

Also, the assertion of the internal write pulse signal must be delayed by an amount equal to the internal access time of the RAM. In this way, the correct data is stored, and not the data previously stored in the input registers. The delay is accomplished by the row delay circuit, which is also simply a string of gates. These features give the STRAM its "self-timed" nature.

Acknowledgments

The authors would like to acknowledge the following individuals who participated in and contributed to the success of the VAX 9000 project: Jerry Weisbach, Andy Moroney, Bob Haller, Marc Lamere, Mark Hamel, Tom Senna, Dave McCall, Patty Kroesen, Rick Jones, Jim Jensen, Terry Skrypek, Eugene Marteney, Paul Guglielmi, Elaine Fite, Larry Herman, Bill Grundmann, Mark Pascarelli, Fran Richard, Linda Greska, Jack Mason, Chris Caiazzi, Roger Dame, Mike Normand Steve Sullivan, Rob Reinschmidt, Bob Bechdolt, Mike Warder, Mike Hickman, Brian Sadler, Wayne Nunn, Rita Wespi, Gene Yee, Bruce Smith, Alisyn Emerson, Jim Glanville.

References

1. D. Marshall and J. McElroy, "VAX 9000 Packaging, The Multi-Chip Unit," *Proceedings of COMPCON '90* (Spring 1990).
2. P. Zdebel et al., "MOSAIC III—A High Performance Bipolar Technology with Self-Aligned Devices," *Proceedings of IEEE 1987 Bipolar Circuits and Technology Meeting.*
3. D. Fite and T. Fossum, "Designing a VAX for High Performance," *Proceedings of COMPCON '90* (Spring 1990).
4. C. Baugh and B. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *Short Note at COMPCON 73, 7th Annual IEEE Computer Society International Conference* (February 1973).
5. C. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers,* Vol. EC-13 (February 1964): 14–17.
6. L. Dadda, "Some Schemes for Parallel Multipliers," *Colloque sur l'Algebre de Boole* (January 1965).
7. K. Hwang, *Computer Arithmetic Principles, Architecture, and Design* (New York: John Wiley and Sons, 1979): 213–283.

Richard A. Brunner
Dileep P. Bhandarkar
Francis X. McKeen
Bimal Patel
William J. Rogers Jr.
Gregory L. Yoder

Vector Processing on the VAX 9000 System

The VAX 9000 system provides the first emitter-coupled logic (ECL) implementation of the VAX vector architecture. The optional vector processor on the VAX 9000 system addresses the computing needs of numerically intensive applications with a peak performance of 125 MFLOPS for double-precision calculations. The innovative design of the vector register file allows the vector processor to overlap the execution of up to three vector instructions. Supported by both the VMS and ULTRIX operating systems, the vector processor on the VAX 9000 system provides four to five times performance improvement for vectorizable applications over its scalar processor.

For a long time, vector processing was the domain of large, expensive supercomputers such as the CRAY-1.[1] However, with the availability of low cost, pipelined floating point arithmetic chips, and the maturation of vectorizing compilers, vector processing has become a mainstream technology for scientific applications.[2] Applications that can benefit from vector processing include finite element analysis, signal processing, and computational fluid dynamics. The recent addition of integrated vector processing to the VAX architecture and its implementation on the VAX 9000 system provides these applications with an improvement in execution time of four to five times over that of a VAX 9000 system without vector processing. Vector processing extends the performance range of VAX systems.

The vector processor on the VAX 9000 system, referred to as the V-box, is the first emitter-coupled logic (ECL) implementation of the VAX vector architecture. The definition of the architecture and the development of the V-box started in 1986, two years after the design of the rest of the VAX 9000 CPU. Thus, the design of the V-box was synergistic with the definition of the VAX vector architecture. The major goal of the V-box design was to provide adequate vector performance (four to five times speed-up over scalar) without impacting the design of the remainder of the VAX 9000 CPU and the memory subsystem, which were too far along in development to change. With vector performance comparable to a CRAY-1 and a peak performance of 125 MFLOPS for double-precision calculations, the V-box fulfills this goal.

This paper describes the VAX vector architecture and its implementation by the VAX 9000 V-box. The first part of the paper discusses the architectural model that all VAX vector processors must follow. The second part shows the actual realization of this architecture in the VAX 9000 V-box and explains the innovative techniques the V-box uses to achieve good performance. The paper concludes with preliminary vector performance numbers for the VAX 9000 system on some standard vector benchmarks and a number of vector code examples.

VAX Vector Architecture

The VAX vector architecture defines the instruction set, registers, and behavior that all VAX vector implementations, such as the VAX 9000 V-box, must follow.[3] The vector architecture effort started in December 1985. At that time several CPU development projects were well underway, including the VAX 9000 system. With the expectation of providing four to five times performance improvement for vectorizable applications, Digital decided to add vector processing to the VAX 9000 system, even though the system was in an advanced stage of development. A decision also was made to provide a complementary metal oxide semiconductor (CMOS) implementation of the architecture on the VAX 6000 Model 400 system.[4]

Because both systems could not tolerate major changes without a major slip in schedule, the architecture required an approach that made few changes to the scalar processor — that part of a VAX processor that executes the regular VAX instruction set. Furthermore, because not all applications and markets can benefit from vector processing, Digital decided not to require vector processing on every new VAX processor. Therefore, vector processing is offered as an optional capability. The scalar processor decodes vector instructions and passed them to its associated vector processor. All processing of vector instructions is handled by the vector processor. Mechanisms are provided for vector-scalar synchronization and handling of vector exceptions by the scalar processor.

Although the architecture had to account for the implementation constraints of both ongoing CMOS and ECL projects, it had to be general and flexible enough to allow future, more integrated implementations at higher performance. The architecture also had to minimize its impact on the existing VMS and ULTRIX operating systems because major changes could significantly delay software support for vector processing.

Basic Architecture

The VAX vector architecture uses a vector-register-based design first pioneered by Seymour Cray.[1] There are 16 vector registers, each of which holds 64 elements; an element is 64-bits. Instructions which operate on longword integers or F_floating point data, only manipulate the low-order 32 bits of each element—sometimes referred to as longword elements.

A number of vector control registers control which elements of a vector register are processed by an instruction. The vector length register (VLR) limits the highest-numbered vector register element that is processed by a vector instruction. The vector mask register (VMR) consists of a 64-bit mask, in which each mask bit corresponds to one of the possible element positions in a vector register. When instructions are executed under control of the vector mask register, only those elements for which the corresponding mask bit is true are processed by the instruction. Vector compare instructions set the value of the vector mask register.

The vector count register (VCR) receives the number of elements generated by the compressed IOTA instruction, which is similar to COMPRESSED IOTA on the CRAY-2.[5] All VAX vector instructions use two-byte extended opcodes. Any necessary scalar operands (e.g., base address and stride for vector memory instructions) are specified by standard VAX scalar operand specifiers. The instruction formats allow all VAX vector instructions to be encoded in seven classes. The seven basic instruction groups and their opcodes are shown in Table 1.

Within each class, all instructions have the same number and types of operands, which allows the scalar processor to use block-decoding techniques. The differences in operation between the individual instructions within a class are irrelevant to the scalar processor and need only be known by the vector processor. Important features of the instruction set are

- Support for random-strided vector memory data through gather (VGATH) and scatter (VSCAT) instructions
- Generation of compressed IOTA vectors (through the IOTA instruction) to be used as offsets to the gather and scatter instructions
- Merging vector registers through the VMERGE instruction
- The ability for any vector instruction to operate under control of the vector mask register

Additional control information for a vector instruction is provided in the vector control word (shown as cntrl in Table 1), which is a scalar operand to most vector instructions. The control word operand can be specified using any VAX addressing mode. However, VAX compilers generally use immediate mode addressing (that is, place the control word within the instruction stream). The format of the vector control word is shown in Figure 1.

The Va, Vb, and Vc fields indicate the source and destination vector registers to be used by the instruction. These fields also indicate the specific operation to be performed by a vector compare or convert instruction. The MOE bit indicates whether the particular instruction operates under control of the vector mask register. The MTF bit determines what bit value corresponds to "true" for vector mask register bits. It allows a compiler to vectorize if-then-else constructs. The EXC bit is used in vector arithmetic instructions to enable integer overflow and floating underflow exception reporting. The MI bit is used in vector memory load instructions to indicate modify-intent. Figure 2 shows the encoding for some typical VAX vector instructions.

Vector Execution Model

With the addition of vector processing, a typical VAX processor consists of a scalar processor and an associated vector processor; the two are referred to as a scalar/vector pair. A VAX multiprocessor system

Table 1 VAX Vector Instruction Classes

Vector Memory, Constant-stride
opcode cntrl, base, stride

VLDL	Load longword vector data
VLDQ	Load quadword vector data
VSTL	Store longword vector data
VSTQ	Store quadword vector data

Vector Memory, Random-stride
opcode cntrl, base

VGATHL	Gather longword vector data
VGATHQ	Gather quadword vector data
VSCATL	Scatter longword vector data
VSCATQ	Scatter quadword vector data

Vector-Scalar Single-precision Arithmetic
opcode cntrl, scalar

VSADDL	Integer longword add
VSADDF	F_floating add
VSBICL	Bit clear longword
VSBISL	Bit set longword
VSCMPL	Integer longword compare
VSCMPF	F_floating compare
VSDIVF	F_floating divide
VSMULL	Integer longword multiply
VSMULF	F_floating multiply
VSSLLL	Shift left logical longword
VSSRLL	Shift right logical longword
VSSUBL	Integer longword subtract
VSSUBF	F_floating subtract
VSXORL	Exclusive-or longword
IOTA	Generate compressed IOTA vector

Vector Control Register Read
opcode regnum, destination

MFVP	Move from vector processor

Vector Control Register Write
opcode regnum, scalar

MTVP	Move to vector processor

Vector-scalar Double-precision Arithmetic
opcode cntrl, scalar

VSADDD	D_floating add
VSADDG	G_floating add
VSCMPD	D_floating compare
VSCMPG	G_floating compare
VSDIVD	D_floating divide
VSDIVG	G_floating divide
VSMULD	D_floating multiply
VSMULG	G_floating multiply
VSSUBD	D_floating subtract
VSSUBG	G_floating subtract
VSMERGE	Merge

Vector-vector Arithmetic
opcode cntrl or regnum

VVADDL	Integer longword add
VVADDF	F_floating add
VVADDD	D_floating add
VVADDG	G_floating add
VVBICL	Bit clear longword
VVBISL	Bit set longword
VVCMPL	Integer longword compare
VVCMPF	F_floating compare
VVCMPD	D_floating compare
VVCMPG	G_floating compare
VVCVT	Convert
VVDIVF	F_floating divide
VVDIVD	D_floating divide
VVDIVG	G_floating divide
VVMERGE	Merge
VVMULL	Integer longword multiply
VVMULF	F_floating multiply
VVMULD	D_floating multiply
VVMULG	G_floating multiply
VVSLLL	Shift left logical longword
VVSRLL	Shift right logical longword
VVSUBL	Integer longword subtract
VVSUBF	F_floating subtract
VVSUBD	D_floating subtract
VVSUBG	G_floating subtract
VVXORL	Exclusive-or longword
VSYNC	Synchronize vector memory access

15	14	13	12	11–8	7–4	3–0
MOE	MTF	EXC MI	0	VA/CONVERT FCN	VB	VC/COMPARE FCN

Figure 1 Vector Control Word

comprises a number of these scalar/vector pairs. Asymmetric configurations can exist when only some of the VAX processors in a multiprocessor system contain a vector processor.

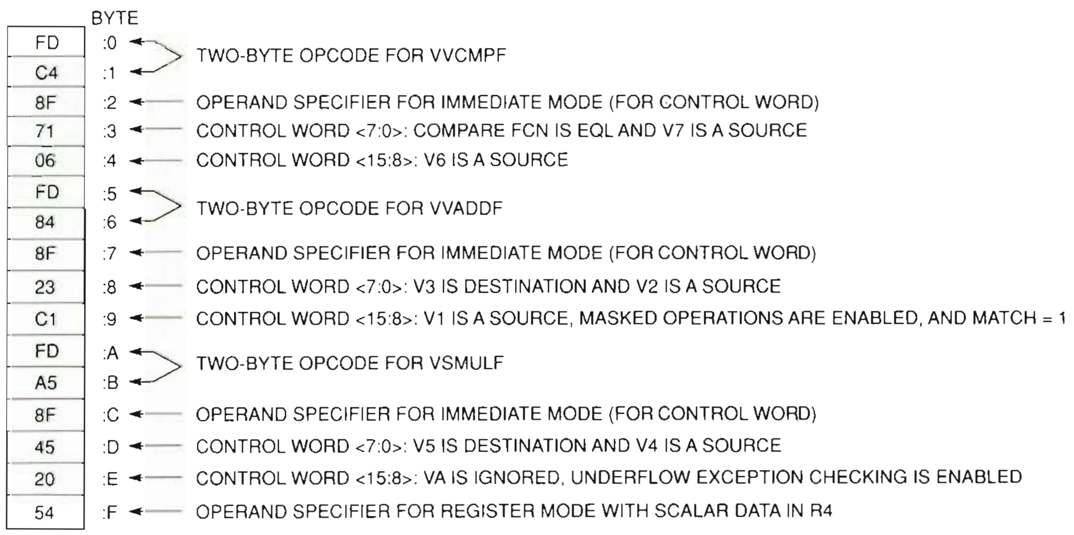
For good performance, the scalar processor operates asynchronously from its vector processor whenever possible. Asynchronous operation allows the execution of scalar instructions to be overlapped with the execution of vector instructions. Furthermore, the servicing of interrupts and scalar exceptions by the scalar processor does not disturb the execution of the vector processor, which is freed from the complexity of resuming the execution of vector instructions after such events. However, the asynchronous execution does cause the reporting of vector exceptions to be imprecise. Special instructions, which are described in the Synchronization section, are provided to ensure synchronous operation when necessary.

Both scalar and vector instructions are initially fetched from memory and decoded by the scalar processor. If the opcode indicates a vector instruction, the opcode and necessary scalar operands are issued to the vector processor and placed in its instruction queue. The vector processor accesses memory directly for any vector data that it must read or write. For most vector instructions, once the scalar processor successfully issues the vector

```
ASSEMBLER FORMAT:
   VVEQLF      V6,V7              ;IF V6[i] = V7[i] THEN VMR[i] = 1, ELSE VMR[i] = 0
                                  ; (VVEQLF IS A VVCMPF PSEUDO-OPCODE)
   VVADDF/1    V1,V2,V3           ; V3 = V1 + V2. DO ADDITION UNDER CONTROL OF VMR
                                  ; WITH MATCH = 1
   VSMULF/U    R4,V4,V5           ; V5 = R4*V4 WITH UNDERFLOW EXCEPTION CHECKING ENABLED

INSTRUCTION FORMAT:
   VVCMPF      cntrl.rw           ; INSTRUCTION CONSISTS OF OPCODE AND CONTROL WORD
   VVADDF      cntrl.rw           ; INSTRUCTION CONSISTS OF OPCODE AND CONTROL WORD
   VSMULF      cntrl.rw, src.rl   ; INSTRUCTION CONSISTS OF OPCODE, CONTROL WORD, AND SCALAR SOURCE

ENCODING IN MEMORY:
```



```
           BYTE
     FD    :0  ┐
     C4    :1  ┘  TWO-BYTE OPCODE FOR VVCMPF
     8F    :2     OPERAND SPECIFIER FOR IMMEDIATE MODE (FOR CONTROL WORD)
     71    :3     CONTROL WORD <7:0>: COMPARE FCN IS EQL AND V7 IS A SOURCE
     06    :4     CONTROL WORD <15:8>: V6 IS A SOURCE
     FD    :5  ┐
     84    :6  ┘  TWO-BYTE OPCODE FOR VVADDF
     8F    :7     OPERAND SPECIFIER FOR IMMEDIATE MODE (FOR CONTROL WORD)
     23    :8     CONTROL WORD <7:0>: V3 IS DESTINATION AND V2 IS A SOURCE
     C1    :9     CONTROL WORD <15:8>: V1 IS A SOURCE, MASKED OPERATIONS ARE ENABLED, AND MATCH = 1
     FD    :A  ┐
     A5    :B  ┘  TWO-BYTE OPCODE FOR VSMULF
     8F    :C     OPERAND SPECIFIER FOR IMMEDIATE MODE (FOR CONTROL WORD)
     45    :D     CONTROL WORD <7:0>: V5 IS DESTINATION AND V4 IS A SOURCE
     20    :E     CONTROL WORD <15:8>: VA IS IGNORED, UNDERFLOW EXCEPTION CHECKING IS ENABLED
     54    :F     OPERAND SPECIFIER FOR REGISTER MODE WITH SCALAR DATA IN R4
```

Figure 2 Vector Instruction Encoding

instruction, it proceeds to process other instructions and does not wait for the vector instruction to complete. An execution model is shown in Figure 3.

When the scalar processor attempts to issue a vector instruction, it checks to see if the vector processor is disabled—that is, whether it will accept further vector instructions. If the vector processor is disabled, then the scalar processor takes a "vector processor disabled" fault. An operating system handler is then invoked on the scalar processor to examine the various error-reporting registers on the vector processor to determine the disabling condition. The vector processor disables itself to report the occurrence of vector arithmetic exceptions or hardware errors. The operating system disables the vector processor, usually to indicate the unavailability of the vector processor, by writing to a privileged vector register. If the disabling condition can be corrected, the handler enables the vector processor and directs the scalar processor to reissue the faulted vector instruction.

Within the constraint of maintaining the proper ordering among the operations of data-dependent instructions, the architecture explicitly allows the vector processor to execute any number of the instructions in its queue concurrently and retire them out of order. Thus, a VAX vector implementation can chain and overlap instructions to the extent best suited for its technology and cost-performance. In addition, by making this feature an explicit part of the architecture, software is provided with a programming model that ensures correct results regardless of the extent a particular implementation chains or overlaps. This approach differs with respect to some other existing vector architectures, such as the IBM S/370 vector architecture, which give the appearance of sequential instruction execution.[6]

A VAX vector implementation may have its own memory management hardware, translation buffer, and cache; or it may share those of the scalar processor. In high-end vector implementations, such as the VAX 9000 system, the vector and scalar processors are tightly coupled. The problems of limited chip area and translation buffer and cache coherency can be lessened by allowing high-speed memory management hardware and cache to be shared by both vector and scalar processors. For other implementations, such as the VAX 6000 Model 400 system, the vector and scalar processors are not so tightly coupled, and there is a performance advantage in allowing separate memory management hardware and cache.[4] Little additional effort is necessary by an operating system to support separate vector memory management hardware and cache.

A vector processor can treat vector memory management exceptions (MME) in a synchronous manner, as the VAX 9000 V-box does. Once the scalar processor issues a vector memory instruction, it pauses until the vector processor determines whether an MME will be encountered by the instruction. If an MME will occur, then a precise

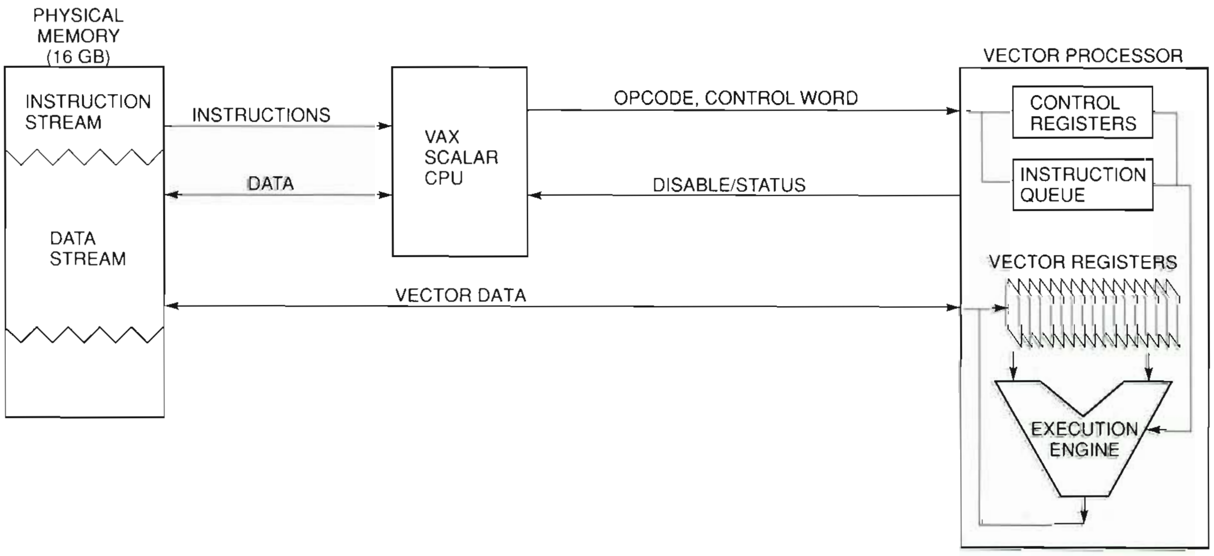


Figure 3 Vector Execution Unit

exception is taken on the scalar processor and the appropriate operating system handler is invoked. If no MME will occur, the scalar processor proceeds to process other instructions and the vector processor completes the memory instruction. In the case of referencing a unity-strided vector, which occurs most frequently, the MME checking takes only a short time at the beginning because the vector is contained in two or less pages. (MME checking is done at the page level.)

Context Switching

Because of the asynchronous operation of the vector and scalar processors, the vector context state of a process is separate from its scalar context state. Thus, it is possible for an operating system to swap in a new process to the scalar processor while allowing the vector context of the previous process to remain on the vector processor. When the previous process is swapped out, the vector processor is disabled by the operating system to prevent other processes from accessing this vector context.

If the subsequent processes do not use the vector processor, then the operating system avoids the overhead of saving and subsequently restoring 8 kilobytes (KB) of vector context state for the original process. If another process does use the vector processor, the operating system must reenable the vector processor, save the vector state of the original process, load the vector context of the new process, and, finally, make the vector processor available. This full context switch can take up to 100 microseconds on the VAX 9000 system.

Assuming that only a few processes require the vector processor, it is likely that when the original process is rescheduled to the same scalar/vector pair, the process will find its vector context state residing on the vector processor. By using this technique, which is referred to as "cheap vector context switching," both the VMS and ULTRIX operating systems reduce the time required to swap in a process that uses the vector processor.

Exceptions

Most of the exceptions encountered by VAX vector instructions are identical to those that occur for VAX scalar instructions. The arithmetic exceptions are exactly the same. The memory management exceptions have been extended to include two new vector exceptions: vector I/O space reference and vector alignment fault. As in the VAX scalar architecture, the reporting of floating underflow and integer overflow exceptions can be disabled by setting the EXC bit in the vector control word.

Vector arithmetic exceptions are reported in an imprecise manner by vector processor disabled faults. When an exception occurs in the processing of a vector element, the vector processor records the exception in both a privileged exception register (the vector arithmetic exception register, VAER) and in the corresponding element of the destination vector register specified by the instruction. The vector processor then disables itself from receiving further vector instructions. However, the vector processor continues to execute the instruction that encountered the exception to completion by processing the remaining vector register elements.

As stated earlier, memory management exceptions can be reported precisely by a VAX vector processor to its scalar processor, as the VAX 9000 V-box does, and the scalar processor takes a normal VAX memory management fault. Exception information is placed on the stack in the same format as for scalar memory management exceptions. The use of the same format minimizes the effort needed by an operating system to support these exceptions.

Memory management exceptions were extended for vectors to include two new exception parameter bits: vector I/O space reference and vector alignment fault. A vector I/O space reference occurs whenever an attempt is made to load or store vector data to I/O space. Because of the performance degradation of unaligned memory data, a vector alignment fault occurs whenever an element being accessed by a vector memory instruction does not begin at an address that is an integer multiple of the length of the element in bytes. For example, a longword (4-byte) element in memory should begin at an address which is an integer multiple of 4 bytes.

Synchronization

In most cases, it is desirable for the vector processor to operate asynchronously with the scalar processor to achieve good performance. However, there are cases in which the operation of the vector and scalar processors must be synchronized to ensure correct results. Rather than forcing the vector processor to detect and automatically provide synchronization in these cases, the architecture provides special instructions, which software can use, to accomplish the synchronization. Some of these instructions are discussed below. Software must determine when to use these synchronization instructions to ensure correct results or establish exception checkpoints. Given the necessary sophistication of vectorizing compilers, this requirement is not onerous.

Vector and scalar memory references may be issued simultaneously. Therefore, these references must be synchronized to prevent a conflict from occurring when accessing shared memory locations. This synchronization is provided by the MSYNC function of the MFVP instruction. Once the MSYNC function is invoked, the scalar processor does not issue further instructions until all previous vector and scalar memory references have completed.

Because the vector and scalar processors execute asynchronously, software cannot determine when a vector exception will be reported. However, software requires that exceptions be reported at certain checkpoints. For example, exceptions incurred in a procedure must be reported within the context of that procedure before another procedure is called. This exception reporting synchronization is provided by the SYNC function of the MFVP instruction. Once SYNC is invoked, the scalar processor does not issue further instructions until the exceptions of previous vector instructions, if any, are reported.

VAX 9000 V-box Overview

The VAX 9000 V-box is one of four tightly coupled, parallel function units that compose the VAX 9000 CPU. As such, it shares, with the rest of the CPU, both the large 128KB data cache and the very fast address translation hardware. As a result, the V-box has very fast access to memory data. The V-box is connected to the CPU through the scalar execution unit as shown in Figure 4. This connection consists of a 64-bit data path, which brings instructions and data to the vector unit, and a 32-bit path, which sends data to the scalar unit. All vector memory instructions send data through this data path.

As Figure 4 also shows, the V-box is composed of the following subunits: vector register unit, vector add unit, vector multiply unit, vector mask unit, vector address unit, and vector control unit. Each of these subunits can function in parallel, which allows up to two vector arithmetic instructions and one vector memory instruction to be executed simultaneously. Crucial to this instruction overlapping ability is the vector register unit, which supports up to eight simultaneous accesses from the other subunits.

Physically, the V-box resides on the same planar board as the remainder of the VAX 9000 CPU. Three multichip units (MCUs) are reserved for the V-box, which is a field-installable option. The V-box comprises 25 ECL Motorola Macrocell Array IIIs (MCA3).[7] (For brevity, a macrocell array is referred to as a "chip" in this paper.) The operation of these subunits and the techniques used to enhance their performance are described in the following sections.

Vector Control Unit

The vector control unit receives and coordinates the execution of vector instructions within the V-box. The VAX 9000 scalar execution engine (E-box) transfers both an encoded version of the vector instruction and the necessary scalar data to the unit, which loads the instruction and data into a

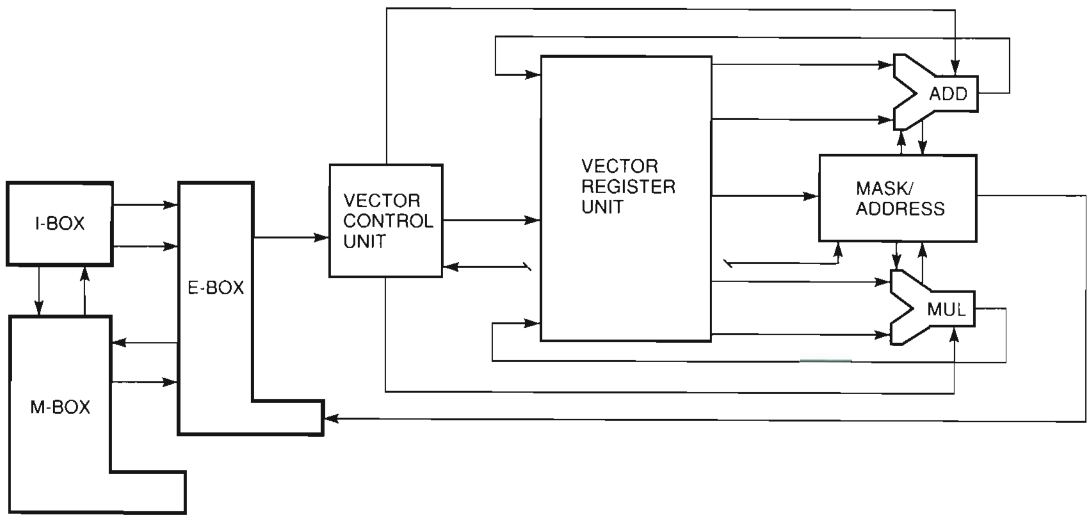


Figure 4 V-box Organization (with VAX 9000 CPU)

circular queue as shown in Figure 5. The queue can buffer a few pending instructions while the remaining V-box subunits are executing others. Without the queue, the V-box could not accept pending instructions when all of its subunits are busy, thus, propagating a stall condition to the scalar execution unit and resulting in poor performance.

The scalar data that is required by a vector instruction is placed in the queue one location behind the instruction quadword. Whenever the queue contains two entries, the vector control unit returns a signal to the scalar execution unit and requests that subsequent instruction issue be delayed until the number of entries in the queue has diminished to one or less. The queue is circular in nature and wraps around to the beginning automatically.

When an instruction is loaded into the queue, a pointer directs the instruction to the decode logic shown in Figure 5. If there is enough instruction data available in the queue and the necessary subunit is not busy, then the vector control unit sends the instruction data from the queue to the register conflict logic. The register conflict logic determines if the vector registers required by the instruction are already in use by the other subunits, a condition called register conflict. The determination is made by comparing the vector register addresses that are to be used by already executing vector instructions in the next cycle against the vector register addresses required by the new instruction. If none of the addresses overlap then the instruction is free to issue. If an overlap does exist, the instruction is held until the next cycle, when it can then be issued to the appropriate subunit. (The lack of significant cycle delay in this case is due to the optimal design of the vector register unit.) If there are no register conflicts, the instruction is issued immediately to the appropriate subunit.

As the vector control unit issues the instruction to the subunit, it also sends scalar source operands, if any, and the addresses of the vector registers required by the instruction to the vector register unit. The vector register unit latches the scalar data for the duration of that instruction. For each cycle of the instruction's execution, the register unit then sends the necessary scalar and register data to the appropriate subunit. The vector control unit also contains the vector length register and sends a copy of it with every instruction that is issued to a subunit. By supplying each subunit with a copy of the vector length register, writes to the register by MTVP instructions do not affect instructions currently executing under the register's previous value. Without this mechanism, writes to the vector length register would be delayed until previously

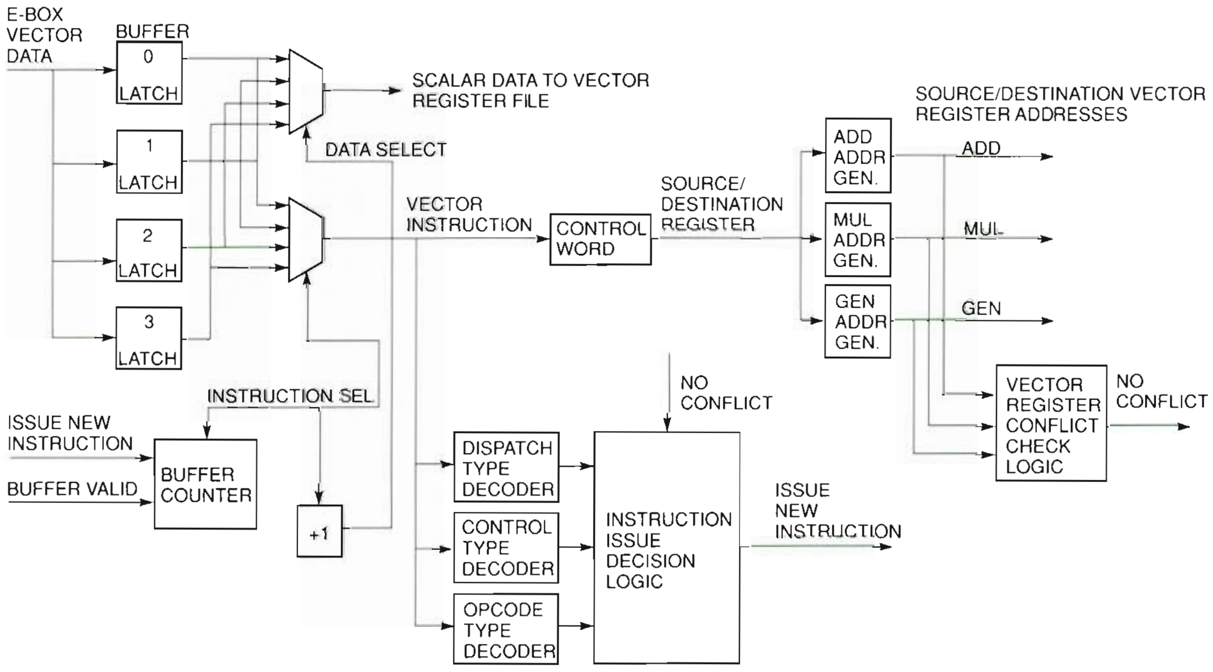


Figure 5 Vector Control Unit

executing instructions had finished, which would result in poor performance.

Upon reaching the subunit, most vector instructions execute at one cycle per element, after the initial pipeline latency. However, the vector divide instructions (VSDIV and VVDIV) execute at a varying number of cycles, depending on the floating point format (F, D, or G). (To simplify the vector control logic, no other vector instructions are issued once a vector divide starts.) Results are returned to the vector register unit or vector mask unit as they are generated, depending on the instruction.

As described earlier, microcode in the scalar execution engine encodes vector instructions into an instruction quadword before passing them to the V-box. Table 2 shows the high-order 32 bits of the format used for every instruction sent to the V-box. This quadword contains fields that indicate the instruction, appropriate V-box subunit to execute the instruction, and format of the vector control word. The low-order 32 bits of the instruction quadword contain the vector control word for the vector instruction. The instruction quadwords present the V-box with a fixed format instruction that smoothly fits into a fixed-length instruction queue, requires little subsequent decoding, and has fields that can be directly gated to selection logic. As a result, the time needed by the V-box to decode vector instructions is reduced and performance is increased.

Vector Register Unit

The vector register unit or file, as its name implies, contains the logic and fast memory that implement the 16 VAX vector registers on the V-box. The block diagram of the vector register file is shown in Figure 6. The vector register file has three write ports and five read ports. By using the innovative technique described below, these ports provide the multiple accesses needed to feed two operands per cycle to the vector add and multiply units, and one operand to the vector address-mask unit. This unit is the single largest contributor to the excellent vector performance of the VAX 9000 system.

The file consists of 16 vector registers. Each register contains 64 elements, and each element is 72-bits wide (64 data, 8 parity). The vector register file is implemented as a byte-sliced custom chip, which has a single parity bit per data port. Three writes and five reads to the file can occur simultaneously in any cycle. All writes must be to different register banks. However, multiple reads can occur to the same bank if the same element is required by each read access. Internally to the vector register unit, reads occur during the first half of the cycle, and writes occur during the last half. A write and read enabling signal is generated for each register bank every cycle. Each cycle, data is selected from one of the three write ports to be written into any enabled register banks. Write port 0 has a four-stage pipe to buffer data coming from the E-box, through the control logic, which cannot be written due to a register bank conflict. The vector register file also has three scalar registers (one each for the vector address-mask unit, vector add unit, and vector multiply unit) to hold scalar source operands for vector-scalar instructions. Write port 0 is used to write these registers. Each enabled read port selects an element from one of the 16 register banks or scalar registers (for vector-scalar instructions) and transfers it to one of the other subunits.

The vector register file uses a technique referred to as "barber poling" to improve the use of chaining and overlapped instruction execution. As Figure 7 shows, barber poling spreads each architecturally defined vector register across all vector register banks. Elements are laid out such that the first vector element of each vector register is in location 0 of the same physical register bank and element *b* of vector register *n* is in location *b* of vector register bank *([n+b] modulo* 16*)*.

By using this technique, a vector register conflict causes the vector control unit to delay the issuing of a new vector instruction for no more than three cycles. If the more standard technique of placing all elements of one vector register in the same bank were used, a vector register conflict could cause the execution of a new instruction to be delayed by 64 cycles. The 64-cycle delay would have frustrated attempts at overlapping and severely degraded the vector performance of the VAX 9000 system.

Vector Add Unit

The vector add unit executes most vector instructions, including both floating point and integer addition, subtraction, comparison; vector convert; vector shift logical; vector logical operations; and vector merges. For brevity, these instructions are referred to as add-class instructions. One of the challenges in designing the vector add unit was the need to perform both integer and floating point arithmetic.

The organization of the vector add unit is shown in Figure 8. It is a pipelined structure that comprises two identical chips for unpacking and aligning operands (VFSA and VFSB); one chip for performing arithmetic and logical operations (VFAD); and a

Table 2 Encoded Instruction Quadword (bits <63:32>)

Vector Instruction	OPCODE <39:32>	Control Word Type <42:40>	Dispatch Type <46:43>
VVSUBF/VSSUBF	0F9	2/6	0
VVSUBG/VSSUBG	0DB	2/6	0
VVSUBD/VSSUBD	0D2	2/6	0
VVSUBL/VSSUBL	0F6	2/6	0
VVCMPL/VSCMPL	0F5	3/7	0
VVSLL/VSSLL	034	2/6	0
VVSRL/VSSRL	026	2/6	0
VVBISL/VSBISL	086	2/6	0
VVBICL/VSBICL	08E	2/6	0
VVXORL/VSXORL	088	2/6	0
VVMERGE/VSMERGE	0AE	5/1	0
VVADDD/VSADDD	092	2/6	0
VVADDF/VSADDF	0B9	2/6	0
VVADDG/VSADDG	09B	2/6	0
VVADDL/VSADDL	0B6	2/6	0
VVCMPD/VSCMPD	0D5	3/7	0
VVCMPF/VSCMPF	0FD	3/7	0
VVCMPG/VSCMPG	0DD	3/7	0
VVCMPD/VSCMPD	0D5	3/7	0
VVCVTDF	011	4	0
VVCVTDL	016	4	0
VVCVTFD	03A	4	0
VVCVTFG	03B	4	0
VVCVTFL	03E	4	0
VVCVTGF	019	4	0
VVCVTGL	01E	4	0
VVCVTLD	032	4	0
VVCVTLF	031	4	0
VVCVTLG	033	4	0
VVCVTDL	017	4	0
VVCVTFL	03F	4	0
VVCVTGL	01F	4	0
VVMULL/VSMULL	003	2/6	1
VVMULF/VSMULF	004	2/6	1
VVMULD/VSMULD	005	2/6	1
VVMULG/VSMULG	006	2/6	1
VVDIVF/VSDIVF	00C	2/6	4
VVDIVD/VSDIVD	00D	2/6	4
VVDIVG/VSDIVG	00E	2/6	4
VLDL	001	0	2
VLDQ	002	0	2
Block load	00C	0	2
VSTL	003	0	2
VSTQ	004	0	2
VGATHL	005	1	2
VGATHQ	006	1	2
VSCATL	010	1	2
VSCATQ	011	1	2
IOTA	012	0	2
Load VLR	007	0	2
Load low VMR	009	0	2
Load high VMR	00A	0	2
Store low VMR	00D	0	3
Store high VMR	00E	0	3
Store unaligned address	013	0	3
Load VPSR	014	0	2
Load VAER	015	0	3
Store VAER	008	0	3
RESET	00F	2	3

Bits <63:47> are reserved.

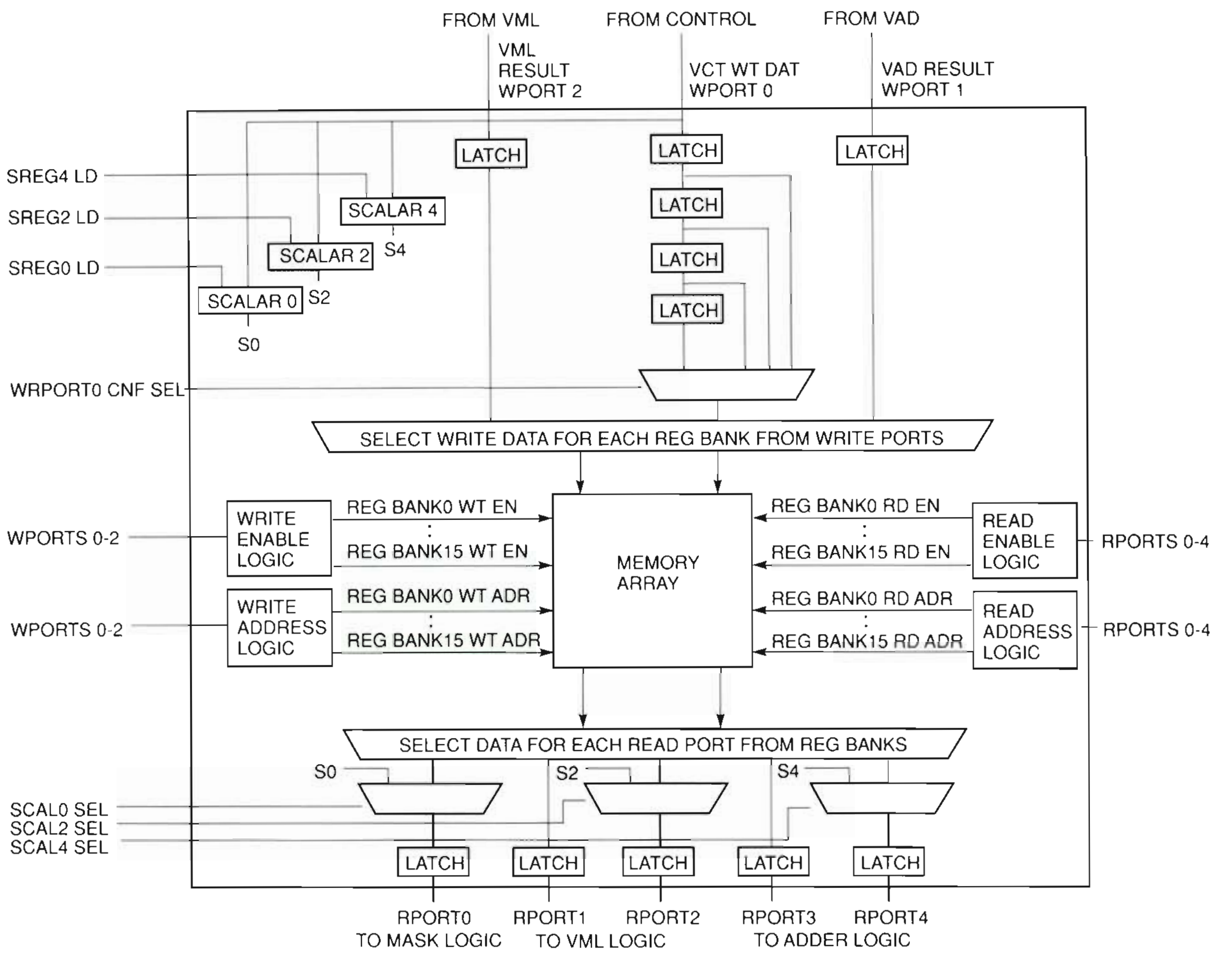


Figure 6 Vector Register Unit

remaining chip for normalizing, rounding, and packing the result (VFPK). The data paths between the chips are all 64-bits wide.

The pipeline latency through this unit for both single-precision (integer and F_floating) and double-precision (G_floating and D_floating) formats is only three cycles. Thus, the vector/scalar cross-over number for add-class instructions is quite small (that is, the minimum number of vector elements needed for the V-box to surpass the performance of the remainder of the VAX 9000 CPU for this class of instructions.) As a result, the V-box achieves good performance for add-class instructions with small-sized vectors and large-sized vectors (large-sized vectors being naturally favored by the technique of pipelining).

When the vector add unit begins to execute an instruction, it receives two source elements from the vector register unit each cycle. The elements are latched into the unpacking logic, one element for

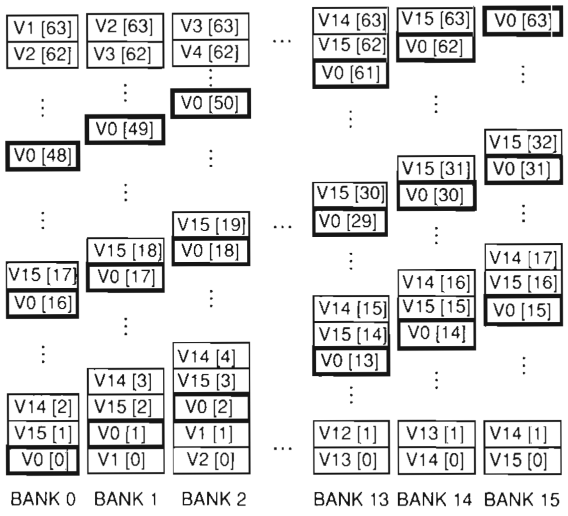


Figure 7 Barber Poling

each of the two chips. During the next cycle, each unpacking chip concurrently unpacks and aligns its source element, if necessary, and forwards the result to the addition or logical-operation logic, depending on the instruction. Within the same cycle, the addition chip uses the two sources from the unpacking logic to generate a result, which is then latched.

During the final cycle, the result is sent to the packing chip, which normalizes, rounds, and packs, if necessary, the result and sends it to the vector register unit to be written. Exception checking and reporting are also done in the last cycle by the packing chip, which maintains the vector add unit's copy of the vector arithmetic exception register (VAER). When the instruction completes, the vector add unit sends its VAER copy to the vector mask unit to be merged with the VAER copy from the vector multiply unit.

The vector add unit does not differentiate between masked and unmasked vector instructions. The complexity of skipping over masked-out elements would have added extra cycles of pipeline latency and resulted in less performance for small-sized vectors. For masked as well as unmasked instructions, the vector add unit operates from the first up to the last element (as indicated by the vector length register) of both source registers. The actual masking of results is handled by the vector control unit, which blocks the vector register unit from receiving masked-out results as they are being sent by the vector add unit. However, the packing chip does use vector mask register bits to suppress exception generation for results that are masked out.

Floating Point Operation When executing vector floating point instructions, the unpacking logic takes the various fields of a floating point element and expands and rearranges it into a more convenient format for the addition logic, i.e., the element is "unpacked." As a result of this process, the addi-

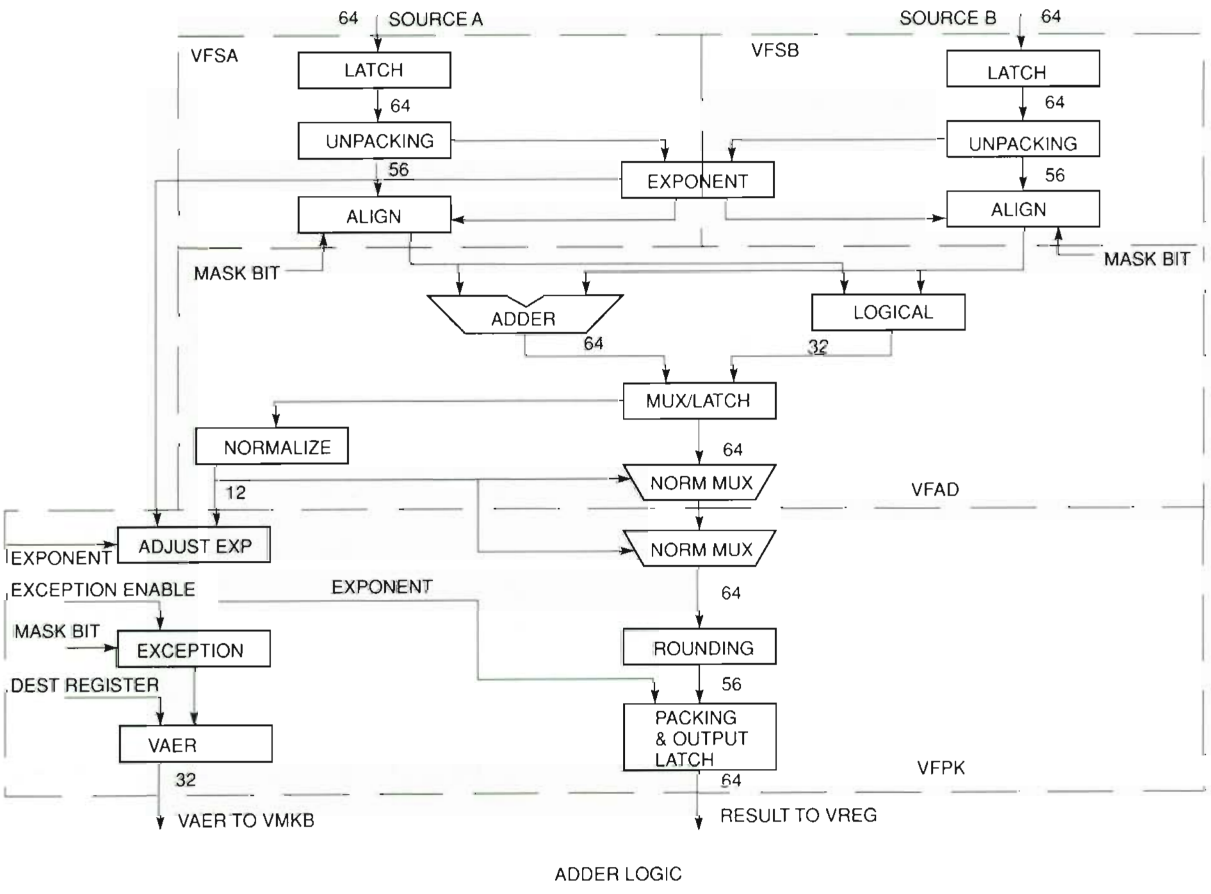


Figure 8 Vector Add Unit

tion logic is simplified because all VAX floating point formats (F, D, and G) are unpacked into an identical format. The unpacking involves decoding the sign, inserting the hidden bit, and rearranging the fraction bits. For all VAX floating point formats, the fractional part is expanded to 56 bits. (F_floating and G_floating are expanded with zeros on the right.) The fractional part is then surrounded on the right with two guard bits and a rounding bit to form a 59-bit fraction. The overflow and guard bits ensure the accuracy of rounded results.

After the elements are unpacked, the unpacking chips align the elements by taking the fractional part of the smaller magnitude number and shifting it to the right until its exponent is equal to that of the larger magnitude number. Each unpacking chip also receives the exponent bits of the other chip's element. Therefore, the alignment process can be done in parallel before the elements are sent to the addition logic that requires the alignment. If during the alignment of an element for a vector floating point subtract instruction, a one is shifted out of the 59-bit fraction field, then a "sticky bit" is generated. This sticky bit is used by the addition logic in the next cycle as a carry into the subtraction.

The unpacked, aligned elements are then sent to the add chip, which produces a result and then partially normalizes the result before sending it to the packing chip. Again, if the shifting during normalization shifts a one out of the fraction field, a sticky bit is generated. Finally the partially normalized result and the second sticky bit are sent to the packing chip which completes the normalization and rounding and adjusts the exponent field accordingly. To save an extra cycle, the packing chip computes two exponents values, one for each value of the carry-over in the rounding process. Final selection of the exponent and its exception is done using the actual carry-over of the rounding logic. The proper exponent and the normalized fraction are then rearranged into the appropriate floating point format, and the assembled element is sent to the vector register unit.

Integer and Logical Instructions For vector integer and logical instructions, the elements bypass the alignment logic and are sent to the add chip (VFAD) for all but the logical shift right instruction (VVSLRL and VSSLRL). For logical shift right instructions, the alignment logic does the shifting because the shifting circuitry is already needed for the alignment of fractions in floating point elements. The exponent unpacking logic is used to pass on the logical shift right count to the alignment logic, which then sends the shifted result to the add chip. The add chip operates on the low-order 32 bits of these elements and passes through the high-order 32 bits unchanged to the packing chip. For logical shift-left instructions (VVSLLL and VSSLLL), the low-order 32 bits also pass through the add chip unchanged.

On the packing chip, the floating point normalize logic performs to do logical shift-left operations. The shift count is passed to the normalize logic from the unpacking logic during the first cycle. For all other integer and logical instructions, the normalize count is forced to zero to pass the add chip result through. Finally, just before sending the result to the vector register unit, the packing chip checks for integer overflow exceptions.

Merge Instructions For vector merge instructions (VVMERGE and VSMERGE), the unpacking chip with the masked-out element, based on the appropriate vector mask register bit, zeros that element out before sending it to the addition logic. The addition logic adds the zero to the other element, which has the effect of passing the value of the other element on to the packing chip.

Vector Memory Operation

Because vector applications tend to issue many vector memory instructions, the execution time of these instructions is a critical factor in the performance of a vector processor. Therefore, the V-box was designed to minimize the execution time by taking advantage of the VAX 9000 CPU's large 128KB data cache, by prefetching vector data, and by fetching it in blocks instead of element by element.

Memory requests by the V-box are sent through the VAX 9000 CPU to the cache and address translation hardware (M-box) of the VAX 9000 CPU. The M-box translates the 32-bit virtual addresses for vector data into physical addresses and accesses the proper locations in the data cache. The vector address-mask unit generates the virtual addresses for the vector elements. For vector load and gather instructions, the vector data is returned to the V-box through the E-box, and written to the proper vector registers. The M-box returns 64 bits of data each cycle. For vector store and scatter instructions, the vector elements are sent through the E-box to the M-box. Although the vector register unit is capable of sending 64 bits at a time, the E-box need only forward 32 bits per cycle to the M-box. The M-box requires two cycles to write the cache and does not actually write the 64-bit data until the second cycle. (The first cycle performs the cache tag lookup.) Because the V-box implements synchronous memory management exception reporting, once a vector memory instruction begins execution, no other vector instruction may be issued until the memory instruction completes.

The VAX 9000 CPU prefetches vector data. This mechanism is used to move data from the main memory to cache in a manner which optimizes memory bandwidth. By using this method, a 25 percent improvement in the performance of vector load instructions is achieved. The prefetching starts when the scalar microcode on the VAX 9000 CPU checks the stride of a VLDQ instruction. If this stride is 8 bytes long (quadwords are contiguous in memory), the microcode converts the instruction into a block load instruction and sends it to the V-box. The block load instruction directs the V-box to issue a series of block load requests for vector data. A block load request moves an entire cache block from the memory into the vector registers. These blocks are loaded into both the cache and the vector registers when they come from main memory. (Bypassing the cached to load the vector registers directly reduces the effect of a cache miss for vector data.) Otherwise, the memory requests are done for one register element at a time.

In addition to converting the VLDQ to a block load instruction, the scalar microcode also issues prefetch requests to the M-box. The M-box determines if the data is valid in the cache. If so, no further action is taken on the request. If not, the data is requested from main memory. In this manner several prefetch requests are started in successive cycles. This method results in multiple memory banks being used in parallel. Vector data comes back to the cache at a rate of 500 megabytes (MB) per second. The microcode stops issuing prefetch requests when all the vector data has been requested. This ensures that the requests from the V-box do not encounter many cache misses.

Vector Address-Mask Unit

The vector address-mask unit performs the address generation and memory requests needed to execute the vector memory instructions VLD, VST, VSCAT, and VGATH. It also contains the vector mask register and support logic for masked instructions. Further, it contains the complete vector arithmetic exception register (VAER), which it updates based on the status sent by the vector add and vector multiply units.

For vector memory instructions, the vector address-mask unit receives the base (starting memory address of the vector) and stride (distance between vector elements in memory) of the instruction from the vector control unit in an indirect manner through the vector register unit. Both the base and stride are 32 bits long.

For most vector load and store instructions, the memory addresses for the vector data are generated in an iterative fashion. During the first cycle of execution, the base address bypasses the address adder and is immediately sent to the M-box to request the first element. Concurrently, the base and stride are added together by the address adder and latched to provide the address of the next element. In the next cycle, the latched address is sent to the M-box and to the address adder, where it is added to the stride to generate the next address. The process repeats until all element addresses have been issued. In tandem with the address generation, the vector control unit directs the vector register unit to send or receive the appropriate vector register element.

For vector gather and scatter instructions, the memory addresses for the vector data are also issued in an iterative fashion. During the first cycle of execution, the base address is sent to the vector address unit. In the second cycle, the vector control unit directs the vector register unit to send the first element of the offset vector to the vector address unit, which adds it to the base and latches the result. In the third and subsequent cycles, the resulting address is sent to the M-box while the base and next offset are added together. The process repeats until all element addresses have been issued. In tandem with the address generation, the vector control unit directs the vector register unit to send or receive the appropriate vector register element.

For masked vector load and gather instructions, addresses for all elements, masked and unmasked, are sent to the M-box. However, for masked-out elements, the request is modified from read to read no-op (i.e., do not actually perform the read). This process prevents the M-box from taking cache misses and address translation exceptions on masked-out elements. For masked-out elements, the M-box returns a dummy value to the V-box, which blocks the value from being written to the vector register unit. The vector address unit directs the control unit to block writes, based on the value of the appropriate vector mask register bit.

For masked vector store and scatter instructions, although both masked and unmasked elements are read from the vector register unit, masked-out elements are stopped from reaching the M-box. The vector address unit, based on the vector mask register, causes the E-box to discard the masked-out element instead of forwarding it to the M-box.

As described earlier, a VLDQ instruction with a stride of 8 bytes (unity stride) is converted by the VAX 9000 scalar processor into a block load instruction when sent to the V-box. The vector address unit, in turn, issues a number of block load requests, each of which is for 64 bytes of data, to the M-box with the appropriate address and selection bits. There are eight selection bits, one for each quadword in the block, which tell the M-box whether to return the corresponding quadword to the V-box for that block load request. Generation of these selection bits by the vector address unit is complicated because the starting address of a vector in memory is not aligned on a block boundary (i.e., starts within the middle of a block). The bits also depend on the vector mask register (for masked block loads).

To handle unaligned, masked block loads, the vector address unit must generate selection bits that deselect those quadwords which are not part of the vector but lie within the same blocks as the first and last elements of the vector. In addition, it must deselect those quadwords within the vector that are masked out by the vector mask register. Both of the above requirements are handled by using an extended version of the vector mask register to generate the selection bits. This process involves conceptually extending the vector mask register on both ends with enough selection bits so that each quadword has a corresponding selection bit. For example, a vector starting at the last quadword of one block requires that seven selection bits be added at the beginning of the vector mask register and one bit be added after the end.

Vector Multiply Unit

The vector multiply unit performs all of the vector multiply and vector divide operations defined by the VAX vector architecture: VVMUL, VSMUL, VVDIV, and VSDIV. The unit can perform either one multiply instruction or one divide instruction at a time, but cannot perform both types of instructions simultaneously. In addition, the unit performs exception checking and reporting, as required, including floating overflow, floating underflow, and divide by zero exceptions. The unit consists of four custom multipliers: a custom divider, a divide unpack chip, and two packing chips. Physically, these chips reside on the VML multichip unit of the VAX 9000 CPU. The custom multipliers and divider are identical to those used in the scalar execution engine (E-box).[8]

Multiplication By using four parallel multipliers, the pipeline latency through the multiplication logic for both single precision (integer and F_floating) and double precision (G_floating and D_floating) is only three cycles. Thus, the vector/scalar cross-over number for multiplication is quite small. As a result, the V-box achieves good performance for vector multiply instructions with small-sized vectors as well as large. As a double-precision vector multiply instruction executes, two 64-bit elements are received from the vector register unit each cycle and are latched in the four custom multipliers, each of which does a 32-bit by 32-bit multiplication.

As shown in Figure 9, the element bits are distributed in such a way that one multiplier operates on the high-order bits of both elements; one multiplier operates on the low-order bits of operand one and the high-order bits of operand two; one multiplier operates on the high-order bits of operand one and the low-order bits of operand two; and one multiplier operates on the low-order bits of both elements.

During the next clock cycle, each of the four multipliers unpacks its inputs and sends them through a large multiplication array, which produces one 64-bit partial product and latches the product. During the third cycle, the pack chips (VMLA and VMLB) add the four 64-bit partial products together to produce one result and prepare the result to be written back to the vector register unit. In this cycle, the four partial products are shifted according to their weight. Weight is determined in relation to which bits the multiplier used to produce a result. For example, the multiplier that operated on the high-order 32 bits (most significant bits) of both elements produces the most significant partial product bits, and the multiplier that operated on the low-order 32 bits (least significant bits) of both elements produces the least significant partial product bits. The partial products must be aligned or shifted properly before they are added together. Once the partial products have been added, the final product is then rounded, normalized, and packed into the appropriate VAX integer or floating point format before being written into the vector register unit in the next cycle.

The process and pipeline stages for single-precision multiplication (VVMULF and VSMULF) are

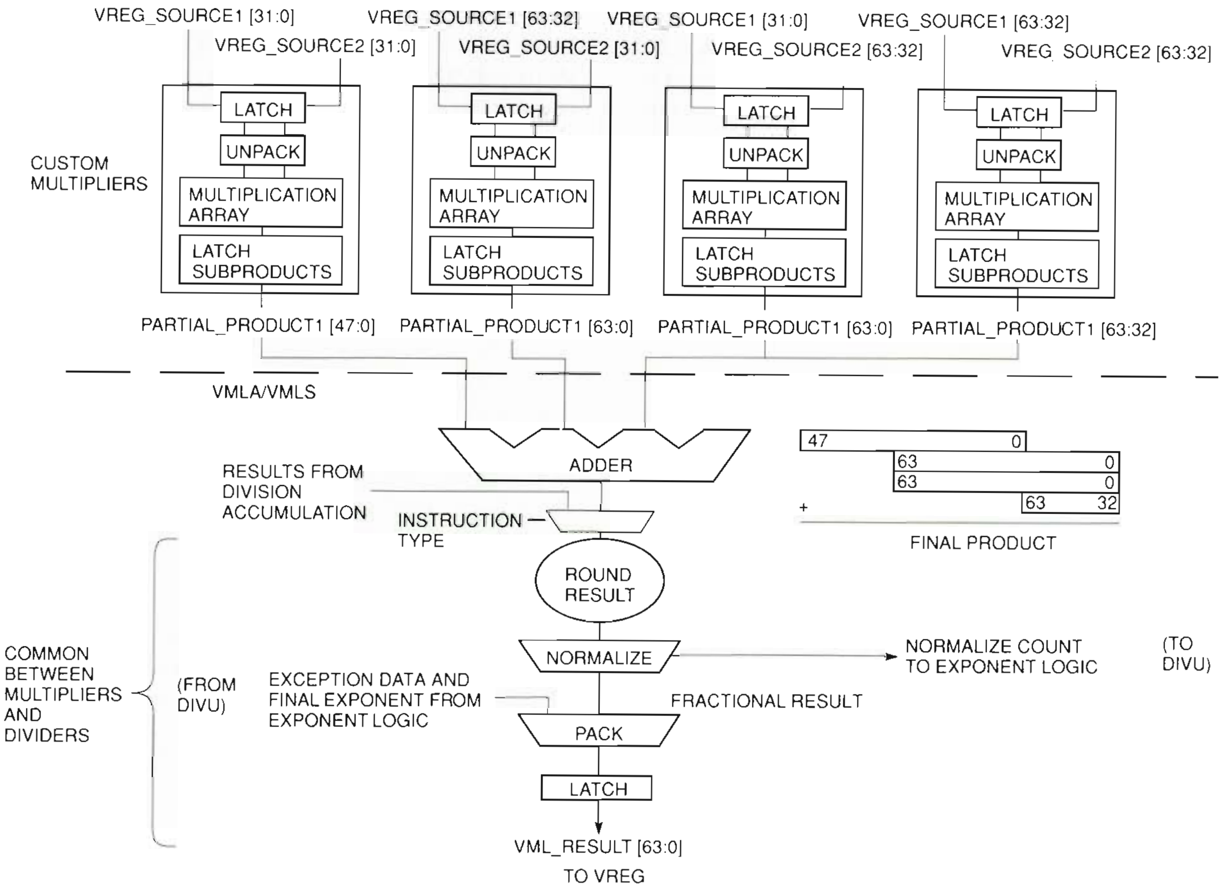


Figure 9 Vector Multiply Unit

similar to the process used for double-precision multiplication. However, in single-precision multiplication, only one multiplier chip is needed to produce the result and the pack chips do not need to sum the partial product. Integer multiplication is slightly different from floating point multiplication because it does not need to be accumulated or rounded. Thus, the correct product is produced by one multiplier. The result bypasses the accumulation and rounding logic and proceeds directly into the packing logic to be sent to the vector register unit.

The exponent handling for both multiplication and division is performed by the same logic on the packing chips. Depending on the instruction being executed, the exponent is either added (multiplication) or subtracted (division). The result of this operation is then piped to the next stage and the position of the hidden bit is determined. If the fractional portion of the data must be shifted to ensure the hidden bit is in the correct position, the exponent is then incremented or decremented accordingly. The normalize count (i.e., shift count) is used to select the correct final exponent. Overflow and underflow exception checking can only be detected and reported after the final exponent is selected. If an exception is detected, then a reserved operand is written to the appropriate vector register element. The first stage of the exponent logic also checks for divide by zero and reserved operand exceptions.

Division Vector division is a variable-cycle function. The number of cycles depends on the format of the operands. The custom divider is capable of producing six quotient bits per cycle. Therefore, F_floating point division is performed in 7 cycles, G_floating point in 12 cycles, and D_floating point in 13 cycles. Because of the variable number of cycles in a divide instruction, no other instruction can execute in the V-box while a divide is in process. Also, because of the iterative nature of division (i.e., one division must be completed before another can be started), the instruction cannot be pipelined.

As a vector divide instruction executes, two 64-bit elements are received from the vector register unit each cycle and are latched in the divide unpack chip. The elements are unpacked, and the fractional portion of the elements is sent to the custom divider in 32-bit slices. The exponent portion is sent to the shared exponent logic on the packing chips, as described in the Multiplication section. During this cycle, time-critical values, such as complemented element values and first-cycle quotient bits, are calculated and forwarded to the custom divider.

When the divider receives the data, it uses an iterative algorithm to produce six quotient bits per cycle. The quotient bits produced are then sent to the packing chips, which may have to increment the quotient, depending on the value of subsequent quotient bits. The divider instructs the quotient accumulation logic whether or not incrementing is necessary. The partial quotient, once decided, is held in a bank of latches until all the quotient bits are received. When the entire quotient is available, the result is rounded, normalized, and packed by using the same logic path as multiplication. A multiplexer switches this packing logic between the multiplication and division logic.

Performance Characteristics

As of this writing, testing of the vector performance of the VAX 9000 system has only just begun. However, some *preliminary* results are presented in Table 3. We expect that these results will improve as testing continues and more code is optimized to take advantage of the chaining and overlapping provided by the V-box.

Chaining and Overlapping

Because of the design of the vector register unit, the V-box can concurrently execute a vector add-class instruction, vector multiply instruction, and vector memory instruction. Unlike the VAX 6000 Model 400 system, vector register conflicts between these instructions have little effect on overlapping.[4] With the VAX 9000 system, a conflict only delays the execution of the subsequent vector instruction by one or two cycles at most.

However, the overlapping behavior of the V-box is sensitive to the issue order of vector instructions. If two vector instructions executed by the same V-box unit are issued one after the other, the second instruction is delayed until the V-box unit has finished executing the first. In addition, vector instructions issued after a vector memory instruction or divide instruction, do not begin execution until the previous instruction completes. A general rule in scheduling code for the VAX 9000 V-box, is to generate, whenever possible, instruction triples, where the first two instructions are a vector add-class and vector multiply instruction and the last instruction is a vector memory or vector divide instruction. Failing that, at least one vector add-class or vector multiply instruction should be issued before a vector memory or vector divide instruction.

The following code examples demonstrate the usage of the VAX vector instruction set and the overlapping behavior of the VAX 9000 V-box. (Note: It should be assumed in the examples that all arrays are 8-byte double precision.)

In the following DAXPY inner loop example, the first two VLDQ instructions do not overlap. However, the VSMULD, VVADDD, and VSTQ instructions do overlap.

```
Do i = 1,64
   DY(i) = DY(i) + DA x DX(i)
enddo
```

vectorizes as:

```
VLDQ      DX, #8, V0  ;Load vector DX
VLDQ/M    DY, #8, V2  ;Load vector DY
                      ;with modify intent
VSMULD    DA, V0, V1  ;V1 = DA*DX
VVADDD    V1, V2, V3  ;V3= V1+DY
VSTQ      V3, DY, #8  ;Store vector DY
```

The first two VLDQ instructions do not overlap in the following MERGE example,

```
Do i = 1, 64
   a(i) = b(i) - c(i)
   if (a(i) .gt. 0) then
      b(i) = a(i)
   else
      b(i) = c(i)
   endif
enddo
```

Table 3 VAX 9000 Model 210 Preliminary Performance Double-precision MFLOPS, Uniprocessor

	Size	Vector
Peak rate	NA	125
LFK (Geometric mean)	441	13.2
LFK (Arithmetic average)	441	20.6
LINPACK	1000^2	80
FFT	4096	26
Convolution	150 x 1500	99.15
Matrix multiply	64^2	111.36

vectorizes as:

```
VLDQ       b, #8, V0   ;Load vector b
VLDQ       c, #8, V1   ;Load vector c
VVSUBD     V0, V1, V2  ;b-c
VSTQ       V2, a, #8   ;Store vector a
VSLSSD     #^X0, V2    ;Test a(*) and set mask
                       ;in VMR. (VSCMP
                       ;pseudo-op doing Less
                       ;Than Signed test)
VVMERGE    V1, V2, V0  ;Merge a and c into b
                       ;using mask in VMR
VSTQ       V0, b, #8   ;Store vector b
```

However, the VVSUBD instruction does overlap with the VSTQ instruction. Both the VSLSSD (VSCMP) and VVMERGE instructions are executed by the vector add unit. Therefore, these two instructions do not overlap. However, the VVMERGE instruction does overlap with the VSTQ instruction.

In an IF-THEN-ELSE example, such as the following,

```
Do i = 1, 64
   if (a(i) .gt. 0) then
       b(i) = c(i)
   else
       b(i) = c(i) / a(i)
   endif
enddo
```

vectorizes as:

```
VLDQ       a, #8, V0   ;Load vector a
VSLSSD     #^X0, V0    ;Test a(*) and set mask
                       ;in VMR. (VSCMP
                       ;pseudo-op doing Less
                       ;Than Signed test)
VLDQ       c, #8, V1   ;Load vector c
VVDIVD/0   V1, V0, V2  ;Masked divide of c by a
                       ;for VMR[i] = 0
VSTQ/1     V1, b, #8   ;Store "then" part of b(*)
VSTQ/0     V2, b, #8   ;Store "else" part of b(*)
```

Nothing overlaps the first VLDQ instruction, but the VSLSSD instruction does overlap the second VLDQ instruction. Nothing can overlap with the VVDIVD instruction. Thus, the VSTQ instruction does not begin execution until the VVDIVD instruction completes. The remaining VSTQ instruction waits for the first VSTQ instruction to complete.

In the following scatter-gather example, none of the instructions is overlapped.

```
Do i = 1, 64
   if (a(i) .eq. 0) then
      b(i) = c(i)/d(i)
   endif
enddo
```

vectorizes as:

```
VLDQ       a, #8, V0   ;Load vector a
VSEQLD     #^X0 ,V0    ;Test a(*) for zero and
                       ;set mask (VSCMP pseudo-
                       ;op doing Equal test)
IOTA       #8, V1      ;Make compressed
                       ;vector of offsets
                       ;write size of vector
                       ;to VCR
MFVCR      R0          ;Move VCR into R0
                       ;(MFVP pseudo-op)
MTVLR      R0          ;Load new VLR value
                       ;(MTVP pseudo-op)
VGATHQ     c, V1, V2   ;Gather vector c
                       ;using offsets in V1
VGATHQ     d, V1, V3   ;Gather vector d
                       ;using offsets in V1
VVDIVD     V2, V3, V4  ;Divide c by d
VSCATQ     V4, b, V1   ;Scatter vector b using
                       ;offsets in V1
```

It should be noted in this example that the VSEQLD and the IOTA instructions do not overlap. This lack of overlap occurs because the IOTA instruction is actually done with microcode on the E-box, and the IOTA instruction cannot begin execution until the VSEQLD instruction has computed all the new vector mask register bits. The vector register access instructions (MFVCR and MTVLR) take only a few cycles and do not significantly affect the overlapping of other vector instructions.

Summary

By taking advantage of key features of the VAX vector architecture, such as instruction overlapping, imprecise exceptions, and asynchronous interaction with the scalar processor, the vector processor of the VAX 9000 system provides supercomputing performance for computationally intensive applications. Through the use of barber poling, the vector processor can overlap two vector arithmetic instructions with one memory instruction to deliver a peak double-precision performance of 125 MFLOPS.

Acknowledgments

The authors wish to acknowledge the technical contributions of the following individuals to the VAX vector architecture and the VAX 9000 V-box design: Wayne Cardoza, Dave Cutler, Tryggve Fossum, Rich Grove, Kevin Harris, Steve Hobbs, Brian Koblenz, Dwight Manley, Dave Orbits, Bob Supnik, Mike Tehranian, Cheryl Wiecek, and Rich Witek.

References

1. Russell, "The CRAY-1 Computer System," *ACM Proceedings,* vol. 21, no. 1 (January 1978): 63–72.
2. *VAX Vector Processing Handbook* (Maynard: Digital Equipment Corporation, Order No. EC-H0419-46/89, 1989).
3. R. Brunner, *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-F576E-DP, 1990).
4. D. Fenwick et al., "A VLSI Implementation of the VAX Vector Architecture," *Proceedings of COMPCON '90* (IEEE, Spring 1990).
5. *CRAY-2 Computer System Functional Description* (Cray Research, Inc., 1985).
6. W. Buchholz, "The IBM System/370 Vector Architecture," *IBM Systems Journal,* vol. 25, no. 1 (1986): 51–62.
7. D. Marshall and J. McElroy, "VAX 9000 Packaging—The Multichip Unit," *Proceedings of COMPCON '90* (IEEE, Spring 1990).
8. M. Adiletta et al., "Semiconductor Technology in a High-performance VAX System," *Digital Technical Journal,* vol. 2, no. 4 (Fall 1990, this issue): 43–60.

Peter B. Dunbeck
Richard J. Dischler
James B. McElroy
Frank J. Swiatowiec

HDSC and Multichip Unit Design and Manufacture

The VAX 9000 system effectively integrates state-of-the-art packaging and interconnects with advanced integrated circuits to achieve a short machine cycle time (16 nanoseconds) and a high rate of instruction execution. To meet high-frequency electrical signal and pin count requirements for the system, engineers chose tape automated bonding technology and consequently conceived and developed the high-density signal carrier (HDSC). The HDSC offers densities three to five times greater than conventional printed circuit boards. This unique technology is manufactured using semiconductor and advanced printed circuit board techniques. The HDSC is at the heart of the multichip unit, a high-performance logic module, with which the VAX 9000 CPUs and system control unit are constructed.

Over the past decade, advances in the performance of integrated circuits (ICs) have outpaced advances in packaging and interconnect technologies. Thus a high-performance mainframe with conventionally packaged bipolar integrated circuits would experience interconnect delays that account for more than 50 percent of the system cycle time. Key to optimizing high-end mainframe performance, then, is the effective integration of state-of-the-art packaging and interconnects with advanced integrated circuits. The high-density signal carrier (HDSC) and the multichip unit (MCU) are proprietary technologies that shrink interconnect paths and thus reduce the distance and electrical loading of signals between chips. These technologies use conventional semiconductor and printed circuit board (PCB) equipment in many areas of manufacturing to improve reliability at a competitive cost. The result is shorter machine cycle time and higher instruction execution rate. The VAX 9000 CPUs and system control unit (SCU) are constructed entirely of multichip units on large planar modules. The SCU is composed of arrays of 6 multichip units, and the CPUs are composed of arrays of 16.

Multichip Unit Design Goals

Beginning at the concept level and throughout the development and test phase, signal integrity considerations guided the development of the HDSC and the multichip unit. Designers had to ensure that the fast signals would not be disturbed by noise. The cycle time goal for the VAX 9000 system, 16 nanoseconds (ns), allows the system to operate at 30 VAX units of performance (VUPs).

To transmit electrical signals quickly between chips, wiring paths must have controlled ratios of wire size to distance from voltage planes. These impedance-controlled paths allow radio-frequency computer signals to propagate with minimal distortion. Prevention of noise on the signals is paramount and many details of the physical implementation, including spacings between wires, are critical to ensuring signal integrity.

To meet the cycle time goal, high-frequency electrical signal concerns needed to be considered in the design, concerns that would have been negligible for slower speed signals. Due to the physics of electrical fields, as electrical signals switch at high frequencies, they succeed in holding their shape (data) only if they are fed power extremely quickly, and if they are given short paths of uniform properties on which to travel. Due to the amount of power and the short amount of time a signal is given to arrive on chip, conventional chip carrier packages were disallowed for the VAX 9000 system. The signal paths had to be very short to be virtually noiseless. To achieve this objective, engineers decided to enhance tape automated bonding (TAB) technology with a ground plane for electrical control of the wire impedances (paths). This reduction in chip package size also allowed all of the chips for the system to be packaged into a tight area. Consequently, to fit wires between chips, extremely dense HDSC technology was conceived and developed.

The multichip unit also required careful thermal design attention because each chip consumes up to 30 watts. Moreover, most multichip units contain four to eight of these chips plus self-timed RAMs (STRAMs). The key to success for the VAX 9000 program was balancing the trade-offs between performance requirements and technology development risks.

To meet the electrical and density requirements for the machine, engineers specified the following for the multichip unit:

1. Series-terminated output drivers were required on chip. Therefore, external resistors are not needed on the multichip units or programmed into the design elsewhere. These external resistors take up space and lower reliability.
2. TAB was specified for manufacturing reasons. Short TAB tape was required to reduce switching noise on chips. Noise would have been generated if the TAB wires were longer. In the case of the noisiest chips, a ground plane was added to the tape to reduce noise.
3. HDSC etch had to be two routing layers of 18-micron by 9-micron wires on 75-micron centers to meet the density, resistivity, crosstalk, impedance and other goals.
4. Four power planes, each one powered from two sides, were required to distribute three voltage rails with acceptably high conductivity.
5. Thin dielectric separates the power planes and produces high capacitance which filters noise and improves performance. This capacitance eliminates the need for discrete parts which consume valuable space and lower reliability.
6. Impedance control of the connectors on the multichip unit was needed to prevent signal disturbance. Rules were generated for the number of ground pins.

The heart of the multichip unit is the HDSC. The HDSC is an interconnect technology consisting of nine metal layers separated by polyimide dielectric and mounted on a copper baseplate. The top metal layer is a pad layer used to solder-attach all of the integrated circuits and connectors. The four metal layers below make up the signal core. The signal core is a controlled-impedance, dual buried stripline interconnect system used to wire all integrated circuits to each other and to the connectors. The power is brought from the perimeter of the HDSC to the integrated circuits through the bottom four metal layers.

All integrated circuits in the multichip unit are attached to the HDSC by a tape automated bonding (TAB) process. The VAX 9000 system uses four types of chips, all of which have emitter coupled logic (ECL): gate arrays, custom chips, and two types of STRAMs. At each chip site, a cutout in the HDSC allows the chip to directly attach to the baseplate. The signals on and off the multichip unit are carried by four signal flex connectors which attach to the perimeter of the HDSC. The signal flex connector provides a separable interface to the planar board and extends the controlled-impedance electrical environment of the HDSC. Power is brought through two power connectors attached to opposite sides of the HDSC. The signal flexes, the power connectors, and the baseplate are attached to the multichip unit housing. The housing provides the structure for the multichip unit and holds the components needed to position and wipe the signal flex. The chips and HDSC surface are covered by a plastic lid.

The high-powered chips are efficiently cooled by a short conductive path through the back of each chip. The thermal power is conducted from the chip to the baseplate and into a pin fin heat sink over which air is impinged to remove the heat.

The following sections describe the implementation of the technology.

The HDSC Design and Manufacturing Process

The goal for the HDSC project was to produce a high-density, high-performance, manufacturable printed circuit board. This goal was achieved. The density of the HDSC is three to five times greater than that of conventional printed circuit boards. Even at this density, the HDSC maintains the signal integrity of bipolar integrated circuits with edge speeds of 200 picoseconds. This section describes how the manufacture of the HDSC pushes the limits of printed circuit board and semiconductor equipment into new types of applications. We also address the integration of computer-aided (CAD) tools, process controls, and test feedback, which helped us to achieve the results we sought.

HDSC Technology

As noted earlier, the HDSC has nine copper layers for power and signal distribution. The insulating material, polyimide, has a low dielectric constant of 3.5 as compared with oxide or nitrides used in integrated circuits or as compared with ceramic, which is used for hybrid circuits. The interconnect is laminated to a copper baseplate to provide mechanical structure as well as attachment of the multichip unit heat sink.

The conducting layers consist of the following:

- Two layers for signal distribution
- Two layers that serve as signal reference planes
- Four layers for power distribution
- One layer with bonding pads to attach the TAB and connectors

The signal distribution is a single x–y pair that uses the reference planes to create a dual strip-line interconnect. This interconnect provides a controlled-impedance signal path with minimal crosstalk. Table 1 lists the electrical and physical design parameters of the HDSC.

Process Overview

The HDSC is manufactured by two types of processes: core processing and assembly processing. Figure 1 is a diagram of the HDSC process flow.

The core process, described further below, uses semiconductor manufacturing equipment and is similar to the manufacturing process for the back end of an integrated circuit. Two cores are manufactured: a signal core for strip-line signal interconnect, and a power core for the four planes (or layers) that distribute power throughout the finished HDSC.

The second process, assembly, uses advanced printed circuit board techniques to laminate and interconnect the signal core and power core. The completed HDSC has solder pads to accept the outer lead bond of TAB integrated circuits, signal flex, and power flex. The HDSC is tested with a custom flying probe tester. Tests are made to ensure the HDSC is functional and meets electrical parameters.

Table 1 HDSC Physical and Electrical Design Parameters

Line pitch	75 microns
Line width	18 microns
Line thickness	10 microns
Dielectric thickness	25 microns
Dielectric constant	3.5
Line impedance	60 ohms
Line resistance	1/0 ohm/centimeter
Crossover capacitance	3.6 femtofarads
Crosstalk	5.1 percent maximum
Propagation delay	66 picoseconds/ centimeter

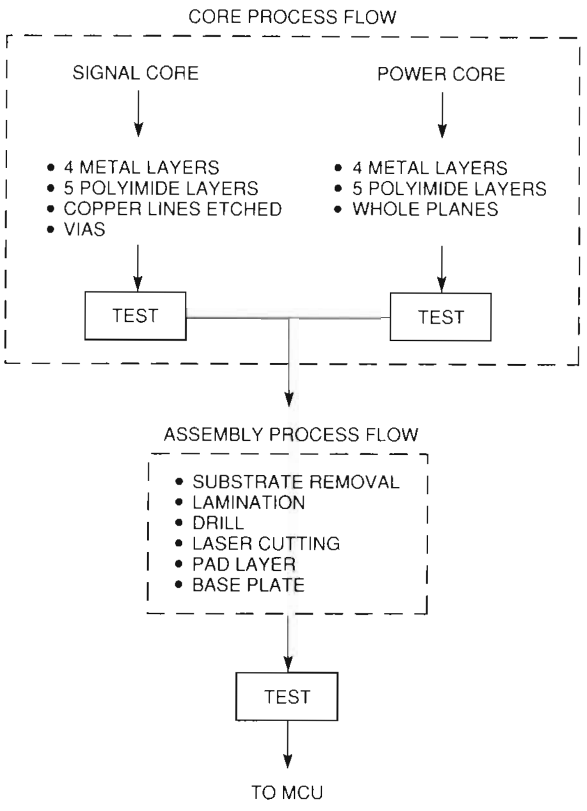


Figure 1 HDSC Core and Assembly Process Flow

Core Processing The process for the manufacture of the signal and power cores, or the core process, consists of alternating between copper deposition and polyimide coating until the completed interconnect layers are built on the metal wafer. The process is performed on a metal substrate shaped like a 6-inch semiconductor wafer. Copper layers are deposited by a combination of sputtering and plating techniques. Patterns in the copper that become signal traces are generated by a semiconductor photolithographic technique. First, a photoresist is applied to the metal wafer. The resist is then exposed to the pattern in the mask that is held by the semiconductor wafer aligner. This pattern is then developed in the resist and etched into the copper. The remaining copper thickness is then added by plating. Another resist pattern is developed over the plated signal traces to define where a copper connection between interconnect layers will occur. This connection is called a via post, and it is also formed by a plating process.

Polyimide is spun on to the wafers by integrated circuit photoresist spin tracks. The relatively thick polyimide (25 microns at signal layers) helps to planarize the surface of the wafers and also to cover the patterned copper lines and copper posts. Semiconductor photolithography equipment is also used to generate patterns in resist through which holes (extensions of the via post) are plasma-etched in the polyimide. These vias are filled at the next copper deposition to create a connection between patterned layers.

Both signal cores and power cores are electrically tested to ensure electrical functionality.

Assembly Processing To complete an HDSC, a signal core is matched to a power core. The metal wafer that acted as a substrate is then removed, and the signal and power cores are laminated together. Connections from the power core to the signal core are made by drilling and then plating up through the drilled holes. The plating and etching processes used to form the plated-through holes also produce copper pads on the bonding layer. Solder is screened and reflowed onto the bonding pads. Die site holes which provide openings for bonding the chips to the baseplate are cut through the laminated signal and power cores (HDSC decal). A laser cuts out the die sites and trims the HDSC decal to its final size. The assembly is complete when the interconnect is laminated to a baseplate which provides mechanical structure.

HDSC Test Process

The goal of the HDSC test process was to ensure that the physical technology met the VAX 9000 signal integrity requirements, discussed earlier in this paper. Equally important was to ensure that the technology was manufacturable and verifiable (measurable). Engineers had to accurately convert design information to masks and to verify HDSC electrical results by testing. We therefore developed modeling and measurement techniques to establish physical and electrical design rules; implemented CAD tools to verify these design rules; developed software to generate test vectors from the CAD database; and also developed production HDSC testers.

Modeling and Measurement Techniques To determine what trade-offs would be required between the VAX 9000 signal integrity and the HDSC physical manufacturing capabilities, engineers needed both modeling and empirical measurement techniques. A software tool based on Monte Carlo Analysis was developed that could drive a three-dimensional capacitance model. This tool predicts electrical parameter sensitivity to different physical processing variations. Early in the project, processing engineers estimated the expected manufacturing distribution of critical signal core dimensions. Once HDSCs were manufactured, actual processing distributions were fed into the model which predicted yield against the specification. Based on this, the processes were adjusted to maximize yield. Models and electrical results were verified by time domain reflectance (TDR) and resistance-inductance-capacitance (RLC) measurements.

The high-frequency measurements necessary to characterize the interconnect were extremely sensitive to probe card inductance and capacitance and probe contact resistance. Custom test fixtures were designed to perform the measurements.

High-frequency test measurements in the production environment are not practical, but we determined that resistance and capacitance testing could be used instead to verify the HDSC signal integrity. A production flying probe tester was developed to test HDSCs. Once again, probe parasitics were large compared to the type of measurements necessary. Custom probe design, calibration methods, and software to drive the tester again were necessary. From HDSC graphic design files, test capacitance limits for every signal net are generated, which ensures electrical functionality as well as signal integrity. In addition, the resistance of every plated-through hole is measured, the integrity of power planes verified, and resistance and leakage measurements performed on test structures within the HDSC.

CAD The VAX 9000 system design includes net lists and graphical files of the HDSC masking layers. To this data is added process control monitors, alignment marks, and pattern modifications required to meet the process design rules. After modifications, all data is verified by software that checks design rules and electrical rules.

The data is used in a variety of applications. First it is converted to pattern-generation format so that from this data masks can be written and an inspection file can be generated to verify the mask-making process. The information is also used to drive numerically controlled equipment, such as the drills and lasers that perform die site cut outs. Finally, it is used to create a net-capacitance test file which drives the HDSC production testers.

MCU Design and Manufacturing

The multichip unit (MCU) takes full advantage of the integrated circuit and HDSC technologies to produce a high-performance logic module. The major components of the multichip unit are shown in Figure 2. The components, their functions, and the assembly and test process are discussed in the next two sections. Table 2 summarizes the multichip unit specifications.

All units have certain features that are fixed, regardless of logic design. These include the clock distribution chip, serialization pattern, signal connector, power connector, housing and heat sink. The VAX 9000 system uses 20 unique logic design implementations, or options. The multichip unit features that make an option unique are the gate arrays (up to 8), the STRAMs (9 replace 1 gate array, 24 replace 3 gate arrays), and the HDSC.

TAB Semiconductors

All semiconductors in the multichip units are integrated circuits. Discrete devices and passives which consume more space and display lower reliability are not used. TAB is a chip-to-substrate interconnect made of layers of copper and polyimide film. The copper signal lines are patterned to mate with gold bumps on the IC perimeter and with solder-plated pads on the HDSC.

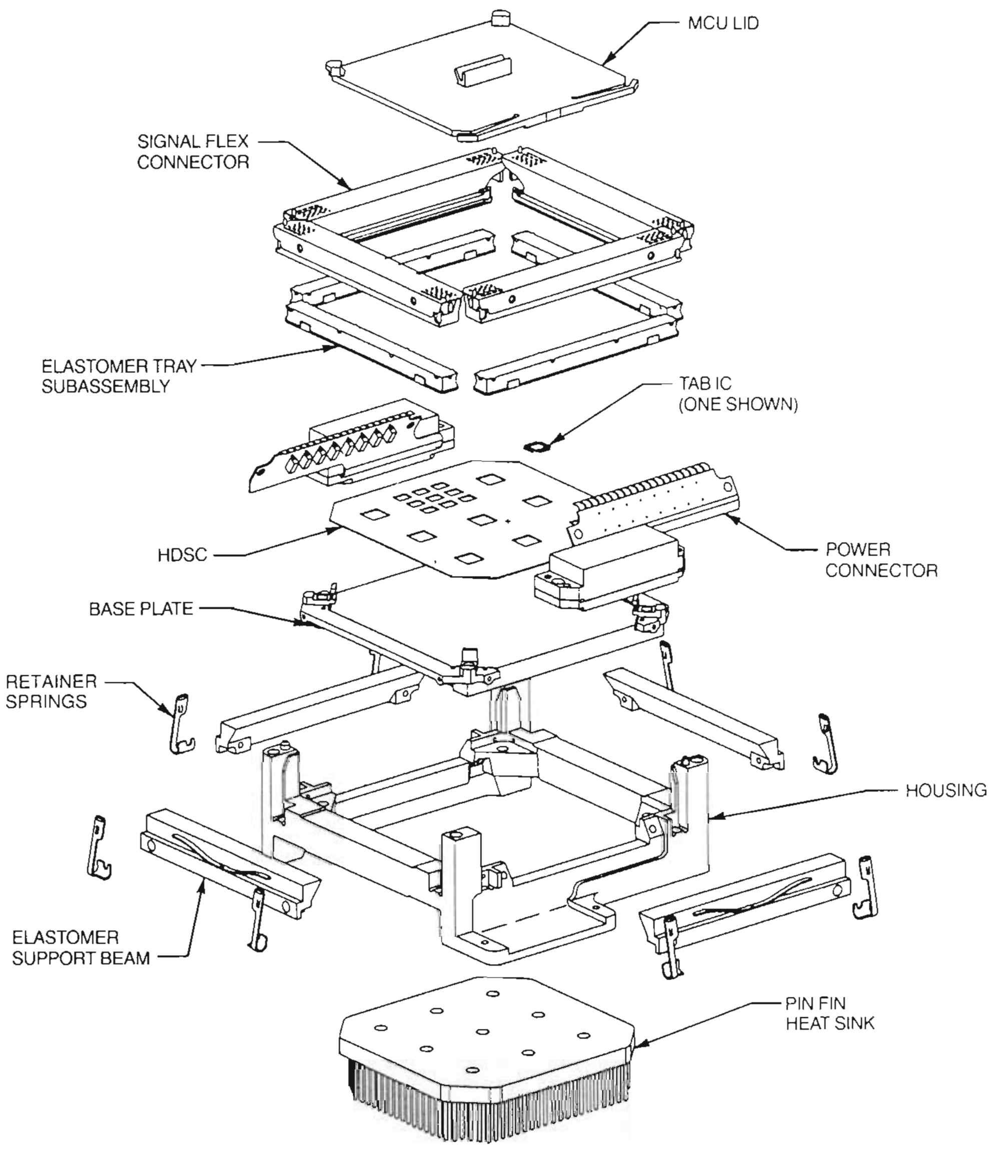


Figure 2 Exploded View of an MCU

Table 2 Summary of MCU Specifications

Maximum power dissipation	270 watts (air cooled)
Maximum IC junction temperature	85 degrees Celsius @ 25 degrees Celsius room temperature
Maximum number of VLSI chips	72
Minimum chip lead pitch	200 microns
Size	
Plane	14.12×13.21 centimeters
Height	5.44 centimeters
Minimum pitch on planar module	14.38×13.46 centimeters
Weight	1.59 kilograms (with heat sink)
Clock input frequency	320 to 580 megahertz
Signal I/O per MCU	800
Signal rise time	600 picoseconds
Voltage levels	2 plus ground
Maximum current	40 amperes per voltage level

The TAB for the gate array and most of the custom chips is a two-metal-layer tape with 360 leads. A cross section of the gate array TAB is shown in Figure 3. The dielectric layers are polyimide film. One metal layer contains the etch lines used for both power and signal I/O and the leads to bond to the chip and HDSC. The other layer is a reference plane to establish controlled impedance and to minimize the inductance of the power and ground paths. The reference plane is connected to the ground leads by vias etched through the Kapton and plated up. The gate array power is brought in through 104 power (two voltage levels) and ground leads. All of the signal leads are 60 ohms controlled impedance. The leads are 35 microns thick and copper coated with about 0.5 micron of electroless tin.

As shown in Figure 3, the TAB has an inner lead bond (ILB) pitch of 100 microns and an outer lead bond (OLB) pitch of 200 microns. The total span of the TAB when mounted is 2.4 centimeters. The leads are formed near the OLB to provide strain relief for the ILB and to protect the OLB from thermal stress. To minimize propagation time and noise, the length of the signal lines has been minimized by keeping the OLB pitch to the minimum compatible with manufacturing processes. The bond at the IC (at the ILB) is a gold–tin eutectic formed by gang thermal compression bonding.

The clock distribution custom chip (CDxx) and the STRAM use single-metal-layer tape with polyimide dielectric. The CDxx has 252 total leads with 84 power and ground leads. The CDxx has the same pin pitch as the gate array. The VAX 9000 system uses two sizes of STRAMs (1K by 4 bits and 4K by 4 bits) that have TAB tapes of different sizes. The STRAMs also use a single-metal-layer tape with 48 leads. The minimum ILB pitch is 250 microns and the minimum OLB pitch is 450 microns. Single-metal-layer tape was selected for these devices because it was less expensive than two-metal-layer, and two-metal-layer tape was not needed because of the shortened lead lengths on the STRAMs. Single-metal-layer tape was acceptable for the CDxx chip because all the outputs are differential and synchronous. Noise cancellation was guaranteed.

All devices that use TAB are shipped in a 35-millimeter slide carrier. The devices are encapsulated in epoxy to minimize infiltration of moisture or corrosive ions and to reduce damage due to handling. The back sides of the chips are bare silicon because

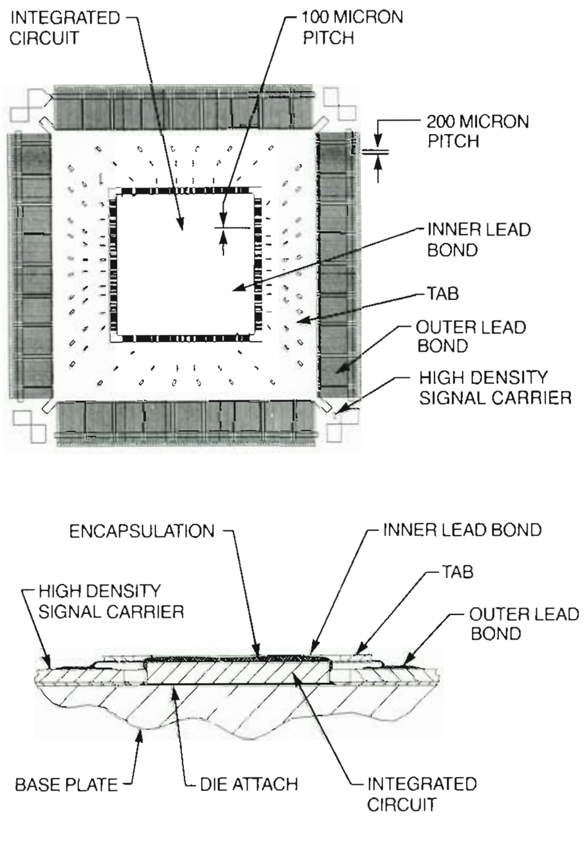


Figure 3 Isometric of a Gate Array Showing Features of the TAB

no plating is required for epoxy die attach. The epoxy die attach is filled with microscopic particles to enhance the thermal conductivity while maintaining electrical isolation between chips.

Signal Flex Connector

The signal flex connector is a high-density, controlled-impedance connector used to transmit signals between the HDSCs and the planar module. Each multichip unit has four flex connectors with a combined signal I/O of 800 in an area less than 40 square centimeters. Figure 4 shows a cross section of one signal flex connector. The body of the connector is a two-metal-layer flex print with 50- and 60-ohm signal lines. The ground plane in the flex circuit is used as an AC return path. No power is carried through the signal flex. The signal plane contains 200 etch lines with a raised gold bump on each at the planar module interface. The connection to the HDSC is a solder bond similar to the solder bonds for the TAB device. A window is opened through the polyimide to allow the formation of cantilevered, exposed, solder-plated leads.

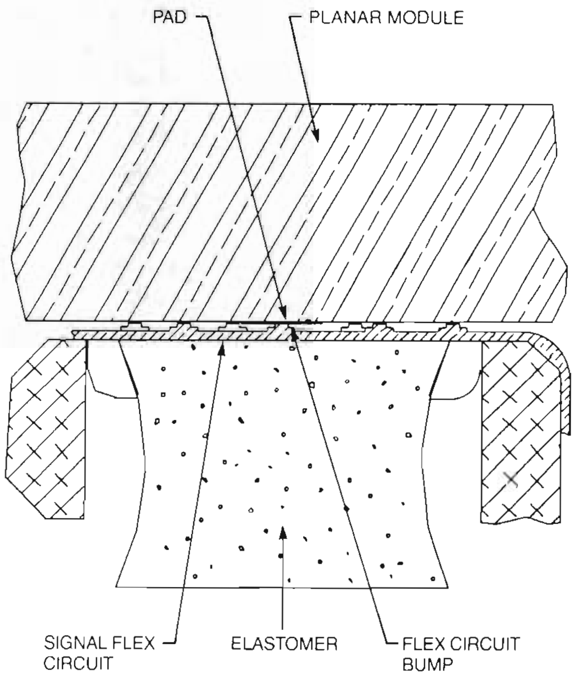
The raised bump on the flex circuit concentrates the contact force into a small area. The bump is solid copper that is plated over with nickel and hard gold. The force on the bump is generated by compressing a molded silicone rubber elastomer. The compression of the connector causes the flex frame to engage a cam on the housing and wipe the contacts across the planar module pads. The connector is compressed, nominally, 1.27 mm and wipes 0.46 mm. The bottom of the elastomer mates with a tray which has a contoured surface to vary the compression along the length of the elastomer. This contoured surface improves the uniformity of the force that the bumps exert on their pads. The connector has been designed to generate 100 grams minimum load on all bumps. The wipe action and the bump force of the connector minimize the effect of dust and environmental films on the mating surfaces.

Power Connector

The power consumed by the multichip unit is brought in through two power connectors mounted on opposite sides of the HDSC. The connector is composed of a flex circuit, a connector, and decoupling capacitors. The flex circuit is solder bonded to large pads on the HDSC surface. The flex has three copper conductive planes separated by polyimide dielectric. The connector has stamped metal contacts soldered into the flex circuit and assembled into a plastic housing. The connector plugs into flat blades on the bus bar of the planar module assembly. The decoupling capacitors on the power flex circuit filter the medium-frequency switching noise on the MCU and the MCU power bus.

Thermal Design

The multichip unit was designed from conception to provide an efficient cooling path for the integrated circuits. Figure 5 shows a cross section of the



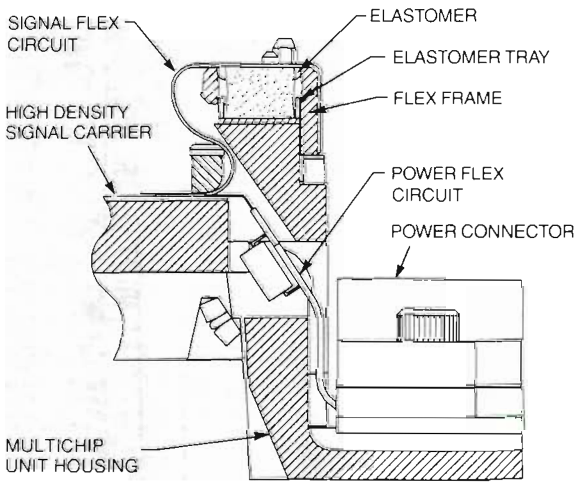


Figure 4 Signal Flex Connector with Detail of Bump

multichip unit. The heat dissipated by the chips is conducted through the silicon and the die attach into the baseplate. As mentioned above, the die attach is an epoxy heavily filled with microscopic diamond particles to increase thermal conductivity. The heat spreads out in the copper alloy baseplate and is conducted across a dry interface to an aluminum base of the pin fin heat sink. The heat sink has 600 aluminum pins, each 0.20 centimeters in diameter, pressed into the base. Air plenums in the cabinets direct at least 14.6 liters per second of air into each multichip unit heat sink. The thermal resistance for a 30-watt gate array is less than 2.0 watts per degree Celsius which gives a junction temperature of 85 degrees Celsius with room air at 25 degrees Celsius. This low junction temperature is a critical part of the high reliability of the multichip unit.

Clock Distribution

The system clock on the VAX 9000 system is distributed to each of the multichip unit clock distribution chips (CDxx). The CDxx generates 40 differential outputs which are routed through equal-length etch to the other chips. The CDxx also distributes and controls the scan lines that test the unit both in manufacturing and in the field. The scan lines also allow the unit serial number and revision status to be read by the system console.

Multichip Unit Manufacturing

Figure 6 shows the manufacturing process flow, which has three major work centers:

- 54-class assembly and inspection
- P1000 assembly and inspection
- Test and diagnose

In the 54-class process, TAB semiconductor devices are assembled to the HDSC substrate, resulting in the subassembly known internally as a 54-class module. In the P1000 process, connector and housing components are assembled. At the last major center, the test process, final units are tested and, if necessary, diagnosed. A shop floor control system tracks the units through the line and provides critical component and process trace information. In addition, this control system is used to monitor process parameters to ensure control of the line and consistent product quality.

The following section provides insight into several of the process technologies we used to meet the manufacturing goals of the VAX 9000 system.

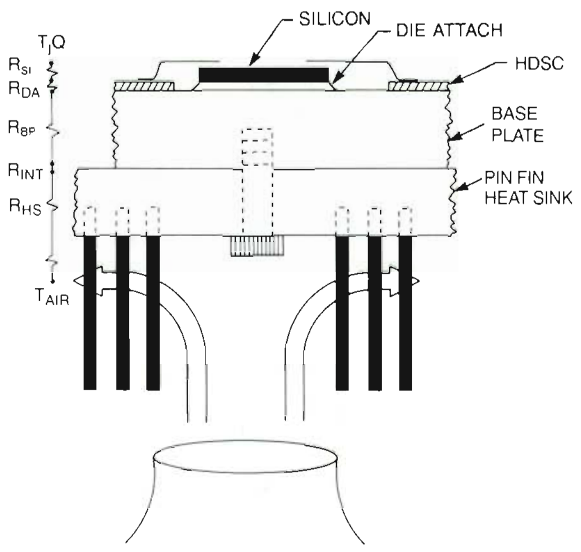


Figure 5 Thermal Path

TAB and Flex Circuit Bonding

The insertion and soldering of leads is the most critical step in the multichip unit manufacturing process. Single-lead and multiple-lead gang bonding approaches were both considered. Gang reflow soldering is an effective way to achieve repeatable, reliable connections for both the TAB semiconductors and the signal flex circuits. Early development work on manual machines required operator action for lead forming, lead alignment, and gang bonding. Today, critical process parameters — time, pressure, temperature — are computer controlled to specified values, and the process uses tools to assist the operator in material movement and vision systems to improve alignment of leads. Before bonding, the leads are covered with a low activation flux which is removed later in the process.

Die Attach

Another critical manufacturing step is the die attach process. The excellent thermal performance of the multichip unit is achieved by following these steps:

- Careful control of the die attach materials with feedback to our suppliers.
- Surface cleanliness specified and also managed with our suppliers.
- Dispensing of epoxy. The filled epoxy is dispensed by an x–y table that is computer controlled to supply the correct pattern for the particular multichip unit type.

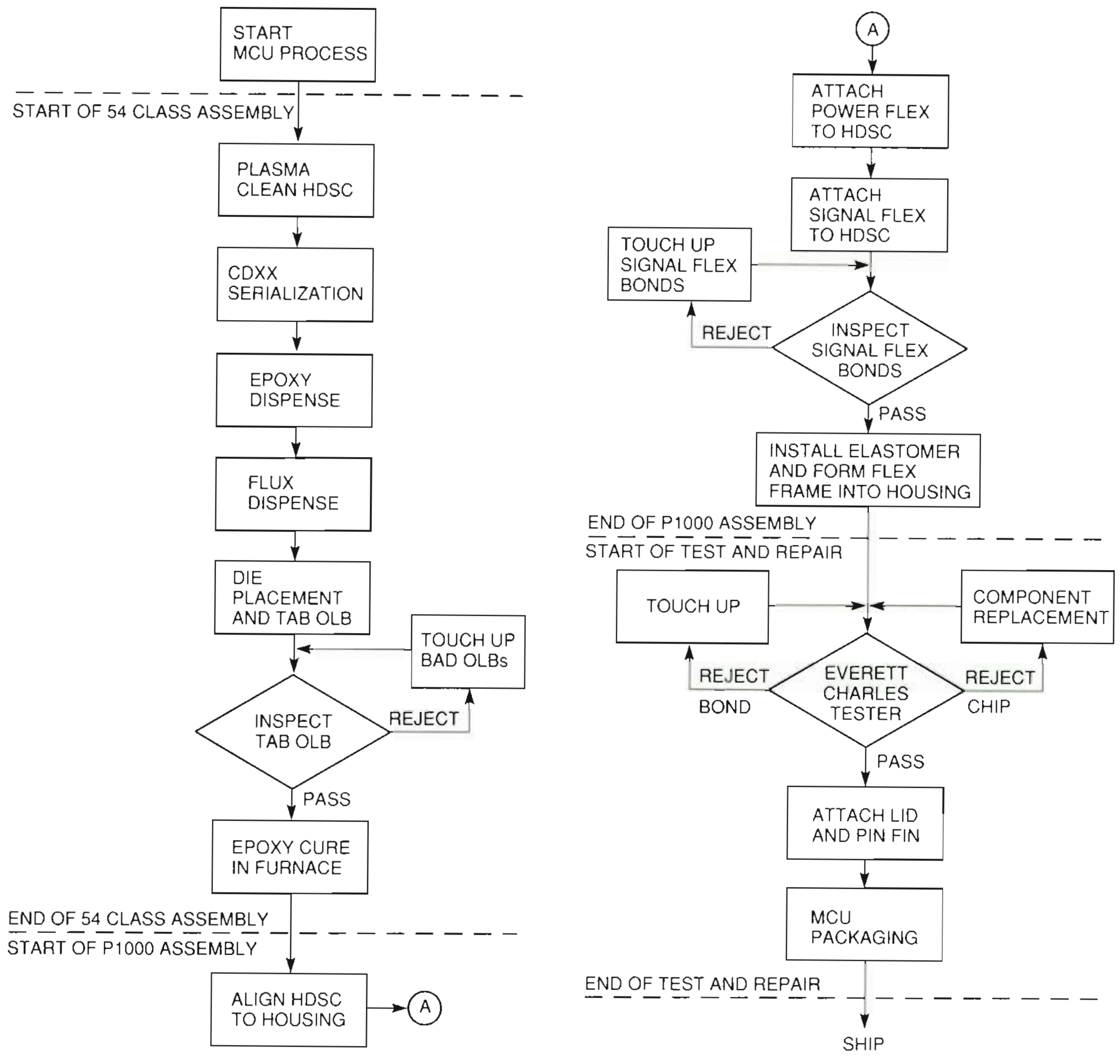


Figure 6 Manufacturing Process Flow

- Establishment of bond line thickness and epoxy cure. Bond line thickness is accomplished by mechanically applying pressure while curing in a purged belt furnace.

Inspection

To ensure that all soldered leads are reliably bonded, leads must be inspected for shorts, misalignments, opens, and weak joints. Shorts and misalignments are discovered by an automated vision system that calls marginal points to the operator's attention. The operator can then determine if repair action is warranted. Inspection for opens and weak joints is done by striking the leads with a pulse of laser energy and then measuring the thermal decay profile. Repair is typically made by localized short removal or single-point bonding. Over time, we believe that our materials and processes can be controlled to the point at which inspection and repair can be dramatically reduced.

Final Test

The goal of our test process was to ensure that multichip units would operate successfully in a system environment. Since no test equipment manufacturer offered a system that met our needs, we developed our own by working with several Digital groups as well as outside suppliers. The system contains three major stations. The first provides alignment information and can also read visual serial and part numbers. In the second station, low voltage shorts are determined between nearest neighbor leads. This step supplements our inspection for shorts described above. In the final station, we test for connector opens, thermal measurement (die attach integrity), scan chain integrity, and scan pattern data. The scan pattern testing is done in several bursts of the clock at system speed. In addition, diagnose capability is provided by flying probes, voltage and clock margining, and a thermal chuck to vary temperature.

Conclusion

Successful use of advanced interconnect technologies requires a seamless phased development process that begins with advanced development and continues through volume manufacture. The HDSC and multichip unit technologies have successfully achieved the volume manufacturing phase. Using the products and technologies described in this paper, we have played a key role in the introduction of the VAX 9000 system to the marketplace. Extensions of this manufacturing process will ensure that this technology base can be applied across a wide spectrum of products of both higher and lower performance.

Matthew S. Goldman
Paul H. Dormitzer
Paul A. Leveille

The VAX 9000 Service Processor Unit

The VAX 9000 service processor unit provides the front-end services needed to support a highly available and reliable mainframe system. The unit is closely linked to the VAX 9000 system to provide realtime detection and recovery of system failures. However, the unit is independent enough to be isolated for maintenance without affecting normal system processor operation. This combination is a first for VAX systems. The service processor also provides various debugging features that were essential for development and early manufacture of the VAX 9000 system. These features utilize a system-wide scan architecture to achieve direct access to machine-state, which provides extensive visibility and control of system logic functions. The inclusion and use of such a scan architecture is a new feature for a Digital processor.

The VAX 9000 service processor unit (SPU) is designed to provide a dedicated subsystem for service and maintenance support for the VAX 9000 family. The SPU serves two distinct roles. It functions as the familiar operator interface (i.e., VAX console) and as a maintenance vehicle used to diagnose and isolate system processor hardware faults.

The SPU performs the following major front-end services:

- System initialization
- Power system control and monitoring
- Environmental monitoring
- Clock control and monitoring
- VAX 9000 operating system access to SPU mass storage devices (disk and tape)
- Remote diagnosis port support
- System error detection, recovery, and reporting

The SPU also provides or assists in the following system diagnosis functions:

- SPU module self-tests
- Scan system diagnostics
- Clock system diagnostics
- Scan pattern structural diagnostics
- Structure cell (e.g., self-timed random-access memory [RAM]) diagnostics
- XMI-to-system control unit adapter interface test
- Symptom-directed diagnosis support

In addition to its use as the front-end processor for the VAX 9000 system, the SPU was embedded in several manufacturing and engineering test vehicles. In the Debugging Features section of this paper, we describe how the SPU was used as a debugging tool during VAX 9000 product development and the various debugging features we provide to help locate design and fabrication problems.

A major goal of the SPU was to perform system-wide error detection and recovery functions for the VAX 9000 processor. In the Error Handling section of this paper, we detail the types of errors that the SPU handles and how error detection, reporting, and recovery occurs.

Another of our design goals was to be able to service the SPU without adversely affecting the operation of the system processor. This feature was needed to support the high availability requirements of a mainframe system. To meet this goal, we designed mechanisms to enable the VAX 9000 operating system to determine that the SPU is not functional (whereupon the operating system takes the appropriate action to secure its own operation), as well as recognize and reintegrate with the SPU when the SPU is functional again.

If the VAX 9000 operating system attempts to access one of the SPU-based processor registers and the SPU does not respond, the failure is detected by using the usual register time-out mechanism. However, because the SPU is responsible for system error handling, SPU failures must be detected quickly to enable the SPU to respond to a system error should one occur. Consequently, we developed a keep-alive protocol with which the VAX 9000 operating system can determine SPU failures without relying on operating system accesses to SPU-based processor registers. The keep-alive mechanism is described in more detail under the Error Handling section of this paper. Both the time-out and keep-alive mechanisms work regardless of whether the SPU has an unexpected failure or undergoes a scheduled power-down.

Should the SPU require service, field upgrades may be performed easily and quickly because of the modularity of the hardware, which is primarily VAXBI bus interface-based adapters. The VAXBI backplane minimizes downtime because modules can be removed or inserted without requiring recabling. When power to the SPU is restored, SPU self-tests are performed. The SPU's operating system then boots automatically and signals its availability to the VAX 9000 operating system.

The SPU is designed to continue operation even if the SPU primary storage device, an RD54 Winchester disk drive, fails, which further increases the availability of the SPU. For customers who require data security and high availability, we designed a system configuration option that does not use a disk drive. In this case, the SPU boots from TK50 cartridge tape. The SPU functions that require a disk drive for data storage (e.g., SPU-generated error logs) are disabled in this configuration.

SPU Architecture

A block diagram of the SPU architecture is shown in Figure 1. The service processor module, scan control module, and power and environmental monitor were designed uniquely for the VAX 9000 system. The disk controller, tape controller, as well as the memory daughter board were available from other

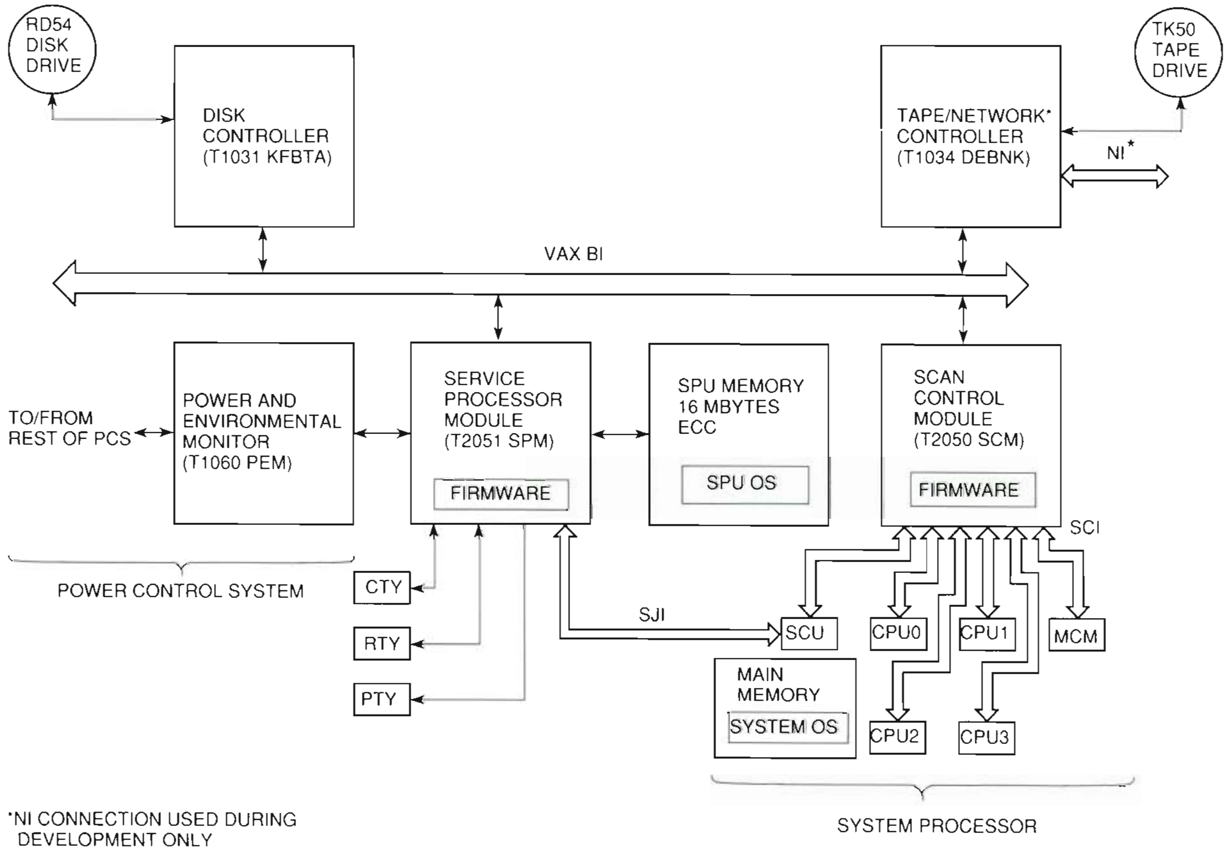

RD54 DISK DRIVE
DISK CONTROLLER (T1031 KFBTA)
TAPE/NETWORK* CONTROLLER (T1034 DEBNK)
TK50 TAPE DRIVE
NI*
VAX BI
TO/FROM REST OF PCS
POWER AND ENVIRONMENTAL MONITOR (T1060 PEM)
SERVICE PROCESSOR MODULE (T2051 SPM)
FIRMWARE
SPU MEMORY 16 MBYTES ECC
SPU OS
SCAN CONTROL MODULE (T2050 SCM)
FIRMWARE
SCI
POWER CONTROL SYSTEM
CTY
RTY
PTY
SJI
SCU
CPU0
CPU1
MCM
MAIN MEMORY
SYSTEM OS
CPU2
CPU3
SYSTEM PROCESSOR
*NI CONNECTION USED DURING DEVELOPMENT ONLY

Figure 1 VAX 9000 SPU Block Diagram and Interconnects

Digital products. Every SPU VAXBI adapter provides its own built-in self-test diagnostics.[1]

SPU hardware is based on either industry-proven (e.g., 7400-series TTL components, complementary metal oxide semiconductor [CMOS] gate arrays) or Digital-proven technology (e.g., VAXBI, Digital custom CMOS devices) to ensure that the unit is an effective debugging platform for a system processor based on leading edge technology. As a result, the inherent risk and learning curve associated with new technology were avoided and the SPU was ready and available during the VAX 9000 system prototype debugging process.

The SPU also was made available to manufacturing process and tester groups (e.g., multichip unit tester) for use with their designs. The advantages to this approach were that technicians became familiar with the same subsystem that would be used in the VAX 9000 family, and the test programs could be transferred for use in other test environments that also used the SPU, including the VAX 9000 system itself.

The service processor module is the primary processing element of the SPU and is the VAXBI host adapter. Based on the MicroVAX 78032 chip and several custom-designed application-specific integrated circuits (e.g., SPU-to-system control unit adapter, SPU memory controller), the module contains all the hardware necessary to store and execute the SPU operating system. The on-board firmware contains a VAX standard console interface to load the SPU operating system during initialization and to assist in subsystem debugging. The SPU-to-system control unit interface (SJI) connects the service processor module to the system control unit and is the primary communication path between the SPU and the VAX 9000 operating system.

The scan control module is the control interface to the VAX 9000 scan system, which is the visibility and maintenance path to the system processor. Like the service processor module, the scan control module is based on the MicroVAX 78032 chip and several custom-designed application-specific integrated circuits (e.g., scan control chip, scan distribution chip). On-board firmware provides high-level functions that allow the service processor module to continue processing while scan-related operations, including logical-to-physical signal translations, are performed concurrently by the scan control module. The scan interconnect (SCI) connects the scan control module to the system processor (i.e., one to four CPUs and the system control unit) and the master clock module. Using this interface, the system processor may also interrupt the SPU when the processor needs service. This type of interrupt request is known as an attention.

The SPU is integrated into the system cabinet to better meet the performance requirements necessary for system error recovery and VAX 9000 operating system boot. Cabinet integration substantially decreases interconnect distances to processor logic and ensures that all cables are kept internal to the cabinet. Another reason for choosing the VAXBI backplane card cage is that its form factor is small, which reduces the cabinet area needed (cabinet area is always in high demand), yet the user-definable zones provide the high pin density required for interconnects (i.e., 180 I/O pins per VAXBI slot).

Communication Path

The SPU communicates with the system processor using the SJI. This interface is used to load the primary bootstrap into the VAX 9000 main memory, transfer error and machine-check information to the VAX 9000 operating system, provide file transfer access between the VAX 9000 operating system and the SPU's RD54 disk drive, access system main memory, and access system I/O registers.

The VAX 9000 operating system accesses the SPU as if it were a standard I/O device. The SPU is an independent subsystem and does not rely on the execution unit of the system processor to be a console processing engine, as was done in previous VAX systems. There are several benefits to this design approach. Each CPU has equal access to the SPU and may interrupt the SPU to request service. In addition, the SPU may interrupt any of the CPUs to request an operating system service. The SPU may be used as a debugging tool during system processor debugging because it does not require that any portion of the system processor be operational. The fact that the SPU could be used as a debugging tool was an extremely important benefit for the VAX 9000 system debugging effort. The debugger did not have easy access to the logic elements because of the advanced packaging and circuit integration of the VAX 9000 system. Therefore, SPU services were utilized in lieu of logic probes. Further, because the SPU no longer uses the CPU for system access, console support microcode (i.e., the collection of microcode procedures traditionally used for access to the system processor, memory, and I/O registers) is not required. The benefit of this process is that valuable VAX 9000 control store space could be used for system microcode or to reduce the control store size. For example, in the VAX 8650 system, console support microcode occupies approximately 180 microword locations.

VAX 9000 operating system access to the SPU is through the VAX console register set. We extended the VAX console register set to provide access to the enhanced capabilities of the SPU. Additional registers include transmit function request and parameter and receive function request and parameter (i.e., TXFCT, TXPRM, RXFCT, RXPRM). Table 1 lists the functions provided by these registers.

SJI communications are in the form of 14-byte packets that contain the command (i.e., function), address, and data. Packets are sent and received over two 8-bit data paths that provide full duplex operation. Data transfers peak at 3.5 megabytes (MB) per second for quadword transfers.

When the VAX 9000 operating system executes a Move_to/from_Processor_Register instruction that specifies an SPU register, the system control unit sends an I/O command packet, through the SJI, to the SPU to initiate the system request. Then the SPU typically uses an interrupt command packet, which generates an interrupt to the specified CPU. The two other packet types are direct memory access and error correction code.

Visibility Path

In the development and manufacture of a complex computer system, extensive testing methods must be available to ensure functional operation and product quality. Design engineering no longer can use manual probing techniques in prototype debugging. Space limitations have resulted from advanced packaging and the close pitch of integrated circuit I/O pins, which is due to high integration levels. Failure isolation must be performed in the manufacturing process, often without an extensive knowledge of the machine design.

A separate visibility and control path in the system processor of the VAX 9000 system provides nearly 100 percent visibility to the machine-state. The visibility path eliminates the need to select a subset of visibility points to meet all test needs, as was done with previous VAX systems. In addition, the path allows designers to directly alter the entire machine-state, which is a major advantage for design and process debugging. A VAX 9000 uniprocessor (i.e., one CPU and system control unit) contains over 26,000 access points.

The path is called the VAX 9000 scan system and is controlled by the service control module. The scan system is the foundation for direct access by prototype debuggers, system error recovery

Table 1 RXFCT/RXPRM and TXFCT/TXPRM Functions

RXFCT/RXPRM Functions (SPU to System Processor)
Remove processor
Add processor
Mark memory page bad
Request pages of memory
Send error log entry
Send OPCOM message
Get datagram buffer
Send datagram
Return datagram status
Set keep-alive state
Abort datalink
Error interrupt
TXFCT/TXPRM Functions (System Processor to SPU)
Get hardware context (of a halted CPU)
Virtual block file operation (access to SPU disk and tape)
Keep-alive
Send datagram
Return datagram status
Switch primary
Reboot system request
Clear warm start flag
Clear cold start flag
Boot secondary processor
Halt CPU and remove from available set
Halt CPU and keep in available set
Console quiet
Set interrupt mode
Abort datalink
Reset I/O system
Disable vector unit
Set keep-alive state
Start processor
Margin power
Margin clock
Fault signal
Start error window
End error window
Report error in window
Get error log entry
Get unmarked error log entry and mark
Enable halt restart
Get I/O physical address memory map configuration
Get physical address memory map configuration

software, and diagnostics to observe and alter the VAX 9000 machine-state. Some functions provided by the scan control module and supporting SPU software are

- Load and save processor state
- Scan pattern execution
- Continuity testing of the processor's scan hardware
- Multichip unit type and revision information extraction
- Processor attention notification

A block diagram of the VAX 9000 scan system is shown in Figure 2. The scan control module connects to the system planar module over the SCI. Scan and clock distribution logic, contained in a macrocell array on the planar module, distributes data and control signals over the scan bus to each of the multichip units. A clock distribution chip at the hub of each multichip unit further distributes the scan bus signals to the macrocell arrays, which are integrated circuits that contain system logic.

As shown in Figure 3, the state devices within a macrocell array are scan latches. The latches are connected serially to form a ring or chain by connecting the Scan_Data_Out line of each latch to the Scan_Data_In line of the next latch. The end links are connected to the clock distribution chip. When the system clocks are running, data is loaded into the latch from the system data input. During scan operation, system clocks are not active. Generated by the scan control module, the scan clocks load the latch with data from the scan data input. Consequently, the scan control module reads system state by issuing scan clocks, which serially shift system data to the scan control module. System state is changed when the scan control module drives new data to the system latches while issuing scan clocks.

An architectural feature permits each multichip unit to generate an attention interrupt directly to the scan control module over the scan data return line. Attentions notify the SPU of system events, such as processor errors, memory self-test completion, CPU halts, and keep-alive responses.

System diagnostics can diagnose the SCI by using the same control signals as used for scan system operation. Dedicated logic and special routing of the scan lines provide failure isolation. Stuck-at faults and disconnect conditions can be isolated to the multichip unit.

Debugging Features

In addition to its use as the VAX 9000 front-end processor, the SPU provides a variety of features for debugging and troubleshooting multichip unit logic configurations. These features were required because all multichip unit logic visibility and control is handled through the SCI, which connects directly to the SPU. The use of scan latches to access internal logic states is a first for VAX systems and challenged the designers to define and deliver the necessary tools and features to assist the multichip unit debugging effort. Furthermore, the features provided by the SPU had to apply to various tester environments, ranging from single multichip units mounted in probe stations to full system configurations. Additional requirements to support the clock and power system test stations made it clear that the SPU would have to be adaptable to a variety of environments.

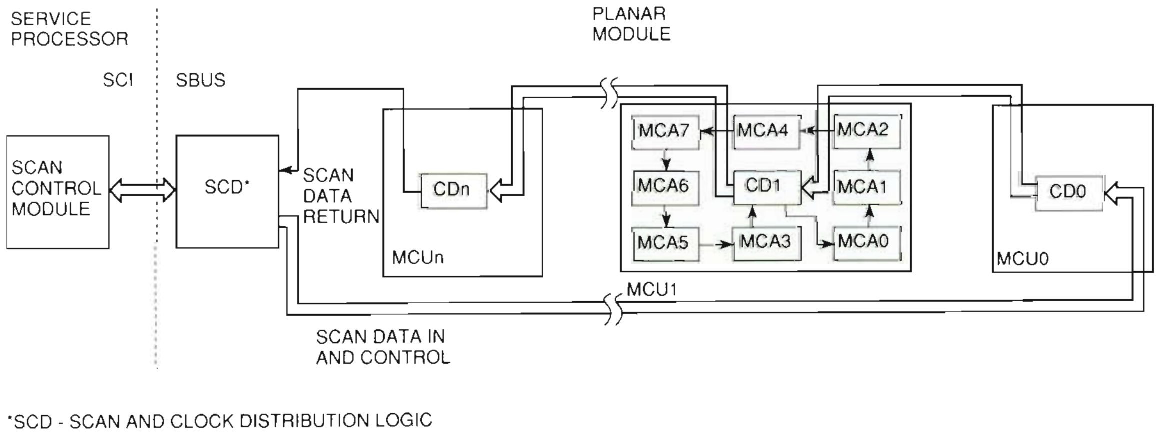


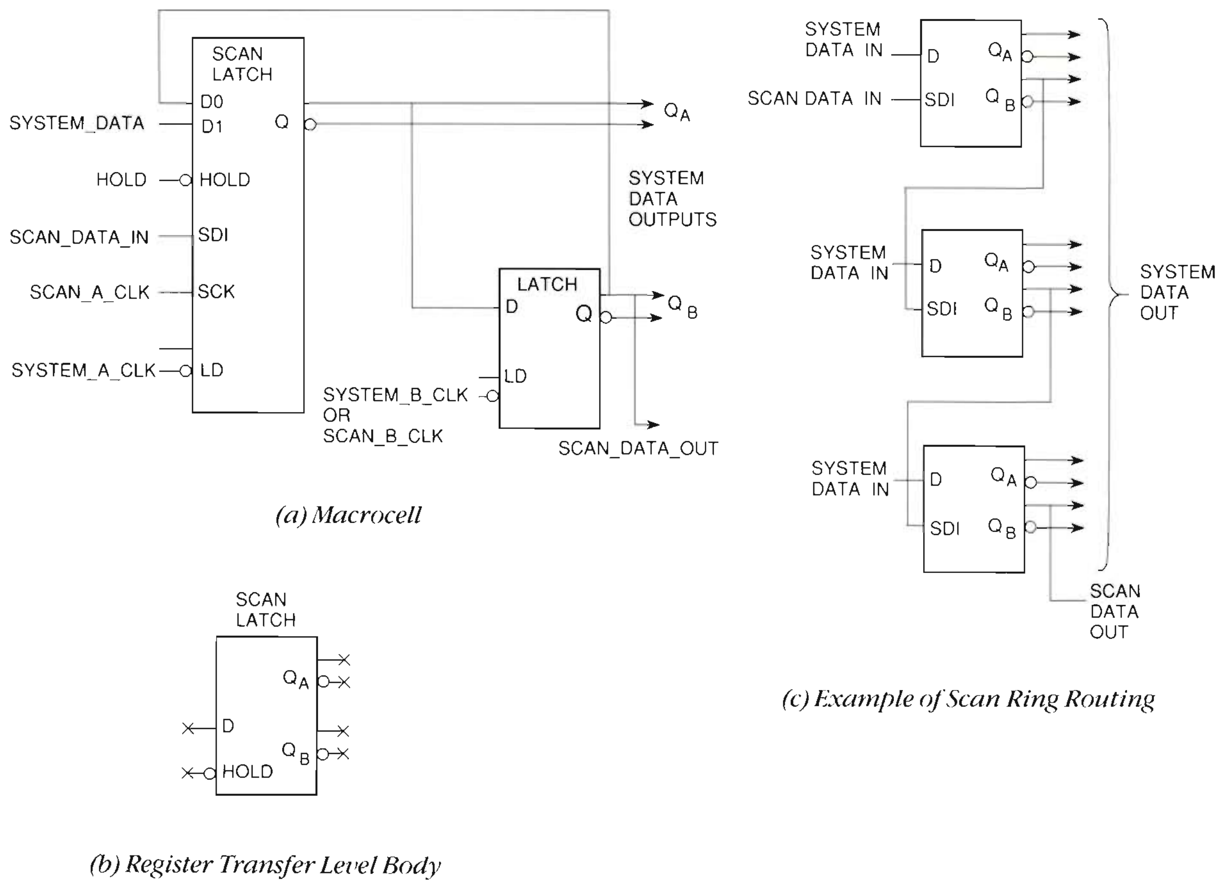
Figure 2 VAX 9000 Scan System

Generic SPU Environment

To satisfy the SPU's fundamental requirement of being adaptive to differing environments, test stations had to comply with the physical interfaces provided by the SPU in the VAX 9000 system. We did not have the resources to develop tester-specific interfaces, so it was agreed by all development groups that testers would comply with the SPU's system environment, as shown in Figure 4.

This generic environment allowed SPU hardware and software development to concentrate on supporting the needs of the VAX 9000 system and to receive valuable feedback and debug time from the test station groups prior to system processor availability. Several major benefits were achieved with this approach:

- Because the SPU software had early exposure in the test station environments, the software was debugged and tested to an acceptable level before full system configuration support was needed. In particular, the command language interpreter had to be ready to provide the basic SPU functions, such as file manipulation, command procedures, symbol and expression evaluation, and command recall.
- Technicians working at the test stations had an opportunity to develop an understanding of the SPU's operation that was then carried forward to other SPU-based debugging environments.
- Economies of scale existed because one front-end development effort supported both in-house test stations and the final product.
- Many of the primitive debugging features developed for tester use were found to be just as valuable during actual system debugging, particularly the fundamental commands that allow direct control of the SCI signals.



(a) Macrocell

(b) Register Transfer Level Body

(c) Example of Scan Ring Routing

Figure 3 Scan Latch

Primitive Debugging Features

One of the most basic features offered by the SPU is the ability to directly access the internal registers of the clock and power systems by using the EXAMINE and DEPOSIT commands. When combined with the general command language of the SPU, these commands allow debuggers to create procedures to control, test, or interrogate various components of the VAX 9000 system. For example, command procedures have been created to monitor and exercise the power, clock, and SPU subsystems as part of the reliability and design verification test plans.

Other low-level commands provide the means for debugging and troubleshooting the scan paths. For example, the SET and SHOW commands permit individual control of the SCI signals. Using these commands, the SCI can be observed statically and be stepped through its operations. Precise control of the SCI signals provides easier debugging of the scan paths in the early multichip units, primarily

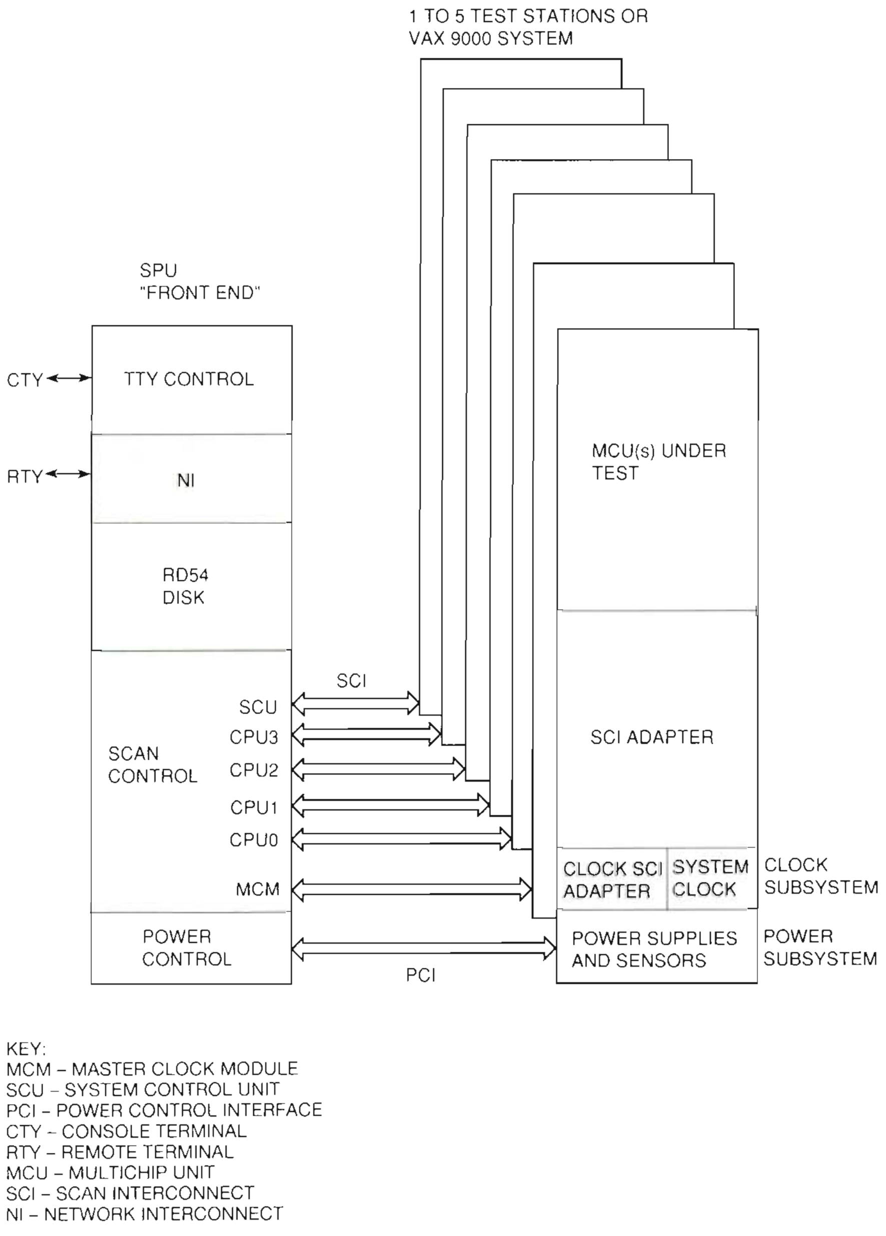


Figure 4 Generic SPU Environment

because static signal operation gives more isolated feedback than an interconnect that runs at full speed through many state changes.

Once the single-step operation of the SCI was verified, the full-speed operation of the scan logic in the multichip unit could be tested. Commands that collectively display and modify the scan latches of a single macrocell array were an effective way to verify the operation of this logic. Latch data are displayed in their physical form as a string of hexadecimal digits, the length of which varies from one macrocell array to the next, in the range of 100 to 300 bits. Provisions also exist to select scan clock rates ranging from 3200 nanoseconds (ns) per bit to the full 100 ns per bit operation.

High-level Debugging Features

The use of the scan architecture as a means for initializing and debugging the VAX 9000 processor was a first for a VAX front-end. Because physical latch information is cryptic and difficult to use, we designed the SPU to provide the necessary translation from a logic signal name to its corresponding scan latch in the machine. We modeled the SPU's user interface after the user interface of DECSIM, one of Digital's logic simulation utilities. Engineering used the DECSIM interface during the VAX 9000 design phase and was already familiar with its user interface.

The majority of the user interface development work involved the EXAMINE and DEPOSIT commands and their associated data structures that resemble the procedure usesd in the DECSIM system. These commands provide access to the more than 26,000 signals accessible through the scan system in a uniprocessor system. The SPU also maintains the design hierarchy of the signals, which permits signals to be referenced as they appear on the pages of the logic schematics. A watch point and trace point capability, modeled after similar features in the DECSIM system, simplifies the task of monitoring state changes in the machine. Because the processor clocks are single-stepped, signals which change state are displayed automatically.

Using the DECSIM system as the model for these SPU features produced two advantages:

- Designers moved from the simulation environment (i.e., using the DECSIM system) to actual debugging (i.e., using the SPU) with virtually no training. Although the precise syntax of the SPU's commands is not always identical to the syntax of DECSIM commands, the concepts are the same. Therefore, first-time users overcome the differences quickly.
- All register transfer-level signal names correspond with those present on logic schematics, including the logical design hierarchy. This correspondence makes the relationship of displayed signal names and schematic signal names easy (e.g., %CPU0.VAP.VAP0.ALU_FUNCTION_H<0>).

The translation from a logical signal to its associated scan latch uses data structures supplied in a configuration database file, which is loaded into SPU memory during SPU initialization. All CPUs with identical multichip unit configurations (i.e., same CPU revision) share the same configuration database memory image. The system control unit always requires its own database. Only two CPU revisions can be supported at one time because of SPU memory constraints for storing the separate configuration databases. However, by providing for two CPU revisions, the needs of single and dual CPU configurations were completely satisfied. Further, it was possible to upgrade homogeneous triple and quadruple configurations in a stepwise manner.

Macrocode Execution

Initial system-level multichip unit configurations consisted only of a scalar CPU. The system control unit was not yet available as a result of the extended simulation of the design. Fortunately, we had anticipated the possibility of running partial configurations and could provide modes within the SPU software to redirect commands that normally access main memory (e.g., EXAMINE, LOAD) to access the CPU's 128 kilobyte (KB) system cache or 8KB virtual instruction cache instead. The first VAX macro-instructions were loaded and executed on the VAX 9000 system using this technique. An additional feature, which involved minor hooks in the system microcode, provided a means for the VAX instruction set diagnostic, EVKAA, to communicate with the console terminal through scan attentions rather than by using the system control unit. Thus, the diagnostic could run to completion.

Advanced Debugging Features

Although not obvious aids to VAX 9000 debug, the following features were indispensable or, at the least, reduced debugging time and effort:

- A character-cell windowing capability that allows system microcode sources to be automatically located, displayed, and updated on the screen as the system is single-stepped. We modeled this feature after the VAX debugger's windowing capability because most VAX engineers are familiar with this capability. Windowing eliminated the need for hard-copy microcode listings and the logistical problems associated with their use.

- By connecting the SPU to the engineering network during development, timely updates of SPU software were made possible. This kept the VAX 9000 debugging effort, which was occurring simultaneously on several systems, up to date with the latest SPU software fixes and enhancements. Together with the multisession capability of the SPU operating system, the use of the network made remote debugging a reality throughout the VAX 9000 debug phase.
- Because the SPU had to initialize the VAX 9000 system thousands of times during system debugging, the unit was designed to perform system initialization as efficiently as possible. For example, the loading of structures (e.g., control stores or cache tags) was optimized by overlapping the operation of three MicroVAX-based processors: the service processor module, the scan control module, and the disk controller.

The debugging features located early design and fabrication problems in the clock, power, scan, and processor logic areas. Ultimately, the features were used to initialize and run the first VAX 9000 system.

Error Handling

To support high system availability, accurate and timely error detection and logging is required. Error data collection cannot depend upon host system availability, and the data must be available when the system is not functional. Therefore, an independent service subsystem that can collect data from all system components, render it into a useful format, and store and display the information is needed.

The service subsystem must also be organized in such a way that if it fails, it does not directly cause system processor failures. Repair, reboot, and system reintegration must occur without interfering with system processor operation. The SPU meets these requirements; it is a fully independent computer that runs its own operating system with dedicated peripherals. The SPU performs system-wide error detection and reporting functions and provides advanced error recovery features for the system processor.

Error Detection

The SPU reports errors in its own VAXBI adapters, the service processor module, the scan control module, the power and environmental monitor, the disk controller, and the tape controller. It also reports errors in various parts of the VAX 9000 system, such as the system control unit, the CPU's, the memory system, the master clock module, and the power and environmental systems. Because failures in any of these subsystems can incapacitate the VAX 9000 system, none of them reports its errors directly to the VAX 9000 operating system.

SPU Errors The disk controller, tape controller, and scan control module use the VAXBI VAX port protocol to report errors. The power and environmental monitor passes error information to the service processor module through its private bus, the SPU-to-power control system interface.

Environmental Exceptions The power and environmental monitor monitors the regulator intelligence cards, airflow sensors, and temperature sensors throughout the system. When it detects any problems in operating voltages, currents, temperatures, or airflow, it notifies the service processor operating system, which logs the error condition.

Clock Exceptions When the master clock module detects an error in either the clock phase or the clock frequency lock, it generates an attention to the scan control module, which interrupts the service processor module. The SPU operating system logs the error condition.

Memory Error Correction Code Events The main memory of the VAX 9000 system contains error-correcting logic to correct single-bit errors and detect double-bit errors. When a memory location with a single-bit error is read, the system control unit corrects the error and passes the corrected data to the requesting device. It also writes an SPU register with the error type and the failing memory address. The SPU operating system writes this information to the error log. If the system control unit detects a double-bit error or reads a marked-bad location, it passes the bad data, marked as bad, to the requesting device and notifies the service processor operating system, which logs the error. The bad data is handled locally by the requesting device, usually by generating an error of its own.

CPU and System Control Unit Errors When a CPU detects an error in a parity checker, it attempts to come to an instruction boundary and halt. Once it has halted, the CPU sweeps its cache. When the cache sweep is completed, the CPU asserts an attention to the scan control module to inform the SPU that recovery is required. When the system control unit detects an error, it first asserts a fatal error signal to each of the CPUs, and then asserts an attention. When the CPUs receive the fatal error signal, they attempt to come to an instruction boundary and halt. Once halted, the CPUs assert attention lines to the scan control module. The caches are not swept since their path to memory, the system control unit, is not working.

Keep-alive, Timeout To ensure that a CPU is not hung by an undetected error, the SPU periodically sends a keep-alive interrupt to each CPU. CPU microcode services the interrupt at the next macroinstruction boundary by asserting an attention to the scan control module. If the CPU should be hung by an undetected error, the SPU times out while it waits for the keep-alive reply attention and, thus, determines that there has been an error. Similarly, the primary CPU monitors the SPU by sending it a keep-alive request through the TXFCT register. If the SPU does not respond to this request within a timeout period, the VAX 9000 operating system assumes that the SPU is hung and reboots it using a VAXBI reset. When the SPU reboots, it reintegrates itself with the rest of the VAX 9000 system without interfering with system operation.

Error Reporting

When errors are reported to the SPU operating system, the error formatting facility logs the error information locally and reliably transmits it to all intended receivers. The error formatter maintains the error log file ERRLOG.SYS on the SPU RD54 drive, passes error log entries to the VAX 9000 operating system to be logged in the system error log, and also passes the entries to any SPU software that requests them. The error formatter writes the error log file using the SPU operating system disk I/O functions, passes the error log entries to the VAX 9000 operating system using an RXFCT function, and passes the error log entries to other SPU processes using the SPU port protocol. If the RD54 drive is not available, which prevents access to the SPU error log, the error formatter continues to send error log entries to the VAX 9000 operating system and to other SPU processes.

The SPU error log contains all the error log entries collected by the SPU (but not those collected by the VAX 9000 operating system) and time stamps, which are logged every ten minutes. Should an SPU operating system crash occur, the time stamps may be used to determine the approximate time of the crash. Errors are logged regardless of the state of the system processor. As a result, information is available for analysis even in the event of a total processor failure. The error log file may also be transferred to TK50 tape for off-site analysis.

The error formatter passes error information to the VAX 9000 operating system by copying the error log entry to system memory and then invoking the RXFCT function to notify the VAX 9000 operating system that the entry is available. Should the operating system not respond to this notification, the error formatter assumes that the operating system has crashed and writes the error log entry to a temporary data file. When the VAX 9000 operating system reboots, it notifies the SPU by using a TXFCT function. The error formatter then reads any saved error log entries from the data file and transmits them to the VAX 9000 operating system. This protocol ensures that all collected error data is eventually reported in the system error log.

The error formatter also maintains a SPU port to which any process running on the SPU may connect. Connected processes receive copies of all error log entries as the entries are logged. This port is used by EWKCA, the symptom-directed diagnosis tool, which analyzes errors as they occur and determines which system components might have caused the failure. The port is also used for system debugging by the error insertion program to verify that errors are being logged and analyzed correctly.

Snapshots In addition to its error logging facilities, the SPU operating system provides the ability to take "snapshots" of the system processor state. The snapshot file provides a detailed record of system context, which allows engineers to take a snapshot of a hung system and reboot it, and then analyze the snapshot file while the system proceeds to perform other useful work. The snapshot display utility is used to examine the data in a snapshot file. In addition to formatting the data in the snapshot file, the snapshot display utility can be used to examine any scan latch in the file, by name, in the same fashion as the console EXAMINE command is used on the actual hardware. The data available in a snapshot file is summarized in Table 2.

Error Recovery

The high level of visibility achieved by the scan system allows the SPU to provide extensive error recovery facilities for the VAX 9000 processor. SPU-based recovery offers several advantages over

Table 2 Snapshot File Contents

Section	Contents
Revision Section	All multichip unit revisions
	All SPU adapter revisions
	Microcode revisions
	All XMI adapter revisions
	All VAXBI adapter revisions
Power Section	All power control system registers
	"Sense power" results
Clock Section	All master clock module registers
SPU Section	All SPU-to-system control unit adapter registers
I/O Section	XMI device error registers
	VAXBI device error registers
	XMI-to-system control unit error registers
System Control Unit Section	All scan latches
	Last 50 entries from system control unit micro program counter history buffer
	All cache tags
	All other logical structures (e.g., control stores)
	Configuration database version
	I/O physical address memory map
	Memory physical address memory map
	Nonexistent physical address memory map
CPU Section (Repeated Once for Each CPU)	All scan latches
	Last 50 entries from program counter history buffer
	All cache tags
	All general-purpose registers
	All internal processor registers
	All other logical structures (e.g., control stores)
	Top 50 longwords of current mode stack
	Top 50 longwords of interrupt stack
	32 bytes of instruction stream around each program counter in history buffer
	Configuration database version
	50 micro program counters, collected by stepping the clocks

traditional microcode-based error handling. The CPU hardware resources that might otherwise be used for error handling were available for the logic designers to improve the system performance. Because the error data is processed external to the failing component, the recovery process itself is not suspect. Finally, because the system clocks are stopped while recovery takes place, erroneous data does not propagate throughout the system.

Traditionally, many microwords in the CPU control store (approximately 500 in the VAX 8600 system) are used for error recovery microcode. However, because the SPU is responsible for VAX 9000 error recovery, additional control store space is available for instruction microcode. If this had not been the case, we might have had to make a space trade-off between instruction and recovery microcode, which could have resulted in more emulated instructions and a performance penalty for VAX instruction execution speed.

Because the scan system allows the SPU to determine the state of every scan latch in the CPUs and system control unit, logic designers were able to place error detectors anywhere in the design without organizing the detectors into microcode-readable error registers. As a result, significantly more error detectors were used for precise error analysis than would have been possible if the scan system were not available. Each VAX 9000 CPU contains over 450 error detector latches.

Several advantages are derived from performing error recovery independently from a failed component. The most obvious advantage is that hardware, which may be failing, is not used to control the recovery. Once the system processor state has been scanned out into SPU memory, analysis is a function of software running on a known good processor. The SPU analyzes the data and then scans a correct state into the system processor. The entire process is performed while the system clocks have been stopped. Therefore, processor errors cannot cause "error loops;" that is, the error recovery process itself gets errors from a corrupt processor state. SPU-based error recovery can completely reset a corrupt system, regardless of the degree of corruption.

The VAX 9000 error-handling facility takes advantage of many advanced software features that are available in the SPU operating system. It uses configuration database information to access system processor signals by name rather than by scan ring locations. Thus, one version of the error handling code can handle several different physical processor variations. The error handler also uses the

SPU operating system structure access routines to read and write the processor structures, again, by burying the physical implementation in the configuration database. As a result, the error handler can look at the architectural features of the VAX processor rather than at the gate-level design of the VAX 9000 system when performing error analysis. The benefit of this approach is that recovery procedures are based on the system architecture, rather than on the machine implementation.

One of our design goals for the VAX 9000 error-handling system was to recover from most errors in under 500 milliseconds. Longer delays increase the probability that I/O devices will time out while waiting for the operating system to respond to requests and cause the operating system to crash, even if the error-handling system successfully recovers from the error. The error handler meets this goal by taking maximum advantage of the multiprocessing capabilities of the tightly coupled hardware design of the service processor module and scan control module. Error recovery is split into a multistep process that keeps both SPU processors working on the problem simultaneously.

The error handler recovers a failed system in five phases: data collection, data analysis, error recovery, macrostep, and cleanup. In the data collection phase, the scan control module scans out all scan rings of the failed CPU or system control unit. In the analysis phase, the scanned data is used to determine which architectural features of the system have been corrupted (e.g., caches, general-purpose registers, internal processor registers, microcode stores, and the translation buffer).

In the recovery phase, the error handler attempts to restore the system to a state in which no software-visible data is corrupt. Therefore, the software running on the VAX 9000 system, including the operating system, is unaware that an error has occurred. The error handler determines whether the system state can be restored successfully or if a machine check must be generated to allow the VAX 9000 operating system to attempt to handle the error on a higher level. It then restores the CPU to a known good operating state, by using latch data from the configuration database, and corrects any corrupted software-visible data.

In the macrostep phase, the error handler turns on the system clocks to allow the failed CPU to attempt to macrostep one instruction. If the macrostep completes successfully, the recovery is considered successful and system operation is allowed to continue. In the clean-up phase, the SPU processes the data from the data collection phase into an error log entry, posts the entry, and cleans up the data structures that will be used to recover from the next error.

Errors that are too severe for the error handler to handle are signaled to the SPU command interpreter, which can run command scripts to completely reinitialize the machine and reboot the VAX 9000 operating system. Examples of such severe errors are hard errors that prevent VAX 9000 operating system machine check code from running and errors that cause a CPU to fail its macrostep.

Summary

The SPU is a dedicated subsystem for service and maintenance support for the VAX 9000 family. It is closely linked to the VAX 9000 processor to provide system error recovery. It also presents a high-level interface with which debuggers may observe and control system processor activity. Through the use of a system-wide scan architecture, the SPU provides access to nearly 100 percent of processor machine-state. Finally, the use of the SPU in various tester environments greatly assisted the multichip unit debugging effort and provided advanced training for VAX 9000 system debuggers.

Acknowledgments

The authors wish to thank Michael Evans, the SPU project leader, whose drive and ambition provided the force behind the project's success. We also wish to acknowledge the other members of the SPU design team: Karen Barnard, Stephen Conway, David D'Antonio, Susan DesMarais, and Brian Rost.

Reference

1. D. Chin et al., "The Unique Features of the VAX 9000 Power System Design," *Digital Technical Journal,* vol. 2, no. 4 (Fall 1990, this issue): 102–117.

Derrick J. Chin
Barry G. Brown
Charles F. Butala
Luke L. Chang
Steven J. Chenetz
Gerald E. Cotter
Brian T. Lynch
Thiagarajan Natarajan
Leonard J. Salafia

The Unique Features of the VAX 9000 Power System Design

The VAX 9000 series represents Digital's first implementation of a mainframe computer system. To be competitive in this market, the power system for the VAX 9000 series had to provide high system availability. To meet this goal, the system includes features neither considered nor found in previous large Digital computer systems. Some of these features are the use of redundancy in parts of the design and the addition of more power system diagnosis capability for quicker fault isolation and faulty unit replacement. Other features provide competitive advantages in specific marketplaces, such as meeting low harmonic distortion for AC input current, which is an emerging European AC power quality standard. Simulation tools, which are used more prevalently in digital logic, were used to improve the power design.

The two key requirements of the VAX 9000 power system are high availability and the inclusion of competitive features. High availability for the power system means we had to achieve the highest unit regulator reliability possible by using the appropriate technology available. Further, we had to deliver both more power system and cabinet environmental monitoring and diagnostic capability that could reduce the time spent in isolating and replacing a malfunctioning unit. Competitive features mean designing into the system features that would be either better than expected or advantageous to the VAX 9000 system in certain markets.

A full discussion of all the methods used to meet these requirements is too long for this paper. Therefore, the discussion in this paper focuses on some of the unique applications of the power technology and tools used in the design of the VAX 9000 system:

- Power system architecture
- Improved load sharing
- Simulation
- Increased control and monitoring
- Low harmonic distortion

One of the issues we had to decide in designing the power system architecture was how many regulators should be used. A large number of regulators in a power system can cause the mean time between failures (MTBF) to be lower than desired. Therefore, we chose to use redundant regulators in the power system architecture for improved availability.

Another means of increasing the MTBF was achieved by improving the load sharing among the parallel regulators that power a low-voltage current load. With this feature, no one regulator operates at a percentage of maximum rating much higher than its parallel regulators, which eliminates the higher operating temperatures that can occur and, as a result, lowers the MTBF.

High regulator reliability results from good circuit design. Three examples of the unique simulation features that were used as checks on circuit designs are discussed in the Simulation section of this paper. In one case, simulation pointed the way to a circuit problem that was not initially apparent. In another case, simulation was used to verify on paper that the number of regulators chosen to power a specific load was sufficient.

High availability can be achieved by reducing the time to isolate a system problem and replace the malfunctioning unit. A power and cabinet monitoring module, EMM, fulfilled this purpose in the VAX 8000 systems. The power control subsystem, PCS, used for this purpose in the VAX 9000 systems, expands on the diagnostic and monitoring features of the EMM.

Meeting emerging European AC power quality standards was viewed by the European sales force as a distinct competitive advantage for the VAX 9000 system. A proposed standard we wanted to meet was to achieve low harmonic distortion of the input AC current wave form, which was met in the utility power conditioner (UPC) front-end design of the power system. High availability was designed into the UPC through such features as redundancy and increased immunity to power line disturbances from a commonly accepted industry practice of one AC cycle to ten AC cycles.

VAX 9000 Power System Architecture

The discussion of the power system architecture will focus on some of the architecture's major features: power zoning, $N + 1$ redundancy, and decoupling.

- Power zoning enables parts of the system to be powered off for maintenance while the rest of the system remains operational.
- $N + 1$ redundancy provides higher perceived system availability to counteract the impact of low system mean time between failures, which is a result of the large number of regulators.
- Decoupling major sections of the power system allows future upgrades to be made without requiring significant changes to the rest of the system.

The basic power system architecture for the VAX 9000 Model 200 and Model 400 series is shown in Figures 1 and 2, respectively. Power processing in each model occurs in two distinct stages. First, an AC front end processes and converts AC utility input power to high-voltage DC, which is then bused about the power system. Second, DC-to-DC switching regulators convert the high-voltage DC to low-voltage outputs, which are then distributed through high-current-carrying busbars to the various logic loads. An intelligent power control subsystem (PCS) provides control, sequencing, monitoring, and diagnostic capabilities. Dedicated bias regulators, which are powered from the high-voltage DC, provide housekeeping control (i.e., low power) and start-up power to each bank of output regulators.

The high-voltage DC bus permits low-voltage output regulators to be added or removed for different system configurations. The high-voltage DC bus also can be backed up with a battery unit that produces high-voltage DC from 48-volt batteries through a step-up switching regulator. This approach allows any specific low-voltage output to be produced, as needed, during the battery backup period without using specific battery-to-logic voltage output DC-to-DC regulators. The battery required to backup the entire computer system would be larger than the computer itself. Therefore, diodes are inserted into the high-voltage DC distribution to partition the high-voltage DC bus, and only sections, such as the memory refresh operation and PCS control, are backed up.

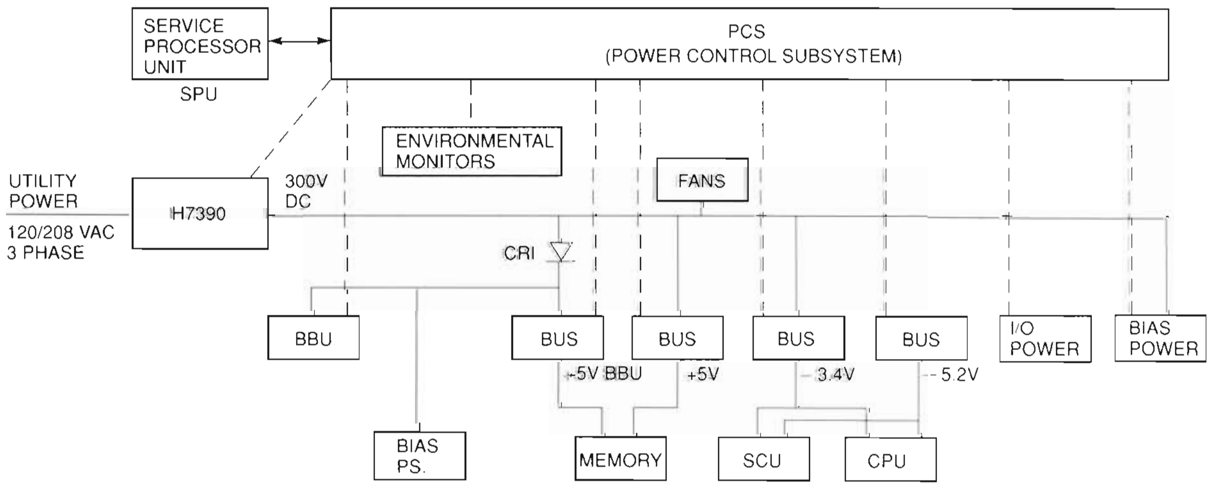


Figure 1 VAX 9000 Model 200 Series Power System

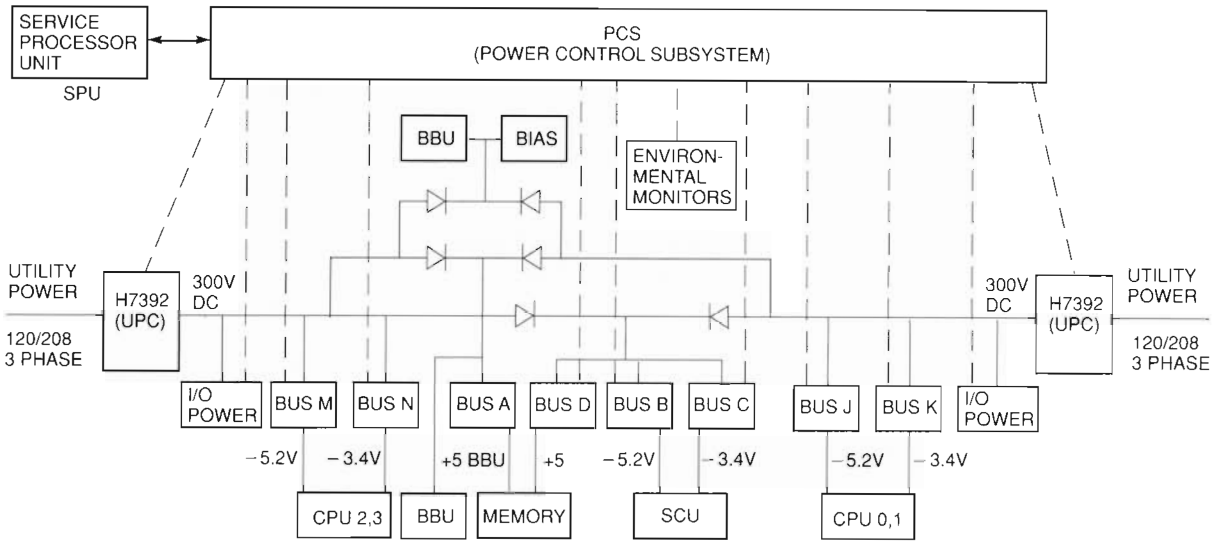


Figure 2 VAX 9000 Model 400 Series Power System

Power Zoning

The power-zoning feature meets the maintainability and high availability goals in the VAX 9000 Model 400 series of triple and quadruple processors. In the power system's configuration, a pair of dual processors can be powered off for maintenance, while the remaining powered-on processors maintain system operation.

A quadruple processor configuration is not composed of two identical dual processors. Some functions of a quadruple processor are not replicated. The system control unit, the memory, the service processor unit, and the PCS are common to both dual processors. Therefore, these functions are powered up by either front end. The high-voltage DC power bus is diode OR'd from either AC power source, through the dual diode, CR1, and then fed to the output stages that power the common elements listed above.

The diode-OR process in the VAX 9000 system does not provide for active loadsharing. Active loadsharing between each AC front end increases the overall actual power system reliability because it ensures that each AC front end supplies half the load. Otherwise, one AC front end could take most of the load (and be stressed higher), which would leave the other unit too lightly loaded. However, active load sharing is complicated by the physical distances between the AC front ends and the complex handling of faults and partial faults in each AC front end. The load of the common elements in the VAX 9000 system is only 20 percent of the total system. Therefore, the worst load imbalance does not justify the added complexity.

The diode does not have a significant impact on overall power load reliability because conservative derating of the diode results in a lower diode operating temperature and hence higher reliability.

We were concerned that power zoning could have an impact on the rest of the system as a result of powering down part of the system. However, analysis of the results showed that such a concern was unfounded. The high-voltage DC bus has relatively long time constants (i.e., slow to react to changes). Therefore, turn-on and turn-off transients on the bus are smooth and gradual and do not generate quick-changing electromagnetic fields that could affect the operation of the sections of the system that are still functioning.

N+*1 Redundancy*

Each processor in the VAX 9000 power system uses approximately 400 amperes from each of the two supply voltages. The ratings of the power semiconductors used in the outputs of the DC-to-DC regulators deliver an optimal regulator rating of approximately 240 amperes. Based on these ratings, powering a CPU in the VAX 9000 system would require two regulators for each voltage. However, in a large system, such as the VAX 9000 system, the number of regulators can quickly add up, which would result in an equally quick drop in overall system reliability. Powering two CPUs from the same voltage bus reduces the number of regulators.

Redundancy is then used to minimize the impact of the large number of regulators in the bus. By using redundancy, additional regulators on a voltage bus increase the perceived time between complete failures.

For example, consider a voltage bus that requires two regulators to supply the load current. A failure in either regulator causes a complete failure. If another parallel regulator is added to supply the load current, the probability of a complete failure significantly decreases. In this case, if one regulator fails, the other two could supply the load. The statistical probability that another failure would occur before the failed regulator is replaced is very small.

A system of N regulators at an individual failure rate of lambda (Λ) would have a system failure rate of N times Λ, or an MTBF of 1 divided by N times Λ.[1] The actual calculations are

$$\Lambda\,(\text{total}) = N \times \Lambda$$

or

$$\text{MTBF} = 1/\Lambda\,(\text{total}) = 1/(N \times \Lambda)$$

The failure rate calculation for a system that contains one regulator more than required ($N + 1$) is

$$\Lambda\,(\text{total observed}) = (N+1) \times N \times \Lambda \times \Lambda \,/\, [\{(N+1) \times \Lambda\} + (N \times \Lambda) + u]$$

$$\text{MTBF}\,(\text{observed}) = ([(N+1) \times \Lambda] + (N \times \Lambda) + u) \,/\, [(N+1) \times N \times \Lambda \times \Lambda]$$

It should be noted for the above equation, that u equals 1 divided by the time between fault and repair (service interval).

Using this calculation, if a bus required 4 regulators and each regulator had an MTBF of 400,000 hours, the observed MTBF would be 100,000 hours. The observed MTBF with five regulators (i.e., $N + 1$) would be 23,989,000 hours, which is 239 times longer than the four regulator case. The maximum time between the fault occurrence and repair would be 2 weeks, or 336 hours. The observed MTBF is so large, compared to other elements in the system, the redundant regulators have an extremely small effect on the overall reliability.

The number of redundant regulators per output voltage bus is limited to one in the VAX 9000 power system for space, weight, and cost reasons. N is the number of regulators required to supply the maximum current of a bus, and the addition of one more regulator is called $N + 1$ redundancy.

$N + 1$ redundancy relies on the good regulators on the output bus to pick up the load from the failed unit. This reliance has a significant impact on the design of the regulator, the regulator response time, and how the regulator handles the faults that can cause a failure. Fast regulator response (the time it takes to respond to a change in input or output) is needed to ensure that the output voltage does not dip too much when each regulator picks up its share of the load from the failed regulator. However, the faster response time makes it more difficult to keep the control functions of the unit stable. Moreover, the regulator input voltage range is designed to be relatively wide to tolerate wide swings in the high-voltage DC input.

When one regulator in a bank of regulators operated in parallel fails, the output bus voltage dips until the other regulators, which are connected in parallel, can react and pick up the load currents. The magnitude of the dip depends on the time the input fuses in each regulator take to open and on the values of the input capacitors and the distribution impedances.

Fast-opening fuses allow smaller voltage dips but are more prone to false nuisance openings. Slow-opening fuses do not open for normal or nuisance surges, but allow a greater voltage dip. Large values of input capacitance provide the energy to open the fuses quickly, but the voltage recharging of the capacitors is longer. A high distribution impedance decouples the faults from other units but has a high power loss.

Simulation and testing showed that the wide input range design of the regulators is sufficient to tolerate the high-voltage input dips caused by other faults. The regulator control and response time keep the low-voltage DC outputs within specification when the input voltage is within its range.

Other faults within the regulator can cause it to fail, but the load is picked up by the other regulators, operating in parallel, on the bus. Clearly, faults such as a permanent short on the output bus, cannot be survived. Because the low-voltage output regulators operate in parallel and in an $N + 1$ redundancy mode, the output voltage is not affected by most common single-fault conditions in the power system hardware.

Decoupling

A key feature of the power system's architecture is that each major subsystem is relatively decoupled from the other subsystems. Decoupling permits each subsystem to be designed for its own requirements and to be changed or upgraded as the requirements change (e.g., more cost effective, improved technology, or different output voltage), provided the interface and critical function remain the same. For example, two significantly different cost and performance options, H7392 or H7390, for the AC front end can be used in different configurations, and the rest of the power system does not need to be changed. Thus, power platforms can be flexibly tailored to meet the needs of different computer systems.

Achieving Low Harmonic Distortion

The AC front end of the VAX 9000 power system processes and converts public utility AC power to high-voltage DC. Our goal was to design the AC front end to be highly reliable, have a high availability, and meet the emerging European AC power quality standards. One of those standards is to have low harmonic distortion of the input AC current waveform. These features were essential to support the VAX 9000 system's entry into the mainframe computer market. We also decided to meet the low harmonic distortion standard of the AC front end because the European marketing and sales force viewed compliance with this standard as a distinct competitive advantage.

Design Factors

The dominating design factor for the AC front end was the size of the input power level, which was approximately 20,000 watts. This size significantly exceeded the power levels of previous AC circuit designs for a single unit. The high power consumption was a result of the use of 250,000 emitter-coupled logic (ECL) gates in the CPU and 512 megabytes (MB) of memory.

High Reliability and Availability To achieve high reliability, we used conservative power derating levels and good thermal management for key devices. Typically, the device voltage ratings used are 80 percent of rating. The main switches and rectifiers used in the power stages used 40 percent of rating. Current derating is also conservatively placed at 40 percent. Stress is lessened because of lower device function temperatures, which results in a longer operational life, which equates to higher reliability.

We designed two approaches to attain high availability. First, redundant circuitry was used for the AC-to-DC circuit function. Second, we increased immunity-to-line outage from the standard practice of one cycle of outage protection to ten cycles. The increase from one cycle to ten cycles of outage immunity provides the VAX 9000 system with a 300 percent improvement in mean time between observed system power outages over standard Digital systems. This feature improves system availability to the customer.

Harmonic Distortion The power system's design had to meet the increasing restrictions on the interface with the public power utility and be able to withstand the occasional availability of only poor power. Utility power is generated as a relatively pure (i.e., low harmonic distortion) sine wave. AC front ends and power supplies must convert this sine wave of voltage to a ripple-free DC voltage for ultimate consumption by the logic chips within the computer system. Standard methods used for this conversion create a nonlinear load on the sine wave of voltage. This nonlinear load distorts the utility's sine wave of voltage for other users, because of the distribution system impedance, and usually appears as interference for other users. In Europe, the occurrence of this type of interference is planned to be limited by restricting how much nonlinear load current an AC front end can have. Therefore, we had to design a unique circuitry that could convert AC power to DC power at 20,000 watts without high levels of current distortion to meet this European requirement.

A design based on commercially available control technology could not meet the stringent technical requirements of high overall conversion efficiency and stability of operation because conventional AC-to-DC circuitry produces up to 30 percent distortion. Our goal was to comply with emerging European requirements of harmonic current distortion levels in the 5 percent range. However, at the time we were designing the system, no circuitry at this power level existed in the power conversion industry. Therefore, we had to develop a unique pulse-width modulator (PWM) circuit and control equations for the input power conversion stage, which is shown in Figure 3.

The pulse-width modulator combines the advantages of low switching frequency, which reduces switching losses in the converter, with exceptionally short response time to all input line voltage disturbances and to rapid changes in the required computer power. The final design produces less than 5 percent total harmonic distortion of the input line current when the UPC is operated at 20,000 watts load. The uniqueness of the PWM increased the immunity-to-line voltage outages from one cycle of outage protection to ten cycles. Furthermore, the increase was achieved without a corresponding tenfold increase in storage capacitors.

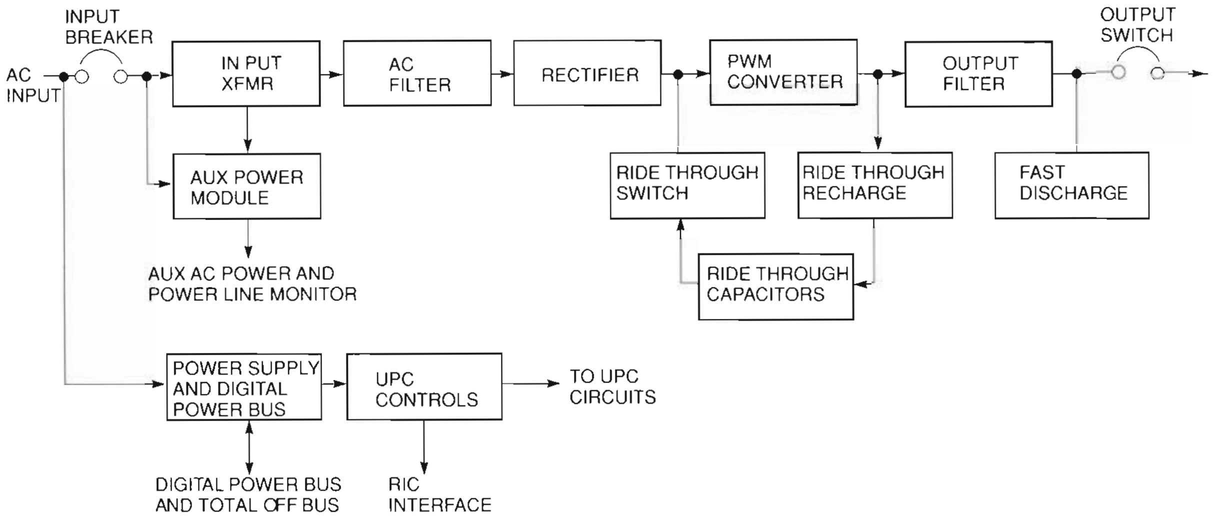


Figure 3 UPC Block Diagram

Flexible Line Cord

The high power level and the requirements for a flexible line cord and plug required that the Underwriters Laboratory (UL) and Canadian Standards Association (CSA) agencies expand the regulations that governed the size of power cordage allowed in a computer room. A flexible line cord connected to the AC service is a requirement by Digital for all its products. This feature is deemed valuable because it is used both to facilitate the initial installation of the computer and possible relocation at the customer's site. Although delays can occur while waiting for a national agency to amend one of its national regulatory codes, the approvals were received in time to maintain the project's schedule.

Improving Load Sharing

Detailed stress analyses show that when regulators are operated in parallel, maximum reliability is achieved when the load current is shared equally among them.

Traditional Approach

A traditional approach to running regulators in parallel may be seen in VAX 8000 series machines. In these processors, regulators that are designed for standalone operation are placed in a parallel configuration. Current sharing is forced by modifying each supply's individual reference voltage through external monitoring and control. In the case of VAX 8000 machines, a maximum of four units may be coupled in this way. Figure 4 shows that this method essentially uses equipment that was designed to function as standalone regulated voltage sources. By adding external control loops, the equipment is forced to provide identical output voltages, as measured at some defined point in the system. If precise voltage matching is not achieved, whichever supply had the higher voltage consumes the load, up to its overcurrent sense point. Thus, equal load sharing cannot happen. Individual external controllers are required for each converter, which makes the system more complex. The VAX 9000 system requires up to five converters per bus, and we could not achieve better than 20 percent power sharing between modules by using this method. No traditional methods could support the number of converters in the VAX 9000 system. Also, most methods had a master-slave relationship that precluded maximizing a regulator's reliability potential.

New Approach

As a result of the limitations of the traditional methods, we developed a new, less complex approach to current sharing between parallel converters. Although developed specifically for the VAX 9000 program, the features and utility of this approach have universal application. The essential technological shift from prior practice is that in this system the regulators are current sources rather than voltage sources.

We designed the current sources to have a compliance range that covers a band of voltages that are

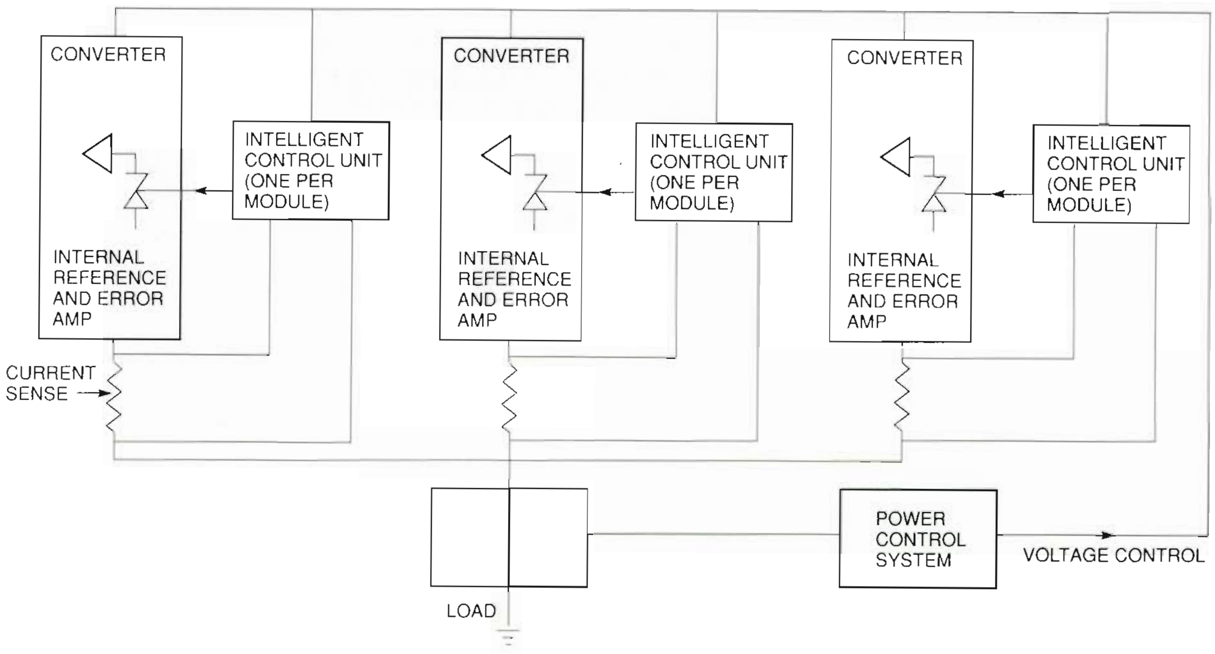

CONVERTER
INTELLIGENT CONTROL UNIT (ONE PER MODULE)
INTERNAL REFERENCE AND ERROR AMP
CURRENT SENSE
POWER CONTROL SYSTEM
VOLTAGE CONTROL
LOAD

Figure 4 Load Sharing by Voltage Control of Voltage Sources

normally found in logic circuits. By making the regulator outputs fully floating, the VAX 9000 system requirements for +5-volt, −3.4-volt, and −5.2-volt buses are met with only one regulator design, rather than a separate design for each voltage. The VAX 9000 design is simpler and has a lower manufacturing cost. The regulator is voltage and polarity "blind" over its compliance range, and any number of regulators may operate in parallel to provide any amount of power required at any voltage within the compliance range. Also, this method automatically compensates for the effects of stray resistances and different path lengths from individual regulators on a bus.

The basic features of this new approach are shown in Figure 5. Individual regulators behave as externally programmed current sources controlled by a common control signal, such that each regulator delivers the same current. If the outputs are connected to a common load, the current in that load is the sum of the individual regulator output currents. The resulting voltage that appears across the load is the product of that current and the equivalent resistance of the load. Furthermore, if that voltage is compared with a reference voltage in a conventional error amplifier and the resulting error signal is used to derive the regulators' external programming source, then a voltage control loop exists around the regulator system. Thus, although each regulator acts as a current source, the system acts as a controlled and regulated voltage source. Because the voltage control loop only contains one pole, the bandwidth of the control loop can be increased by up to a factor of at least 15. As a result, the substantially high current change requirements imposed by high-speed memories, such as those used in the VAX 9000 system, can be accommodated.

Principle of Operation

A two-transistor forward regulator is shown in Figure 6. In this regulator, S1 and S2 are switched

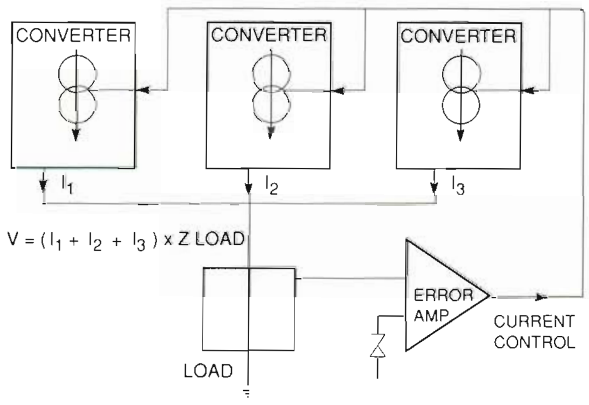


Figure 5 Load Sharing by Current Control of Current Sources

into conduction simultaneously, which causes the current to flow in the primary winding of transformer T1 at a level that is directly proportional to the output current Iout plus the slope of the current due to Lout. This current also flows in the primary winding of current sense transformer T2. The resulting current that flows in T2 secondary winding develops a voltage across the load resistor, RL, which is amplified in A1 and applied to the input of comparator C1. Therefore, at this point, a voltage pulse appears, the amplitude and shape of which are directly proportional to the current flowing in the output choke Lout during the S1-to-S2 conduction period.

A conventional reference source/error amplifier combination is placed across the output of the supply. The resulting error signal, called Vcontrol, is applied to the other input of comparator C1 as a DC level. The comparator is followed by gating and drive circuits to the power switches.

Switching is initiated by a pulse within the gating circuit that drives the power switches on. The current flows in the output choke, Lout, and a proportional voltage appears at the output of the amplifier A1. As this voltage ramps, it crosses the threshold set by Vcontrol at the C1 input. The comparator output then changes state and causes the drive pulse to the switches to cease.

If Vcontrol were a fixed value, the system would be a constant current source. Therefore, the voltage that would appear at its output would be the result

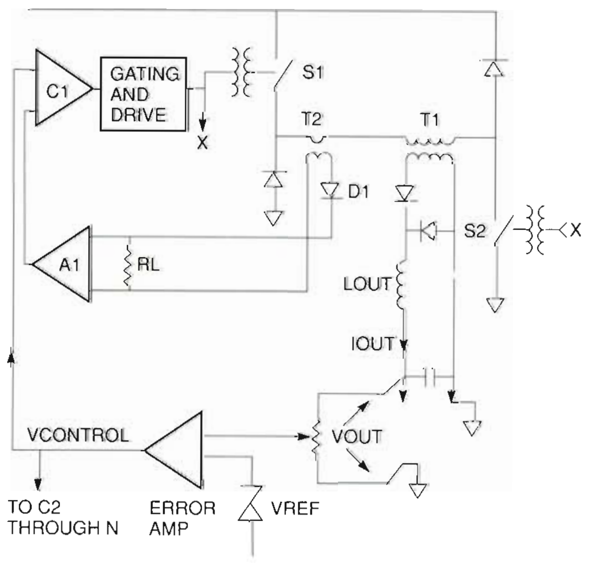


Figure 6 Two-transistor Forward Regulator

of that constant current, and whatever load is placed across those terminals (i.e., Vout) would be determined by the load value. By using an error amplifier and reference, Vcontrol can be made a variable quantity. Therefore, the regulator transfer function can control its output current to any level necessary to produce the desired voltage. In such a system, a control voltage, which is derived from a single error amplifier and reference, can be used as the control input for several regulators that are running in parallel. Thus, the current from multiple regulators that feed a common bus can be shared.

Increased Control and Monitoring

In the VAX 8000 series, power and environmental monitoring and control is provided by the H7188 environmental monitoring module (EMM). In the VAX 9000 system, these functions are provided by the power control system (PCS).

Basic Design of EMM and PCS

The EMM monitors the DC-to-DC regulator control, air flow sensor, and cabinet temperature. It is also the interface between the system console and the power system. Conceptually, the EMM functions as a peripheral device to the console similar to the way an intelligent disk controller is a peripheral to a CPU. The EMM is a single module that plugs into a power back panel.

The PCS is a distributed data acquisition and control system. It also interfaces between the power and environmental systems and other parts of the computer system. The PCS takes commands from, and reports status changes to, the service processor unit.

However, in the PCS, the conceptual model of the EMM is extended to provide additional support in hardware and firmware to off-load the service processor unit and to simplify the software interface to the PCS. The PCS includes many features that enhance testability, fault coverage, fault isolation, and system availability. The relationship of the PCS modules to one another and to other system components is illustrated in Figure 7. There are five PCS modules:

- Power and environmental monitor (PEM)
- CPU regulator intelligence card (CPURIC)
- I/O regulator intelligence card (IORIC)
- Signal interface panel (SIP)
- Operator control panel (OCP)

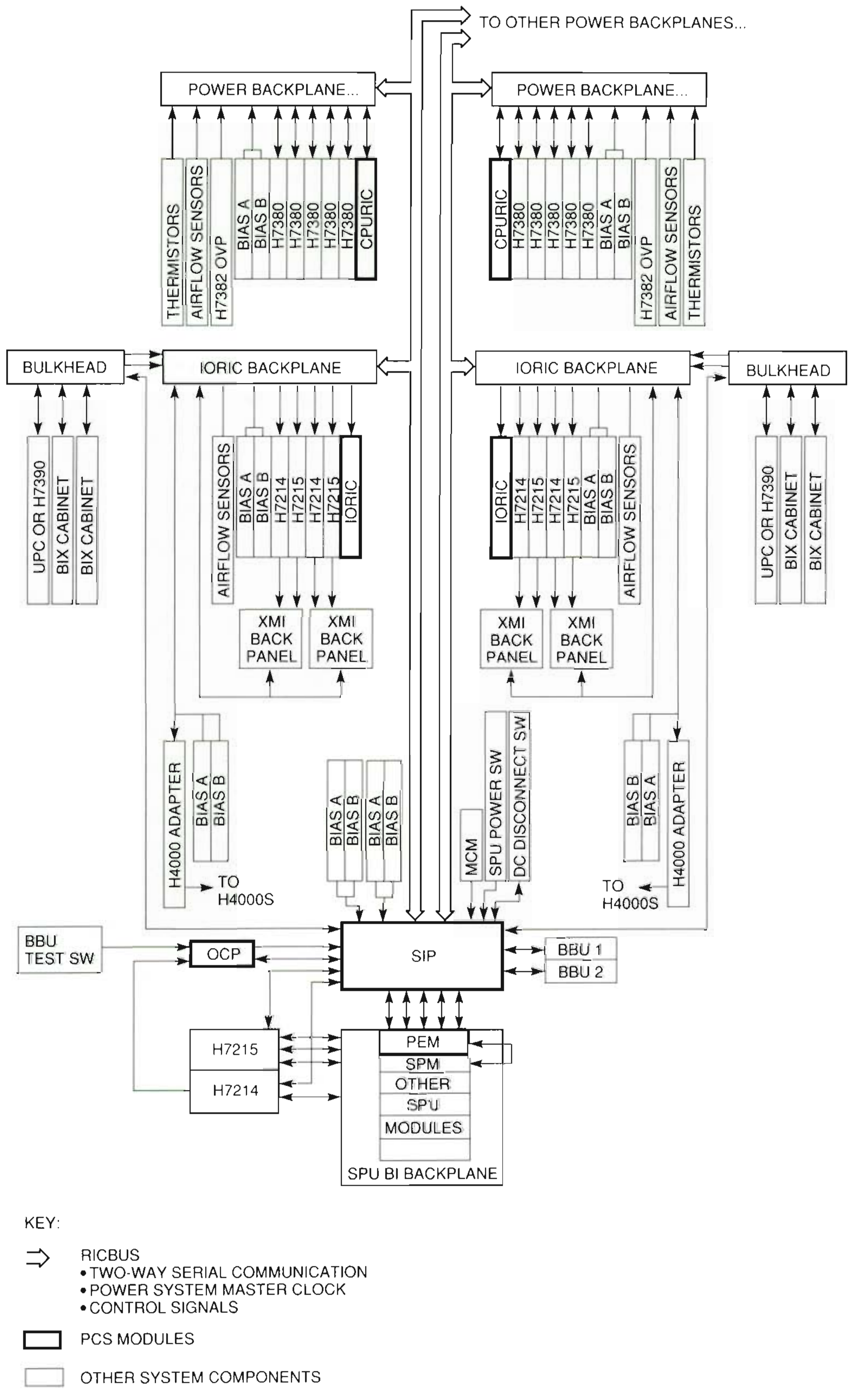


Figure 7 PCS Block Diagram

Comparison of PCS and EMM

The differences between a power system that uses an EMM and one that uses the PCS are illustrated in Table 1, which details the functions of each system. Five-to-ten EMMs would be required to control and monitor a system of the size and complexity of the VAX 9000 system. Additional modules would be required to support some of the functions provided by the signal interface panel and PEM module.

Command Set Enhancements In comparison to the EMM, the PCS offers an enhanced command set. The PEM commands of READ, WRITE, BIS, BIC, and MEASURE provide the same capabilities that the seven EMM commands READ, WRITE, BIS, BIC, MEASURE, EXAMINE, and DEPOSIT do. In addition, the PEM supports six commands that are not implemented by the EMM command set: DOWNLOAD, MARGIN, SENSE, POWERON, POWEROFF, and PASSTHRU.

The regulator interface card firmware supports the DOWNLOAD command, which allows the service processing unit's software to update, with some restrictions, the PEM's or regulator interface card's on-board EEPROM with new firmware. Thus, the need for Customer Services to replace EPROMs in the field if the firmware needs to be updated is reduced because the latest PCS firmware is stored on the service processor unit's load device.

The MARGIN, SENSE, POWERON, and POWEROFF commands off-load work and complexity from the service processor unit's software. By using these commands, the service processor unit never needs to interact directly with the regulator interface card modules during normal system operation. Thus, the amount of software required by the service processor unit to perform these functions is reduced. All regulator interface card interaction is handled by the PEM firmware.

The MARGIN command causes the PEM to margin the specified bus voltages by ± 5 percent for fault isolation purposes, such as trying to aggravate an intermittent CPU hardware problem by reducing a logic supply voltage by 5 percent. In response to the SENSE command, the PEM returns a record that contains the specified power or environmental data to the service processor unit.

The POWERON and POWEROFF commands cause the PEM firmware to turn the specified power buses on or off in the proper sequence. When executing all of these commands, the PEM firmware must send messages to one or more regulator interface card modules and perform extensive error checking to verify that the power sequencing is proceeding correctly. The PEM then returns a status byte, which describes any error that occurred during command execution, to the service processor unit.

The PASSTHRU command allows the service processor unit's software to send commands directly to the specified regulator interface card modules. This command bypasses the PEM and allows the operator to use other PCS fault isolation functions that are not used by the service processor unit in normal operation. The PASSTHRU command is used for fault isolation purposes only and is not required for normal system operation.

Measurement Accuracy The EMM's best measurement of accuracy is ± 54 millivolts, which is achieved when it is using an 8-bit analog-to-digital converter. The CPURIC measurements are substantially more accurate and repeatable for several reasons. The CPURIC uses a 12-bit analog-to-digital converter that is calibrated for offset and gain by the automatic calibration routines that run during power-up self-test. To filter out noise, each parameter is measured 64 times. These measurements are averaged by the firmware before the parameter is used by the monitoring or sense commands.

Through comparison measurements with a voltmeter and a thermometer, the CPURIC measurements have proven to be repeatable. Also, the measurements are accurate to better than 10 millivolts, when measuring voltage, and to within one degree Celsius, when measuring temperature.

Diagnostics and Testability Support The EMM provided some visibility into the power and environment system of the VAX 8000 series of computer systems to aid diagnostic and testing. The PCS hardware and firmware extend the functionality of the EMM with features such as hardware loopback circuitry which, when combined with diagnostics included in the firmware, provide better fault detection and isolation than the EMM.

Enhanced Support for Increased System Availability The features designed into the power system and the PCS hardware and firmware support $N + 1$ power buses and bias power supplies. The PCS also supports the partitioning of power. The PCS allows certain cabinets in a VAX 9000 Model 430 or Model 440 system to be powered-off for maintenance or repair, while the remainder of the power system continues to function to provide system availability at reduced performance. The PCS recognizes when power is reapplied to these cabinets and notifies the service processor unit. The system then can be reconfigured to include these cabinets.

Table 1 Comparative Functions of the Environmental Monitoring Module and the Power Control System

EMM in the VAX 8600 System	PCS in the VAX 9000 Model 440
Digitally controls seven DC-to-DC regulators configured in five buses	Provides analog and digital control of up to 29 H7380 DC-to-DC regulators, configured in eight power buses
Measures and monitors four cabinet air temperature thermistors	Measures and monitors ten cabinet air temperature thermistors
Monitors two air flow sensors	Monitors 20 air flow sensors
Controls and monitors one H7231 battery backup unit	Controls and monitors two H7231 battery backup units
Measures and monitors one ground current input	Measures and monitors two ground current inputs
Provides voltage sequencing in hardware	Provides voltage sequencing in hardware and software
Displays up to 16 unique shutdown codes on four magnetic indicators	Displays over 80 unique shutdown codes on a diagnostic display
Measurement accuracy voltage: ±54 millivolts temperature: ±2 degrees Fahrenheit	Measurement accuracy voltage: ±10 millivolts temperature: ±1 degree Fahrenheit
Digitally controls ±5 percent voltage margining for eight DC-to-DC regulators	Provides analog and digital control of ±5 percent voltage margining for eight power buses
Measures and monitors 12 DC-to-DC regulator voltage outputs	Measures and monitors eight power bus voltage outputs
Monitors ten DC-to-DC regulator "module OK" signals	Monitors 29 H7380 DC-to-DC regulator "module OK" signals
	Monitors 36 H7382 bias power supply "module OK" signals
	Monitors up to ten H7214 and H7215 "module OK" signals used for I/O and service processing unit power
	Monitors up to 16 H7189 "module OK" signals in the optional bus interface expansion cabinets
	Provides bus overcurrent protection and monitoring for eight power buses
	Measures the output current from 29 H7380 DC-to-DC converters
	Provides $N+1$ support for eight power buses
	Monitors the environmental status from four optional bus interface expansion cabinets
	Monitors the status from three H7386 overprotection modules
	Monitors seven status lines from each of the two utility port conditioners
	Controls and monitors the operator control panel
The VAX 8600 system consists of two cabinets of which one was monitored by a single EMM.	The VAX 9000 Series 440 is a quadruple CPU configuration of up to eleven cabinets. A PCS configuration composed of eight CPURICs, two IORICs, one operator control panel, one signal interface panel, and one PEM is required to support this system.

Design for Further Improvement The EMM uses the actual analog-to-digital converter to represent temperature, voltage, and current. However, the PCS represents voltage, temperature, and current in a format that is independent of the actual analog-to-digital converter values. Future upgrading of measurement circuitry can be done without modifying the service processor unit's software.

Power System Test Programs We developed extensive power system test programs by using the programmable console command language. These scripts provided step-by-step control of power sequencing and margining, and proved extremely invaluable in processor system debugging, system qualification, and manufacturing and field testing. The tests were developed through a cooperative effort of design engineering, manufacturing engineering, and field engineering.

Simulation

The use of simulation in power converter design is not as advanced as the use of simulation tools in digital circuits. The level of complexity and number of parasitic elements in power devices have pushed computer CPU requirements beyond the reach of many power circuit design groups. However, as more computer power is becoming available at a lower cost, simulation is being used increasingly to improve power circuit design. The simulation tool most widely used is Simulation Program with Integrated Circuit Emphasis (SPICE) because of its ability to be configured to any circuit configuration.[2]

In this section, we illustrate the benefits of simulation in the VAX 9000 power system design. We provide examples of the use of simulation for correcting designs, improving circuit designs through inclusion of parasitic elements, and transient analysis.

Simulation to Correct a Design

Simulation was used to correct a design in the linear post regulator that was developed for the H7382 bias supply used in the VAX 9000 power system. The design required that two regulators operate in parallel for redundancy purposes. We wanted to achieve good transient response by keeping the output voltage within operating tolerance should one of the two regulators fail. Because good transient response depends on good frequency loop response, we had to determine the optimum frequency response for the circuit.

Because simulation models for many of the circuit components were not yet available, we could not simulate the design. Therefore, we built the circuit without simulation. The resulting frequency response was lower than expected, and the circuit tended to oscillate at the maximum output current limit. Multiple attempts to improve the hardware proved ineffective and time-consuming because we did not know the cause of the problem. Then, the actual schematic of the linear regulator controller internal circuit became available, as did SPICE models of components.

We ran an accurate SPICE model but did not find anything outstanding on the gain/frequency plots. Next, we tried to find the cause by exaggerating some simulated changes, such as removing the current limit amplifier portion of the circuit from the controller. With this change, we found that the gain was close to being the same at two different frequencies, 5 kilohertz (KHz) and 40 KHz. This similarity meant that if the phase margins were correct, instability might exist. To prevent this possibility, we decided to increase the gain of the regulator circuit below 30 KHz by making simulated modifications to the circuit. With these modifications, the gain plot below 30 KHz increased and the waveform evened out close to what we wanted it to be. We then modified the hardware and achieved the desired performance. However, we would have saved a substantial amount of time if we could have simulated the circuit before we built it.

Improving Simulation Accuracy

In switching regulator design, parasitic (small, undesirable but existing) elements of seemingly negligible values, such as printed circuit board etch inductances and transistor capacitances, can have a significant impact on the behavior of the circuit. For accurate simulation these elements must be included in the simulation models. An example is shown in the design of the output stage of the H7380 regulator.

We wanted the regulator to take a high-voltage DC input and produce a low-voltage (i.e., 3.4-volt DC to 5.5-volt DC) regulated output. Figure 8 shows how this process is done by changing the high-voltage DC input voltage to an AC square wave through turning the transistors, Q1 and Q2, on and off. The transformer, T1, steps down the AC square wave and is followed by an output section for rectification and filtering.

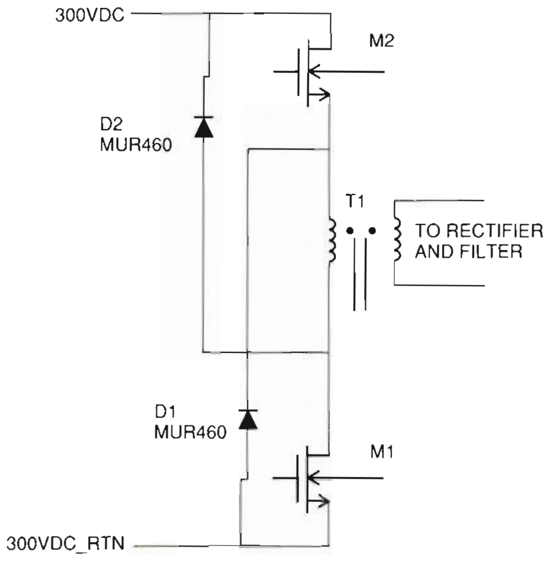


Figure 8 H7380 Output Switching Stage

The initial model of the H7380 inverter stage used simple component models and did not consider any printed circuit board inductances or transistor capacitances because they seemed negligible compared to other elements. We noted a discrepancy in the voltage across the transistor Q1 (Vds) during the turn-off process between the simulated waveform, shown in Figure 9, and the measured waveform, shown in Figure 10.

Figure 9 shows that the voltage is initially zero while the transistor is conducting but rises to 200 volts when the transistor is turned off. Figure 10 shows that ringing occurs as the voltage approaches 200 volts, with an overshoot to 240 volts. The ringing and overshoot, not shown in Figure 9, are caused by the circuit board inductance, transformer leakage inductance, and the capacitance of the transistor.

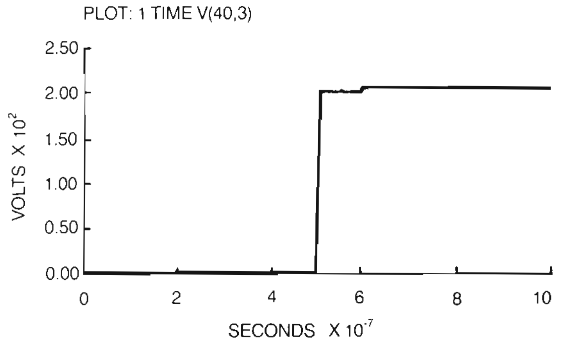


Figure 9 Vds (Q1) Simulated Turnoff without Parasitics

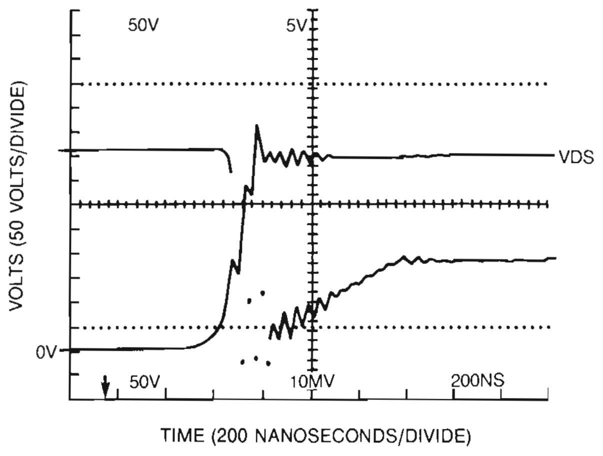


Figure 10 Vds (Q1) Measured Turnoff

Figure 11 shows a more accurate model of the output stage because the L1 through L4 etch inductances and C1 and C2 transistor capacitances are included. The current source, IPULSE, and the resistor, RT, approximate the transformer. Figure 12 shows the result of the simulation model that includes the L and C values shown in Figure 10.

When the simulation and the measured data are correlated, the advantage of accurate simulation becomes apparent. By using worst-case values for the circuit parameters, the simulation can determine the maximum peak voltage. The model depicted in Figure 12 shows that a device capable of withstanding the expected 240 volts is needed. Reliance on a less accurate model without parasitics could lead to the selection of a device capable of withstanding only 200 volts. Thus, accurate simulation allows the correct components and component ratings to be chosen and ensures a robust design.

Transient Analysis

A memory system that includes dynamic random-access memory (RAM) chips presents a difficult transient load problem to its power supply. The problem arises from a combination of very high changes in dynamic RAM supply current and current change rise times that are typically more than a thousand times faster than the reaction time of a power system. The result is a temporary change in the load supply voltage. To handle these fast current edges, high-frequency capacitors are mounted on memory boards near the dynamic RAMs. Also, low-frequency, electrolytic capacitors, which provide a source of local charge storage, are mounted on the

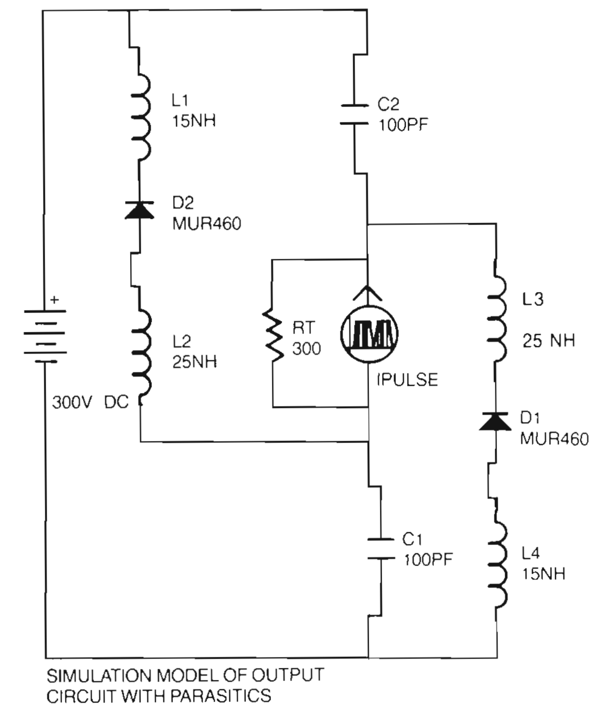


Figure 11 Final Model of H7380 Output Switching Stage

memory boards to handle the magnitude of the change. The capacitors help keep the supply voltage within its operating range until the power supply can react and sufficiently change the current it supplies to the memory to stabilize the supply voltage. An adequate supply design with specified capacitors can keep the supply voltage within its operating tolerance. Simulation is used to determine the correct mix of high and low frequency capacitors and the number of regulators required to support this high transient load.

Another power supply problem arises from the use of $N + 1$ redundancy for parallel regulators. When one of the regulators in a parallel regulator configuration fails, the remaining regulators must be able to take on the load from the failed regulator and keep the supply voltage within operating tolerance. Because the remaining regulators cannot react instantaneously, the load voltage drops until a sufficient increase in current can be provided by the remaining regulators.

For the VAX 9000 series memory system, a proposed dynamic RAM power supply design consisted of three H7380 DC-to-DC regulators, which would operate in parallel (including $N + 1$ redundancy) and be connected to the memory through power distribution busbars. The numbers of high- and low-

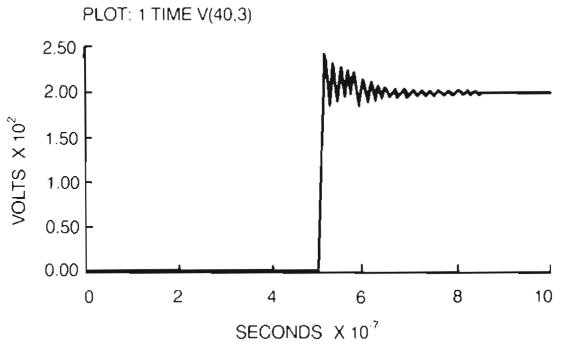


Figure 12 Vds (Q1) Simulated Turnoff with Parasitics

frequency capacitors were also proposed. The power supply was expected to be ready for load testing before the memory or the busbars would be available. Therefore, we had to verify that this design could keep the memory supply voltage within operating tolerance. We verified the design by simulating the performance of the power system and measuring the performance of the actual power supply with a simulated load.

Power Supply Operating Voltage Tolerance The memory designers specified the operating tolerance of the dynamic RAM supply as +5 volts, ± 10 percent. Using 10 percent as the supply tolerance budget, the supply designer made the allocations shown in Table 2 to all the factors that would cause the load voltage to deviate from its nominal value of +5 volts. As can be seen from this table, the sum of x and y must be less than 350 millivolts or 7 percent of +5 volts.

Memory Load The dynamic RAM supply current was calculated to be a steady-state pulsed current of 256 amperes that would last for 92 nanoseconds (ns) and with rise and fall times of 20 ns, as shown in Figure 13. The initial pulse magnitude was 1024 amperes.

Table 2 Supply Tolerance Budget Allocation

Causes of Voltage Deviation	Millivolts	Percentage of +5 Volts
Regulator tolerance	100	2
Back panel distribution	50	1
Transient load with two regulators	x	
Failure of one regulator	y	
Total deviation budget	500	10

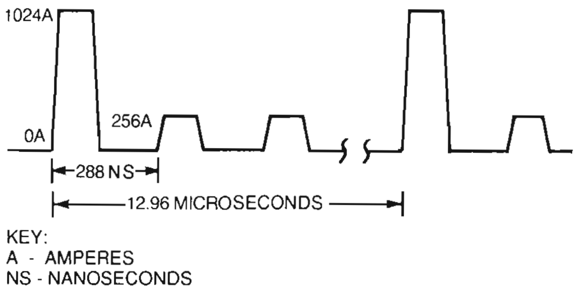


Figure 13 VAX 9000 Model 400 Series Memory Power System Dynamic RAM Load

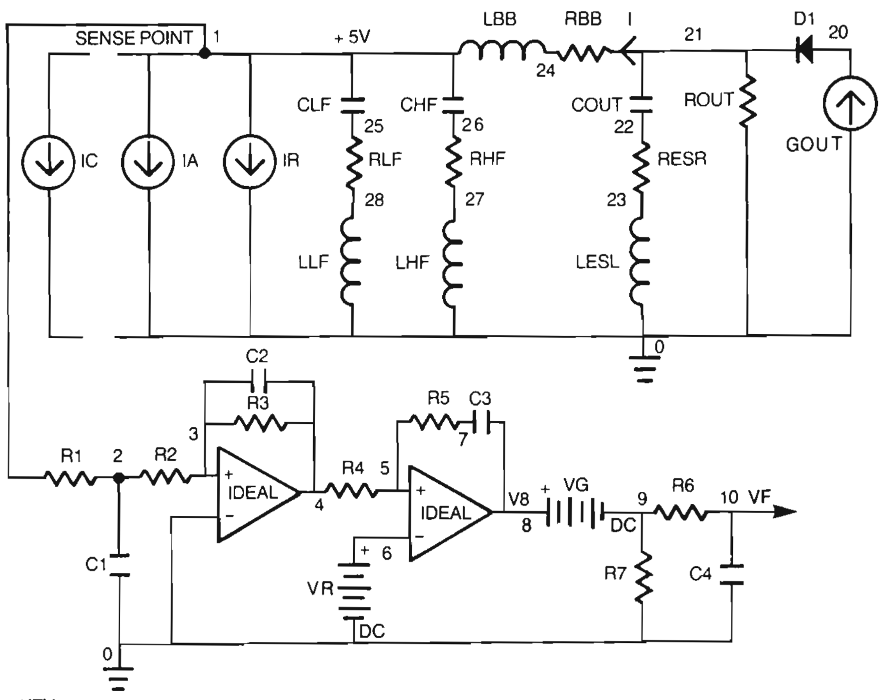
Memory Power System SPICE Model In the SPICE model of the supply, busbar, load and capacitors that is shown in Figure 14, the three regulators are modeled as a current source, Gout, controlled by the regulator feedback voltage, Vf. Cout and Rout represent the regulators combined output capacitors and resistors. Most of the other elements in the model are determined from component specifications. The relationship between Gout and Vf was determined by laboratory measurements on a regulator and resulted in the following equations. For two regulators,

$$Gout = 339 \times Vf = 339 \times (V8 - 2.5)$$

For three regulators,

$$Gout = 678 \times Vf = 678 \times (V8 - 2.5)$$

The load is represented as two current sources, IA and IR, the characteristics of which were obtained from the loads shown in Figure 13.



```
KEY:
R1      1   2   10K                    COUT   21  22  12300U  IC=5.0
C1      2   0   0.6N IC=2.5            RESR   22  23  1M
R2      2   3   10K                    LESL   23  0   2.4N
R3      3   4   20K                    RBB    21  24  300U
C2      3   4   18P  IC=5.0            LBB    24  1   150N
R4      4   5   1K                     CHF    1   26  1.3M
VR      6   0   DC 5                   RHF    26  27  21U
R5      5   7   2K                     LHF    27  0   1.4P
C3      7   8   68N  IC=3.0            CLF    1   25  108.8M
VG      8   9   DC 2.5                 RLF    25  28  400U
R7      9   0   10MEG                  LLF    28  0   0.3N
R6      9   10  10K                    IA     1   0   PULSE  0  512 A   0 NS
C4      10  0   0.757N                  20 NS  20 NS  92 NS  288 NS
GOUT    0   20  POLY(1)  10  0  0  678 IR     1   0   PULSE  0  512 A   0 NS
D1      20  21  DIODE                   20 NS  20 NS  92 NS  12.96μS
ROUT    21  0   17K
```

Figure 14 SPICE Model of VAX 9000 Memory Power System

Simulation and Laboratory Measurements The two previously stated conditions of interest resulting in large load voltage changes are the transient load with two regulators and the failure of one regulator.

For transient loads, a larger voltage change occurs with two regulators rather than with three because two regulators take longer than three to adjust the supply current to the new load value.

Simulated Load For laboratory measurements, the actual dynamic RAM load, as shown in Figure 13, is difficult to design and build in a reasonable time because of the magnitude and rise time combination. However, a load with a much slower rise time could be easily built. Such a load, (I in Figure 14) is expected through the busbar as the capacitors and busbar slowed down the fast edges of the dynamic RAM load. This simulated load was built and connected to two regulators. The predicted waveform and the measured waveform showed that the initial shapes of the peak change, the peak magnitudes (80 millivolts), and the times of occurrence of the peak (300 microseconds) were all similar. However, we could not measure the overshoot and ringing after the peak because the busbar was not available.

Failure of One Regulator When one of the three regulators fails, the other two regulators cannot meet the increased load instantaneously. As a result, the load voltage drops until the two regulators can increase their output current sufficiently to reverse the direction of the drop. The SPICE model for this condition was run and the load voltage of the drop was predicted. Laboratory measurements were then taken with the simulated load and one regulator was turned off. Both the predicted and measured waveforms had the same shapes, peak magnitudes (100 millivolts), and times of occurrence of the peak (200 microseconds) after the regulator was turned off. Therefore, we concluded that the proposed design could meet the load requirements.

References

1. P. O'Connor, *Practical Reliability Engineering* 2d ed. (New York: John Wiley and Sons, 1985).
2. SPICE is a general-purpose circuit simulator program developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Donald F. Hooper
John C. Eck

Synthesis in the CAD System Used to Design the VAX 9000 System

The design of the VAX 9000 system represents a sixfold increase in complexity over the VAX 8600/8650 system. This increased complexity posed a significant challenge because of the concurrent need to shorten the duration of the project design cycle and convert all high-performance systems computer-aided design (CAD) software from the DECSYSTEM-20 system to the VAX system. As part of the task of meeting these challenges, the CAD Group proposed the implementation of a design methodology that used logic synthesis for the first time in the development of a major product for Digital. The primary objectives of this methodology were to increase the productivity of the logic designers and to reduce the number of errors introduced during conversion of high-level designs into gate-level structural designs.

Methodologies

Previous Methodology

In the previous development methodology, as shown in Figure 1, logic designers specified high-level designs on paper, and simulation engineers transferred this rendition into a behavioral model. Technology engineers developed the gate-level cells. After the cells were defined and characterized for function and timing, the logic designers generated schematic drawings by using graphical bodies that represented the cells.

As changes were made to the schematics, the simulation engineers attempted to reflect these in the behavioral model. Finally, a gate-level simulation model was assembled from the completed schematics to verify that the design represented a valid VAX system. This process was extremely laborious, error-prone, and time-consuming. Therefore, we concluded it could not be used to develop the VAX 9000 system, which is a 700,000 gate design and for which the technology cells would not be defined and characterized until late in the design stage.

Logic Synthesis

Our early research into logic synthesis began in 1982. Over the next two years, we explored new synthesis ideas and constructed prototypes to determine the feasibility of those ideas. For example, one of our early logic minimization efforts was a program that emulated Brown's *Laws of Form* for transformations of Boolean logic to reduce gate counts and improve critical timing paths.[1] However, this program has had only limited success and is not really usable as a released computer-aided design (CAD) product. For example, the program does not deal with selections of cells for combinational logic nor does it consider the myriad problems involved in assembling a database for a buildable gate array chip.

During 1984 and 1985, new artificial intelligence (AI) and synthesis ideas were being developed. Universities and technical communities were exploring the potential of object-oriented databases, rule-based AI, data flow design entry, and algorithmic minimizations. We began the prototype development of our system for integral design (SID) at approximately the same time as the ideas for the VAX 9000 hardware architecture were beginning to be developed. In 1985, the SID program became an internal CAD product for use in the development of the VAX 9000 system. By combining the most advanced rule-based AI techniques with an object-oriented database, the core SID was designed to be a repository of logic design knowledge. We hoped that, over the years, SID would mature to perform many highly repetitive logic design tasks at an expert level.

From 1985 to 1988, the capabilities of the SID system gradually improved until it was producing gate array chips that met the VAX 9000 machine cycle time, power, and electrical rules requirements.

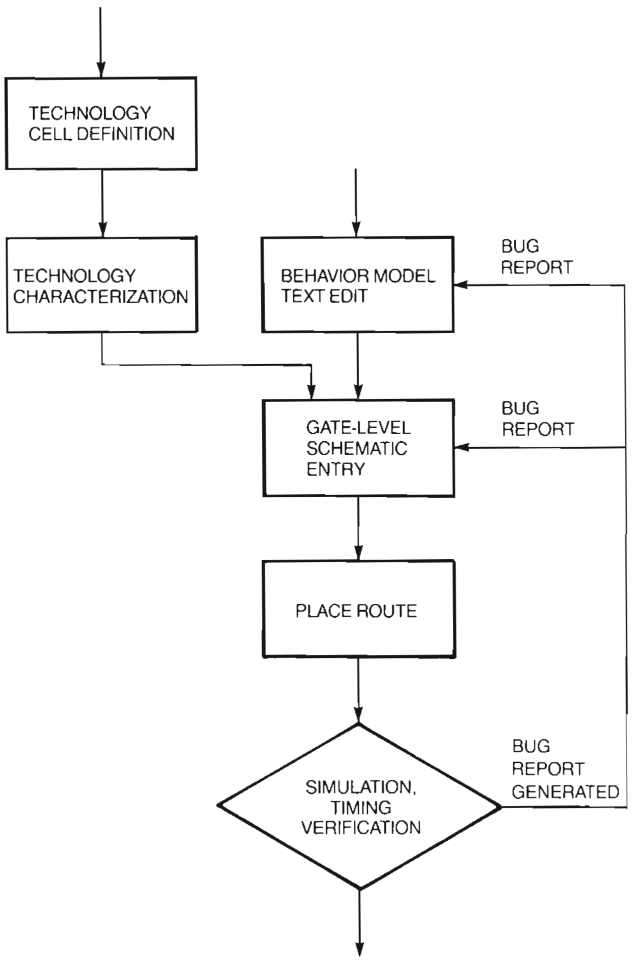

TECHNOLOGY CELL DEFINITION
TECHNOLOGY CHARACTERIZATION
BEHAVIOR MODEL TEXT EDIT
BUG REPORT
GATE-LEVEL SCHEMATIC ENTRY
BUG REPORT
PLACE ROUTE
SIMULATION, TIMING VERIFICATION
BUG REPORT GENERATED

Figure 1 Previous Design Methodology

New Methodology

The VAX 9000 development methodology, shown in Figure 2, circumvents the need to wait for the technology cells to be completely specified before beginning logic design. This methodology uses schematic entry and simulates the technology-independent, register transfer level (RTL) bodies.

The RTL library for this type of entry includes MUXes, latches, adders, comparators, incrementers, decoders, and simple Boolean gates. The entry is extracted to a common database format, called CADEX, from which a simulation model is built. A behavior model still exists, but its hierarchy matches the RTL schematic hierarchy at key physical boundaries. Thus, simulation models can be built that consist of a hierarchy of mixed behavior and RTL models.

While logic designers are creating the RTL design, technology engineers are defining the technology cells. In parallel with these activities, synthesis knowledge engineers are writing rules to transform the RTL design into technology cells. These three activities should be completed at the same time, at which point, synthesis produces each of the VAX 9000 system's 77 gate array chips. The goals for the synthesis program were to

- Simplify design entry and thereby reduce schematic complexity by a factor of 4
- Generate 90 percent of the VAX 9000 system's logic through synthesis
- Reduce the number of simulation errors introduced in the design
- Reduce the number of electrical rules violations in the design

To generate a database for a buildable gate array chip, the synthesis tool is required to

- Read technology-independent input standard net list format, which can be in DECSIM behavioral notation or CADEX common database format
- Minimize Boolean gates through state-of-the-art minimization techniques
- Improve timing-critical paths through Boolean transformations, cell/pin selections, power settings, and net load allocations
- Choose the best available technology cells based on timing, size (area), and power estimates
- Insert the clock system for the gate array chip
- Insert testability access logic for the service processor unit
- Obey all electrical design rules for the gate array chip
- Make it easy to detect whether the tool has performed well
- Simplify the improvement of the tool

SID Database

The design of the SID database is fundamental to the robustness of the CAD system. Previous CAD databases have all assumed that the data is stable at the time that the CAD tools are working with it. Simulation, timing verification, design rule checkers (DRCs), and many other CAD tools assume that net lists and components are fixed and unchanging.

In synthesis, although the data is maintained in a form that makes it easy to update its parameter values, the basic structure of gates, pins, and nets remains the same. However, throughout most of the synthesis process, the basic structures are in a state of change. In fact, it is a characteristic of synthesis that logic functions are removed and replaced with new, functionally equivalent logic. Because of this difference, we designed basic data structures and

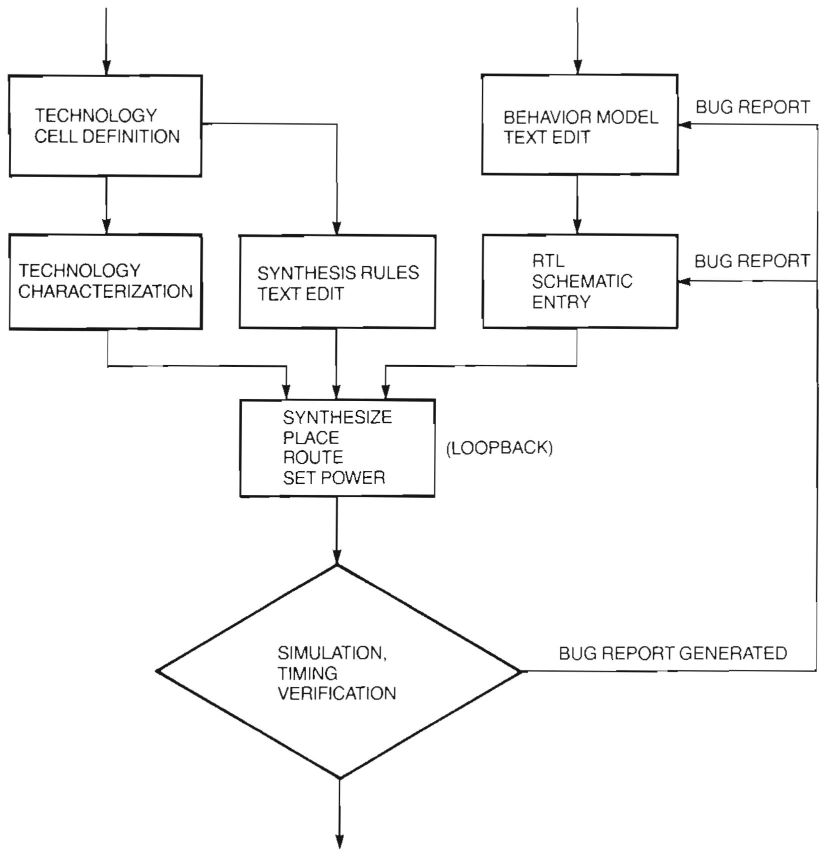

TECHNOLOGY CELL DEFINITION
BEHAVIOR MODEL TEXT EDIT
BUG REPORT
TECHNOLOGY CHARACTERIZATION
SYNTHESIS RULES TEXT EDIT
RTL SCHEMATIC ENTRY
BUG REPORT
SYNTHESIZE PLACE ROUTE SET POWER
(LOOPBACK)
SIMULATION, TIMING VERIFICATION
BUG REPORT GENERATED

Figure 2 VAX 9000 Development Methodology

manipulation functions that would allow efficient removal and replacement of logic.

We did use the primary objects of other CAD systems: gates, pins, and nets. However, we made a distinction between the definition of an object and its use or instance. Also, because we wanted these objects to be used at very high (i.e., behavioral and RTL) levels and at the gate level, we renamed them as models, ports, and signals. The primary database objects for SID are

- Modeldef. The modeldef is the definition of a logic function element. Analogous to a vendor data sheet, modeldef contains parameters that describe its function, timing, power, size, and other general information. All bounded blocks of logic function, from high levels of hierarchy (e.g., floating point unit) to low levels (e.g., simple Boolean gates), are kept as modeldefs. Typically, modeldefs are used multiple times and used more at the lower levels of the database hierarchy. For example, in the VAX 9000 system, there are two cache data multichip units, eight multiplier chips, and many thousands of two-input NOR gates.

- Modelinst. The modelinst is a use of a modeldef that contains only those parameters unique to itself. For example, two instances of a two-input OR cell may be in different places on a chip and, therefore, have different placement designators and timing characteristics. Each modelinst points to its modeldef definition to inherit the set of common definition parameters.

- Portdef. The portdef is the definition of an interface to or from a modeldef. Portdef contains parameters that describe its function, timing, data width, and other general information.

- Portinst. The portinst is an interface to and from a modelinst. Portinst contains parameters unique to itself, such as timing and power settings. Each portinst points to its corresponding portdef definition to inherit the set of common definition parameters.

- Signal. The signal is the means of connectivity among modelinsts and between hierarchical partitions. As shown in Figure 3, this connection is established through the interface portinst or portdef. For behavioral logic, the signal acts as a data flow arc; for RTL logic, the signal acts as a bus; and for gate-level logic, the signal acts as a net.

Synthesis rules must be able to walk the database in any direction (i.e., backward, forward, through hierarchy) looking for electrical rules violations or logic function redundancy, and testing for timing-critical path relationships. To perform these tasks, we added a series of multidirectional pointers to the SID database objects by using LISP capabilities. When an object is declared as a symbol in the LISP programming language, pointer management is included automatically. The LISP language is well known in the industry for its use in AI applications, but it has a reputation as being slow. Our special handling of direct database pointers enabled us to produce a LISP application that resulted in excellent run-time performance.

Once the data structures and their pointers were defined, we began to create a rich set of database access functions that had to be failproof. Therefore, we wrote functions to insert and remove the instance objects to ensure that the database pointer connectivity was properly maintained. These functions allowed us to effectively perform a many-for-many replacement of modelinsts with a single command.

Other secondary objects were defined to contain such types of information as synthesis knowledge (i.e., rules and groups of rules), general technology characteristics (i.e., the maximum number of cells on a chip), and general project-specific characteristics (i.e., the cycle time of the machine).

The synthesis knowledge in the form of rules occupies the majority of SID-compiled code and over 10 megabytes (MB) of run-time memory.

Rule Language

Based on research and the perceived complexity of the task at hand, we estimated that, to perform synthesis at an expert level, possibly thousands of rules would have to be written.

In researching current AI literature, we determined that existing rule languages were either too cryptic or too verbose to allow us to write and maintain a large rule set in a short time frame. Also, we preferred to write more powerful rules than those of previous rule-based systems. We wanted each rule to be used for making complex decisions and logic transformations based on timing, size, power, and logic connectivity. The rule does not "think." Instead, it mimics a logic designer looking at the characteristics of some pre-existing design, who then changes the design to improve it or make it more compatible with the new technology. The rule, for example, tests whether A and B are true, and if so, performs transformation C.

Based on these needs, we began developing the language Ruleform as the means for approxi-

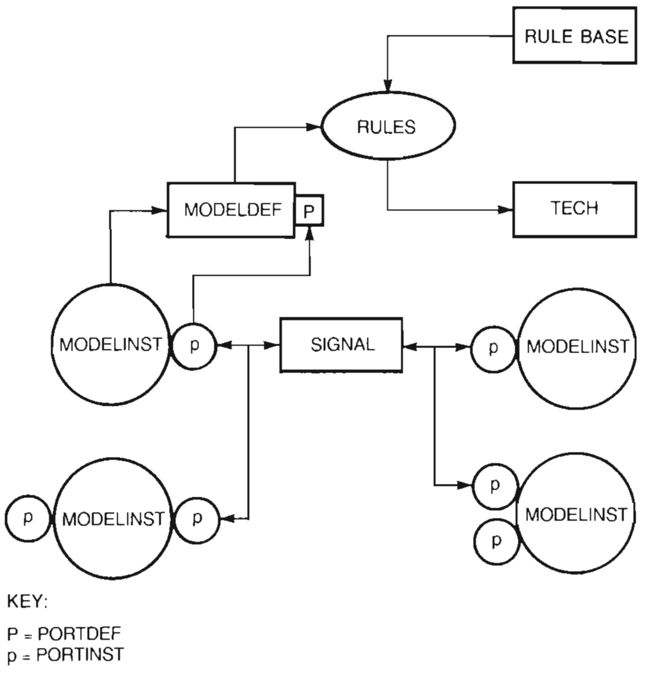


Figure 3 SID Database Objects

mating the designer decision and logic synthesis task. With this approach, the rule would mimic what a logic designer had done once and that action could be repeated again automatically in similar circumstances.

In Ruleform, rules have a left side for decisions and a right side for transformations, e.g., OPS-5, as do other rule-based languages. However, to make rules easier to read and write, Ruleform uses English language sentence structures to describe both tests and actions. The following predicate forms are used for left-side tests:

- Dbobject verb
- Dbobject verb dbobject
- Adjective dbobject verb
- Adjective dbobject verb dbobject

Verbs are words such as IS, ARE, =, >, IS_BOOLEAN, IS_A_NUMBER; adjectives are words such as ANY, ALL, NO. Dbobjects are database objects or the parameters of these objects.

The command forms used for right-side actions are command dbobject and command dbobject preposition dbobject. Commands are words such as INSERT, REMOVE, REPLACE, MODIFY; prepositions are words such as WITH, TO, FROM. The dbobject can be any of the primary database objects, secondary objects, or their parameters.

For more complex operations, we also allowed LISP functions to be called by prefixing them with the keyword LISP, or by insertion of a LISP expression. Thus, if the rule language cannot implement a required function, a LISP algorithmic routine is called. We used algorithmic transforms in the generation of adder carry-lookahead.

Ruleform Database Access

Because the database could be traversed in any direction for any arbitrary distance through the multidirectional pointer system, rules had to have the same traversal capability. Therefore, the dbobject of the Ruleform language is a shorthand notation of the "database walk." Dbobject can be used in a sentence to compare two database objects by walking to both of them and using a predicate for the comparison.

Had the database access been implemented in pure LISP programming notation, the sentence form would be lost in the many levels of expressions enclosed in parentheses. One test would occupy many lines of code and would read more like a software program than an English sentence. In this case, the chain of thought of the rule writer, the purpose of which is to capture the step-by-step thoughts of a logic designer in words, would probably be broken.

To improve the comprehension of the notation used for identifying the database object, we developed an <object><dash><object><dash><object>... notation for the walk to a database object. We also developed functions that would compile this notation into a LISP expression, complete with all the appropriate declarations for the most efficient runtime performance. Figure 4 shows the use of this notation.

Further, we incorporated into Ruleform a parameter definition mechanism that allowed us to define any arbitrary parameter, name what object it would attach to, and then use the name of the parameter in the Ruleform database access. This greatly expanded the role of synthesis in that it could now be used for passing controls and information to other CAD tools through parameters. Parameters relieved the designers of much tedious work, such as identifying clocks, logically equivalent signals to the placement program CUT, and the parity generator and checker signals to the diagnostic program, called HIDE.

Writing the Rule

Many of the tasks logic designers perform become automatic and intuitive over time. However, for a computer-based tool to develop a design, it must be able to measure cell counts, power and timing, and compare alternative implementations against budgets of cell counts, power, and timing. To find the critical path, a computer-based synthesis tool must perform timing analysis in the same way that the traditional timing verification tool does. In a sense, the synthesis tool must preverify the decision before casting the synthesis transformation in concrete. Therefore, for a computer to do logic design, we had to analyze the steps that had become automatic and intuitive, break those steps down, and formalize them in minute detail.

Rule Example

Consider an example of a simple cell-mapping rule. The purpose of this rule is twofold: pick the most appropriate cell for a configuration of Boolean gates and attach the most critical path signals to those input portinsts that have the fastest propagation delays through them.

A designer might determine the critical path from experience or through trial and error. The designer also might actually count loads on signals and add estimated signal delay to gate delay of all paths that might involve the timing-critical piece of logic.

dbobject	means
MODEL	get the name of the modeldef of the current instance
INPUTS	get the inputs of the current modelinst
SIGNAL-2ND-INS	get the signal of the second input of the current modelinst
INSTANCES-DRIVERS-SIGNAL-2ND-INS	get the instances whose outputs are the driving pertinacities of the signal of the second input of the current modelinst
SOURCES	get the instances whose outputs are the driving pertinacities of the signals of the inputs of the current modelinst
DUSTS	get the instances whose inputs are the load pertinacities of the signals of the outputs of the current modelinst
MODEL-SOURCES	get the name of the modeldefs of the source modelinsts of the current modelinst

Figure 4 Example of a LISP Expression

These alternatives are all very time-consuming. We decided a computer is best suited to do this type of work. In SID, a timing analysis routine is run repeatedly, as the database changes, to set timing parameters on every portinst of the design. The product of these calculations is a timing debt number set on every portinst. If the number is positive, the path is over budget (i.e., is in timing trouble) by the number in picoseconds given. If the number is negative, the path is under budget (i.e., has slack). The timing debt number allows the rules to access the timing debt parameters to find critical paths.

In the example shown in Figure 5, four Boolean gates exist as a tree in the middle of a gate array cell. The dest-side gate is a three-input OR, and the source-side gates are two-input ANDs. The entire cycle time of the machine depends upon the most timing-critical path, which runs through the first input of the second AND gate.

Because this rule replaces four gates, it has higher priority over other rules that replace fewer gates. When the rule arbiter is called with the OR as the current instance, the arbiter executes the left side of the rule (i.e., the first part up to the arrow). The left side of the rule checks that the current instance is a three-input OR and all source instances are two-input ORs. It then chooses the most critical path from among the inputs of the sources and notes the other inputs of the sources that were not critical. Because all of the tests in the left side of the rule returned true, the rule is said to have "fired." The right side of the rule may now be applied.

The right side removes the current instance, i.e., OR, and inserts the cell with the most critical path connected to the input that has a fast propagation delay to the output. By removing the OR, the destination of the ANDs is removed. REMOVE_IF_NO_DESTS then removes these ANDs.

The actual rule that does this transformation is

```
(defrule "mapCELLn")
   (model has_profile '(or3))
   (all models-of-1st-sources-of-inputs
        have_profile '(and2))
   (found{critical}-inputs
        is_most-critical 1)
   (any {tagged}-inputs
        are_not_in {critical})
   -->
   (replace *instance* with
        out = (CELLn
                (not ins-sources-{tagged})
                (not ins-sources-{critical}))
   )
   (remove_if_dests sources)
)
```

The rule also checks whether signal outputs of the ANDs have more than one load. The rule allows the transformation if the critical path has more than one load, but disallows the transformation if the noncritical paths have more than one load at the output of the ANDs. Thus, duplication of the source AND logic is prevented except when absolutely necessary, e.g., to remove a wire delay to increase the speed of a critical path.

Organization of Rules into Rule Bases

The quantity of rules required that the rules be organized into groups, called rule bases. As we defined the minute steps of the logic design tasks, it was apparent that groups of rules were separated by levels of abstraction, as depicted in Figure 6. For example, a sequence of logic design can be characterized as a progression through the levels:

behavior → RTL → Boolean → technology map → wiring, tweak → parameter set → placement → route → power adjust

We organized the rules by type of activity into rule bases. We also developed a run-frame sequence process that would apply these rule bases, in order, from behavior through detailed adjustments at the technology level. The rule bases are

- Behavior rule base, which contains rules to expand behavior and RTL instances. These rules transform high-level instances of adders, incrementers, comparators, decoders, encoders, and DECSIM behavior expressions into generic Boolean instances. They also perform simple bit replication for data path instances, such as 32-bit MUXes.
- Optimize rule base, which contains rules to transform Boolean logic for minimization or timing improvements. These rules performed the well-known D'morgan, distributive, and associative transformations to mold networks of generic Boolean instances into a configuration that is best suited to map into the cells of the target technology.
- Map rule base, which contains rules to transform generic Boolean and bit data path instances to technology cells. This rule base actually was divided into two rule bases, one that mapped I/O cells and one that mapped internal gate array cells.
- Wiring rule base, which contains rules to improve timing by loading and logic adjustments

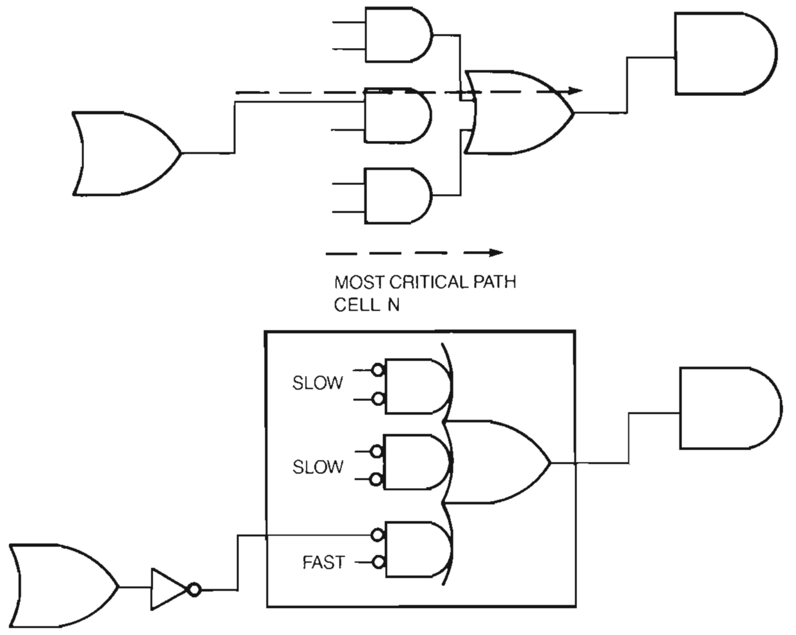


Figure 5 Critical Path Map

and rules to detect and correct electrical rules (ERC) violations.

- Tuning rule base, which contains rules to adjust power on intersections of multiple timing-critical paths.
- Parameter rule base, which contains rules to set parameters for the placement and route programs. These rules include setting parameters to identify logically equivalent signals, setting pin groups to force collections of pins to be near each other in placement, and weighting parameters to force timing-critical signals to be shortened.
- Placement and route, which are not rule bases but CAD tools separate from the synthesis tool. Placement and route of Motorola Macrocell Array III (MCA3) gate arrays occur here in the overall design sequence.
- Power rule base, which contains rules to adjust power on gate array cells and cell output followers. After initial placement and routing, a more accurate assessment of signal delay can be made. The power rules track the power distribution of the gate array cells and the contribution of each cell's current settings to the power budgets for ten regions of the chip. These rules adjust current upward to improve the speed of critical paths. The timing calculations for this rule base use actual routed wire delays.

The Larger Physical Design Process

SID is part of a larger chip physical design process that includes synthesis, placement, and routing. The entire process is linked together in a two-pass process, called loopback.

The first pass of loopback accomplishes three goals: the initial synthesis of the chip based on estimated interconnect delays, placement of those synthesis results, and routing, which includes accurate interconnect delay calculations.

The second pass of loopback accomplishes the final synthesis with much more accurate interconnect delays and a high probability that the subsequent physical instantiation will achieve all design goals for timing, space, and power. The final synthesis made changes only where required. Final placement began with the results of the first pass of loopback, except where changes were made, and routing rerouted only those nets that had been modified. Our objective was to limit the number of passes through loopback to two and avoid endless cycles through the CAD tools.

The placement process itself consists of three major phases. The global phase, called gravity collapse, attempts to achieve relative orientation of the various gates and disregard density. The distri-

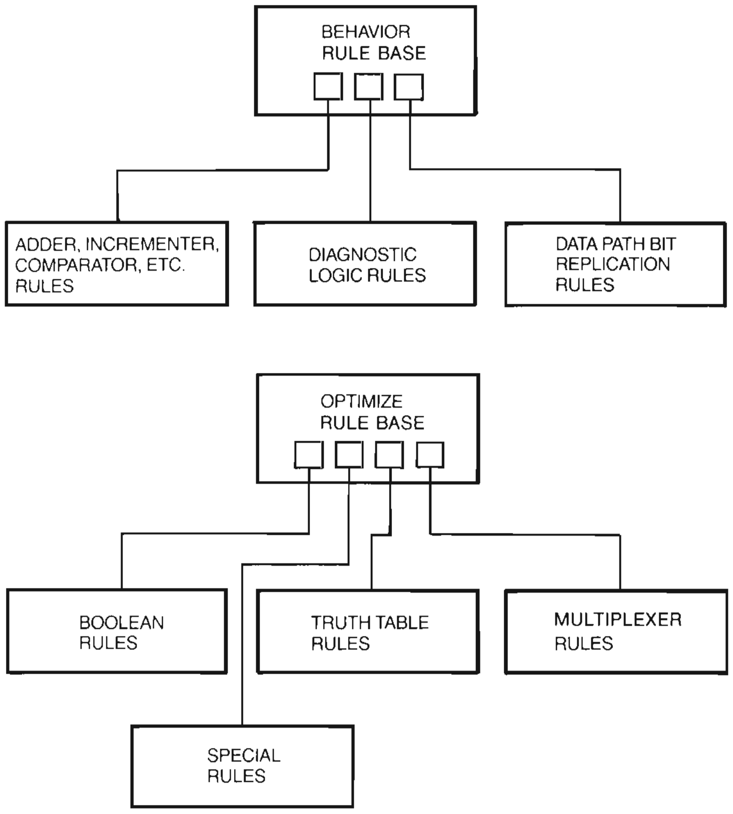


Figure 6 Knowledge Representation Hierarchy

bution phase, called regridding, attempts to assign the gates to available positions and maintain the desired orientation. The multipass final placement phase swaps cell locations, gates, nets, and pins to reduce weighted net lengths but still adhere to the technology-supplied design rules.

During the local placement phase, synthesis-supplied net weightings and equivalent net parameters are utilized. The net weightings are part of the complex algorithm used to determine whether a potential swap of a cell location, gate, pin, or equivalent net is beneficial. The equivalent net parameters allow the placement program to detach a net from one pin and reattach it to another pin to supply an equivalent signal. This process was a particularly common occurrence because synthesis had to supply the same or complementary signals to many destinations and still adhere to a technology-driven limitation of no more than four loads from any one source.

Because the placement process introduces so many changes in pins, gates, and nets, we felt it was prudent for the placement program to simply regenerate the CADEX database format when it was finished. This approach avoided the problem of developing a "back-annotation" process, which would be required if modification of the existing database were attempted and would be a complex process, given the number of changes made. Also, because most of the schematic source design is in RTL format, many, if not most, of the placement-introduced changes are not visible in the schematic.

Placement results were entered into our internally developed routing CAD tool, called Chameleon because of its ability to adjust to its environment. Chameleon is a highly rules and parameter-driven tool. It was used to route all 77 gate arrays, all 22 multichip units, and both the CPU and system control unit planar boards of the VAX 9000 system. For the 77 gate arrays in the VAX 9000 system, the router achieved better than 99 percent completion. Further, the average net was routed to between 101 and 102 percent of its Manhattan net length. (Note: Determination of Manhattan length is somewhat ill-defined when copper-sharing is allowed, and some net segments are common to multiple source-destination paths.) The routing was so efficient, with regard to length, that the synthesis and placement programs could assume that determination of a net's end points would effectively determine its eventual routed length and, by extension, its interconnect delay.

Upon completion of the actual routing, interconnect delay calculations were made for every net source-load combination by a router-related program. We did these calculations at this point in the process for two reasons. First, all the necessary data was readily available in the router's database. Second, accurate delay calculations were needed by synthesis during the second pass of loopback to verify the assumptions made during the first pass and make any adjustments necessary to achieve timing and power constraints. For those few connections that were not routed completely, calculations were made based on the Manhattan length and a small contingency factor.

Problems

Developing and using the new synthesis design methodology was not without problems. We were able to fix some of these problems for the first VAX 9000 system generation. However, because of time and resource problems, others were deferred to the next project.

Digital's previous CAD system ran on a combination of 36-bit DECSYSTEM-20 computers and 32-bit VAX computers. In switching completely to the VAX system for all CAD processes, we had to rewrite much of the DECSYSTEM-20 computer's existing code and replace the Stanford University design system (SUDS) schematic drawing program with the CAE2000 system. In replacing SUDS, we lost nearly one million lines of code, which was used for such tasks as wire listing, drawing, back annotation, and electrical rules checks. Some of these were available in the CAE2000 system, but others had to be developed external to that system.

We designed a common database format, called CADEX, that allowed the CAD tools to communicate with one another. For example, a design could be extracted from CAE2000 drawings to CADEX, which would supply the design to SID. In turn, SID would write CADEX output, which would be read by the placement program cut. This program would then write CADEX output, and the cycle would continue. New libraries had to be created for RTL schematic bodies and MCA3 cells. Data formats and parameters had to be defined for passing information between the CAD tools.

In SUDS, results could be written back to the drawing program (i.e., back annotation). We had hoped to be able to use the same process in the new process, but were prohibited from doing so by deficiencies in the CAE2000 system. As a result, the overall CAD process had to be repeated from the beginning many times.

As with any new development tool, we experienced setbacks. For example, we developed the generation of an adder that algorithmically picked each gate of the carry-lookahead for optimum configuration of the Boolean trees with respect to fan-out and path delays. We then wrote Boolean optimization rules that attempted to merge ORs together as if no fan-out existed between them. However, when we did this, the carry-generate, least-critical OR paths merged, which forced the more critical AND-OR combinations into simple one-stage cells, with extra levels of wire delay between them. Within a few weeks, we were able to correct the faulty rules to also consider the timing-critical paths, which allowed the adder to improve along the carry-propagate paths. Eventually, by working on the cell selection, load allocations, and power setting, we were able to produce a 64-bit adder at 3.2 nanoseconds (ns) in the MCA3 technology. The best hand design was 3.3 ns for a 59-bit adder. At the time, the logic designers estimated that an extra stage delay, or 3.7 ns for the 64-bit manual design, would be required.

Since synthesis was new, there was a great deal of skepticism as to whether it could perform as well as a manual design. Some logic designers never gave it a chance. Other designers encountered early problems with it or experienced schedule pressures and, as a result, resumed using hand-design methods. However, most designers stayed with the process until it produced acceptable results. In the process, they supplied feedback and algorithms that were converted into additional SID rules. This work was crucial to the evolution of the program. Only by adding new rules provided by logic designers could SID be improved for future designs. The successors to the VAX 9000 system will reap the major benefits from this work, through improved designer productivity and time savings.

The effect of timing constraints and the general accuracy of timing calculations on synthesis results cannot be underestimated. We required that timing budgets be specified to every chip to indicate the timing criticality of each I/O pin. The budgets were specified from the I/O pin to latches of either of the two clock phases (TA or TB). Default budgets were applied on paths between latches that were contained on the same chip. If a budget was missing, SID considered that the speed of the path did not matter. However, as we ran the paths without budgets, we frequently found that SID had designed the path to be too slow. We then had to specify a budget for that path and repeat the testing process. A better alternative would have been to have a tool set the initial budgets for all I/O pins and for the user to modify budgets as necessary.

The accuracy of timing calculations is another factor in the design results. Because SID used worst-case gate delays rather than rise and fall delays, its calculations of timing debt generally produced incorrect numbers that indicated timing problems that did not actually exist. Triggered by this inaccurate timing data, the rules generated duplicated logic to reduce signal load fan-outs and needlessly increased chip power.

Ultimately, synthesis methodology enabled the CAD system to produce accurate gate array designs. With the Ruleform language, we improved the rules to meet the changing needs. In the majority of cases, rules that were not in time for one designer benefited many other designers at a later design stage. Using an ECO (engineering change order) process, we adjusted the placements and power and improved the timing-critical paths by requesting specific power settings or fan-outs that the regular execution of SID does not normally generate.

Results

Approximately 93 percent of the gate-level database was synthesized from source RTL design, DECSIM, and microcode truth tables. The other seven percent was implemented in the source schematics in the form of technology cells, known as CLEGOs. Since the RTL often is quite similar to the finished gates, this percentage is not an accurate reflection of the amount of work involved. The RTL bodies were quite simple and without technological aspects, such as strange polarity inversions and clock connections. However, they did require that the designer specify all data paths and control logic in the true and false sense; e.g., A and B but not C feeds the select to 32-bit data path MUX.

The ratio of database size for RTL bodies compared to synthesized gates is a better measure of how the design entry was simplified through the use of RTL schematics and SID synthesis. The comparison was done for CADEX file sizes of the RTL designs versus the synthesized gate designs just prior to placement. The ratio of RTL logic complexity versus gate logic complexity, for each CPU box, is shown in Table 1.

Table 1 Ratio of RTL Logic Complexity to Gate Logic Complexity

	RTL Logic Complexity	Gate Logic Complexity
E-box	1	4.73
I-box	1	4.92
M-box	1	4.40
V-box	1	3.17
Average	1	4.30

The average ratio of 1 to 4.3 is interpreted to indicate that 23 percent of the logic design work (not counting placement, routing, simulation, and timing verification) was done by logic designers and 77 percent by synthesis.

Another perspective is gained when we consider the amount of synthesis rules that were applied. The number of rules varies tremendously in relation to the impact. For example, the adder-generation rule, which takes about 1 CPU minute to complete for a 32-bit adder, performs the equivalent of approximately 4 person-days of work. On the other extreme, a parameter-setting rule that is tested and applied in .1 CPU seconds performs the equivalent of 15 to 30 person-seconds of work.

Table 2 shows the approximate number of SID rules applied during synthesis runs. The rules performed different categories of activities for the 77 VAX 9000 system MCA3 gate array chips.

Thirteen bugs caused by synthesis were found in the gate-level simulator. These bugs were either typographical errors in the rules or incorrect interaction within a set of rules. Although each rule was tested independently for correctness through

Table 2 Activity Categories for the VAX 9000 Gate Array Chip

Rules	Number
Expand behavior instances to bit level	28,567
Minimize and optimize Boolean logic	11,550
Initially select macros and macropins	58,597
Detect and correct electrical and design rule violations	7,392
Improve timing by loading and logic adjustments	85,008
Set high-power on common-critical paths	3,850
Set parameters needed by other CAD tools	165,000
Set high-power on timing-critical paths	24,024
Total number of rules applied	383,988

full-pattern simulation, it was not possible to completely test the interaction of several rules. The simulator found approximately 500 designer-introduced bugs, and the breadboard found another 41 bugs. The breadboard was an early version of the VAX 9000 system that was built with printed wiring board technology for the purpose of faster simulation and debugging of the design.

These numbers translate to about 1 bug per 200 gates designed by hand, compared to 1 bug per 20,000 gates designed by synthesis, when viewed from the above ratio of 23 percent hand design versus 77 percent synthesized design, using the 400,000 gate VAX 9000 source design. The fully realized VAX 9000 system is 700,000 gates when multiple uses of chips are considered.

In addition to the more traditional synthesis functions of logic minimization and technology cell mapping, we used the tool to insert clock logic, scan logic, AC test circuits, parameters to control placement and routing, information to the test pattern generation tool, diagnostic isolation tool, and simulation tools. All these functions made the logic designers' job much easier through automation of some tedious and error-prone work.

We also found a unique application for synthesis rules in the improvement of wire delays on the chip by rearranging and rebuffering signal nets, based on timing debts. A set of nearly 15 rules resulted in a 10 percent path delay improvement across the board for all gate array chips, including the 7 percent of logic done in CLEGO technology cells.

SID-synthesized gate arrays were found to have no electrical rules violations caused by the tool. A few electrical rules errors were, however, introduced by manual ECOs. An example of electrical rules error is the connection of two incompatible technology gates, such as an internal chip cell to a chip output pad.

As shown in Table 3, the run-times for synthesis, placement, and route for MCA chips varied greatly, depending on design complexity.

Table 3 Average Run-times for MCA Chips

MCA Chip	Average Run-time
Synthesis	30 minutes to 3 hours
Placement	4 to 10 hours
Route	4 to 12 hours
Delay calc	1 to 2 hours

Acknowledgments

The authors would like to acknowledge the developers of the SID, CUT, and Chameleon programs for their contributions and dedication to the development of the VAX 9000 product: Ron Bosslet, Mike Boucher, Ed Fortmiller, Herb Kolk, Snehamay Kundu, Nevine Nassif, Jayanth Rajun, Steve Root, Dave Tongel, Dave Wall, Yu Wang, and Greg Zima.

Reference

1. G. Brown, *Laws of Form* (New York: E.P. Dutton Publishers, 1979).

General References

R. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis* (Boston: Kluwer Academic Publishers, 1984).

J. Darringer et al., "LSS: A System for Production Logic Synthesis," *IBM Journal of Research and Development,* vol. 28, no. 5 (September 1984): 537–545.

T. Kowalski and D. Thomas, "The VLSI Design Automation Assistant: What's in a Knowledge Base," *22nd Design Automation Conference* (1985): 252–258.

K. Bartlett et al., "Synthesis and Optimization of Multilevel Logic under Timing Constraints," *IEEE Transactions on Computer-aided Design,* vol. CAD-5, no. 4 (October 1986): 582–595.

A. Goldberg and D. Robson, *Smalltalk 80, The Language and Its Implementation* (Reading, MA: Addison-Wesley, 1983).

M. Burstein and M. Youssef, "Timing Influenced Layout Design," *22nd Design Automation Conference* (1985): 124–136.

R. Brayton et al., "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-aided Design,* vol. CAD-6, no. 6 (November 1987): 1062–1081.

Karen E. Barnard
Robert P. Harokopus

Hierarchical Fault Detection and Isolation Strategy for the VAX 9000 System

The VAX 9000 system was designed to compete in the mainframe market. Mainframe customers not only require high processor performance and throughput, but also a system which is reliable and always available. This paper demonstrates how the newly implemented scan system, in conjunction with scan pattern testing and symptom-directed diagnosis (SDD), is essential to satisfy these needs. SDD is the use of on-line error detectors and state information saved at the time of an error to isolate the fault that caused the error. The scan system of the VAX 9000 system allows individual state elements in the processor to be set and sensed, and is the basis for fault detection and isolation.

As computer technology becomes more advanced, designs are becoming more dense. Density implies volume. For example, the typical chip on Digital's most advanced CPU, the VAX 9000 system, contains 8000 gates. The VAX 9000 logic packaging also compounds the diagnostic problems by making the gates physically impossible to reach with a logic analyzer.

As the gate volume increases and the logic becomes less accessible, the problems increase for manufacture of the design, debug of the prototypes, and repair of the machine in the field. The debugging stages and the hardware repair process require that faults be found quickly and accurately to ensure that downtime is minimal and valuable resources and inventory are not wasted. This paper presents the solution used in the VAX 9000 system for detecting and isolating hard and intermittent faults. The diagnostic solution for the VAX 9000 system comprises the scan system, tools to generate test data, utilities to submit the test data to the scannable logic, and an expert system to record symptoms and produce a callout over time.

Traditional Fault Detection and Isolation Methods

Excluding the MicroVAX chip, all VAXCPUs designed prior to the VAX 9000 CPU are supported by macrodiagnostics and microdiagnostics.

Macrodiagnostics execute from the system's main memory and verify that the CPU can successfully behave as the VAX architecture mandates. Microdiagnostics execute from control store random-access memory (RAM) and have access to internal state elements of the CPU. Although microdiagnostics were intended to provide better fault isolation than macrodiagnostics, isolation is still not optimal because the access points comprise only a fraction of the total CPU. Since both types of diagnostics provide imprecise fault isolation, an engineer must be highly skilled in the analysis of both the code and the internal workings of the CPU to repair a broken machine.

To avoid the time-consuming process of manual fault isolation, the field engineer often extensively replaces modules, which is a costly repair method. Historically, this practice has been a problem not only for Digital but for the entire computer industry. Another disadvantage is that these diagnostics are executed using the suspect logic, which can produce incorrect test results. Finally, both diagnostics often fail to provide the desired fault resolution because of fault propagation. The fault spreads across module boundaries because a typical test executes several instructions and each instruction requires one or more CPU cycles before the results are analyzed.

Solution for a High-volume Design

The solution used in the VAX 9000 system to improve fault detection and isolation preserves the role of the functional diagnostics and addresses the inherent problems of those diagnostics. An integral scan system addresses the density and packaging issues. The CAD processes that automatically generate test data address the complexity and schedule issues. The scan system also reduces the problem of fault propagation by providing a mechanism to stimulate the machine and examine the results after just one machine cycle. Further, more direct control over most of the internal logic elements improves test coverage and fault isolation.

Increased Visibility

In the VAX 9000 CPU design, multichip unit technology created a machine that could not be built or diagnosed without a marked increase in visibility points. In previous VAX systems, the number of visibility points for microdiagnostics varies from one machine to another. For example, the VAX 8700 system has just over 150 visibility points, and the VAX 8600 and 8650 systems have over 3000 visibility points. Most points are read-only points, and the diagnostic processor has limited direct control over initializing individual CPU state elements. In contrast, the VAX 9000 scan system provides access to over 20,000 internal machine state elements for both reading and writing and direct access to all internal RAM and register structures. The design of the VAX 9000 system significantly improves CPU and system control unit logic visibility.

The scan system is used for diagnostic purposes and to initialize the state of the CPU and system control unit. Because the number of windows into the system are increased, the design can be partitioned into smaller regions, which improves fault isolation. The fault detection and isolation strategy depends on components that vary in complexity from a simple scan latch to automatic generation of the test data.

An important feature of the scan system is that the scan latches can be influenced by the scan system and by the system logic. Therefore, the scan system functions can be tested and verified independently of the system logic. However, if the scan system is not functioning properly, the scan pattern diagnostic cannot produce valid results. Therefore, scan system faults must be fixed before running the scan patterns.

Testing Hierarchy

Testing for the VAX 9000 system begins with diagnostics that are run automatically when the system is powered up. These tests ensure that the service processor unit and the scan system's basic components do not contain any faults.

After the tests are executed, the service processor unit's operating system is booted and the scan hard-core diagnostic is run. The hard-core diagnostic assumes that the system has passed the power-up diagnostics and, therefore, the scan controller will operate properly. The hard-core diagnostic tests the components of the scan system that reside on the CPU and system control unit planar modules. If the scan system is broken, the machine cannot be initialized. When the system passes the hard-core diagnostic, the scan pattern diagnostic tests the integrity of the scannable system logic.

The functional diagnostics and system exercisers are run after the scan pattern diagnostic has verified that no structural problems exist. In the testing hierarchy of the VAX 9000 system, functional diagnostics are as important as structural testing. Functional diagnostics verify that the design represents a valid VAX system.

Timely Testing

In the course of developing a system design, several revisions of the system components are usually required. Each revision represents a different machine for testing purposes. To test each revision of the VAX 9000 design in a timely manner, a tool, called Scan Environment Patterns for Test and Repair (SCEPTER), was developed to generate test data. SCEPTER, an automatic pattern and test generator, takes as input the data used to manufacture the multichip units and structural models that simulate the design. Both inputs are produced during the logic design process.

SCEPTER Process

Pattern generation for the VAX 9000 system is a recursive process that initializes the scannable latches in a logic model, simulates one or more system clocks, and reads the contents of the scan latches. The contents are read as variable-length vectors, one bit for each scan latch in the design. The vectors to initialize the logic, the timing definitions and the expected result, and mask vectors are written by SCEPTER into an ASCII file. The scope of the testing for a particular file is determined by the scope of the model that was input to SCEPTER. SCEPTER is quite flexible and can generate data files that target a Motorola Macrocell Array III (MCA3), a multichip unit, or CPU.

The SCEPTER data files are translated into binary formats to reduce the size of the files and to allow one pattern generator to satisfy the requirements and abilities of every test environment. The environments that use data generated by SCEPTER are

- Trillium's Micromaster Plus, the MCA tester used by the MCA3 vendor and Digital for the manufacturing test process. The tester ensures that the chip internal logic is fault-free prior to mounting it on the multichip unit.
- MCU tester, which is a comprehensive multichip unit tester Digital developed for the manufacturing test process. This multistation high-volume tester has access to the multichip unit's I/O pins and can probe the multichip unit and MCA pins.
- Manual probe station, which started out as an engineering tool but has evolved into auxiliary diagnosis too in the multichip unit test process. It is used to diagnose multichip units that have been removed from a VAX 9000 system because of suspected faults.
- VAX 9000 kernel environment is used by Engineering, Manufacturing, and Customer Services to verify MCU installation.

Pattern Generation Process

The automatic test and pattern generation (ATPG) process begins by determining what portion of the design is to be tested. A computer-aided design (CAD) tool partitions the models into chunks before SCEPTER is run. A chunk can consist of an MCA, a multichip unit, one CPU, or any portions of these units.

Typically, one multichip unit is considered to be the targeted device for testing, and any multichip units that communicate with the targeted multichip unit also are included in the test process.

Once the model is established, the recursive process of initializing, simulating the system clock, and reading the results continues until the generation algorithms are satisfied that no additional faults can be detected. The simulation time can be lengthy in this process, and, therefore, reduced coverage limits are set for multichip unit pattern generation. MCA pattern generation usually produces better than 98.5 percent coverage.

Basic Theory of Structural Testing

As discussed earlier, the scan system narrows the scope of testing to an area as small as one system clock cycle. If the scan latches are strategically placed, a fault's source can be pinpointed to a relatively small region. The size of the region is directly related to how far apart the scan latches are placed. The spacing of the scan latches also affects the isolation callout. The number of components involved in a callout can be decreased if scan latches are situated such that, on input to a chip, a signal feeds a scan latch prior to the signal being used in combinational logic. Figure 1 illustrates the callout that occurs if a fault is detected on either of the two signals shown.

The callout for signal A, where the chip is insulated with scan latches, is 30 percent smaller than the callout for signal B, which does not have boundary scan latches. The difference is even larger for signals that converge on a common area of combinational logic.

Using Boundary Scan to Improve Fault Isolation

The example in Figure 1 does not illustrate what happens to the isolation callout in the case of a multibit signal that communicates with several chips. Figures 2 and 3 demonstrate the effect that fanIN has on fault isolation. Figure 2 has boundary scan, and Figure 3 does not. The following discussion centers on these two figures.

The callout in Figure 3 includes an extra component, i.e., combinational logic, for each chip. If a scan latch is placed between the combinational logic and the chip boundary, the callout list can be reduced. The reduction between the callouts in Figures 2 and 3 is 55 percent.

The hierarchical test strategy, which was detailed earlier, confirms the following items as good on the MCU tester prior to installing the multichip unit on the planar module:

- Chips 1, 2, 3, and 5
- MCUx HDSC
- MCUy HDSC
- MCUx flex connectors
- MCUy flex connectors

Although these items are confirmed as good, a fault can still exist on the planar module or in the flex connectors. Because the flex connectors are moving connectors and subject to abrasion, they have the highest probability of breaking and are, therefore, the weakest link in the multichip unit assembly.

As such, these connectors require the greatest protection and deserve a high-level of suspicion

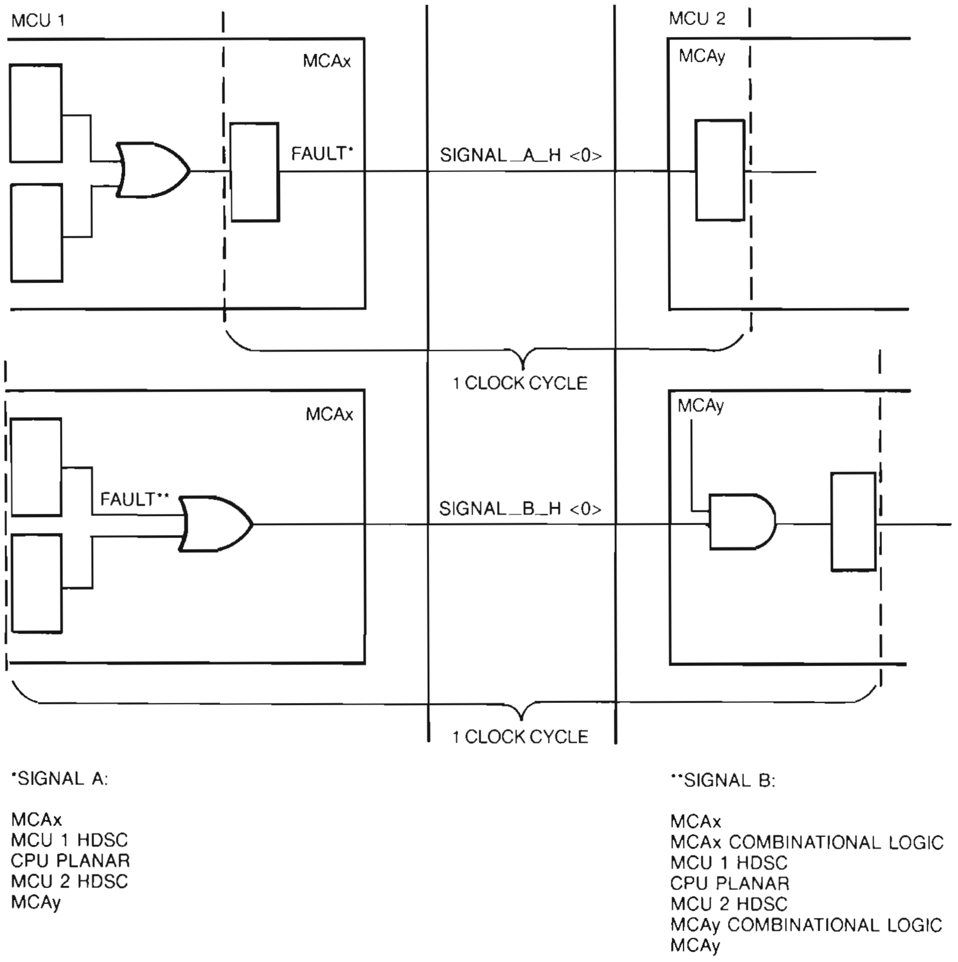


Figure 1 Example of Two Faults with and without Scan Latches on the Module Boundaries

when isolating faults. As a result of testing in the manufacturing process, the chip's internal logic and the HDSC systems can now be temporarily removed from the callout. If a fault occurs, boundary scan latches can isolate the fault to one signal. Without boundary scan, all three signals have to be included in the callout because the fault source cannot be accurately pinpointed.

In this example, provided that no other faults are present, each multichip unit's flex connector has an equal probability of having caused the fault. For every inch of flex connector, there are 30 signal connections. A minor piece of debris on a connector can potentially cause a number of failures. Therefore, the least costly repair strategy is to reseat the multichip unit after cleaning the flex connector and the pads on the planar module.

If the fault is still present after reseating the multichip unit, the multichip units are swapped and the patterns are rerun. If the run is failure-free, then the cause of the fault is on the removed multichip unit. The defective unit is sent to a repair depot for diagnosis and repair.

For discussion purposes, if additional faults are detected on one of the multichip units during the testing, then that multichip unit should be the first one to be reseated or swapped. Thus, the highest number of faults can be eliminated for one MCU replacement. The remainder of the repair verification procedure would be the same as in the case where no other faults existed.

Scan Pattern Diagnostic

The scan pattern diagnostic is a utility that resides on the service processor unit's system disk and processes the structural test data generated by SCEPTER. The scan pattern diagnostic runs each file based on the user's input.

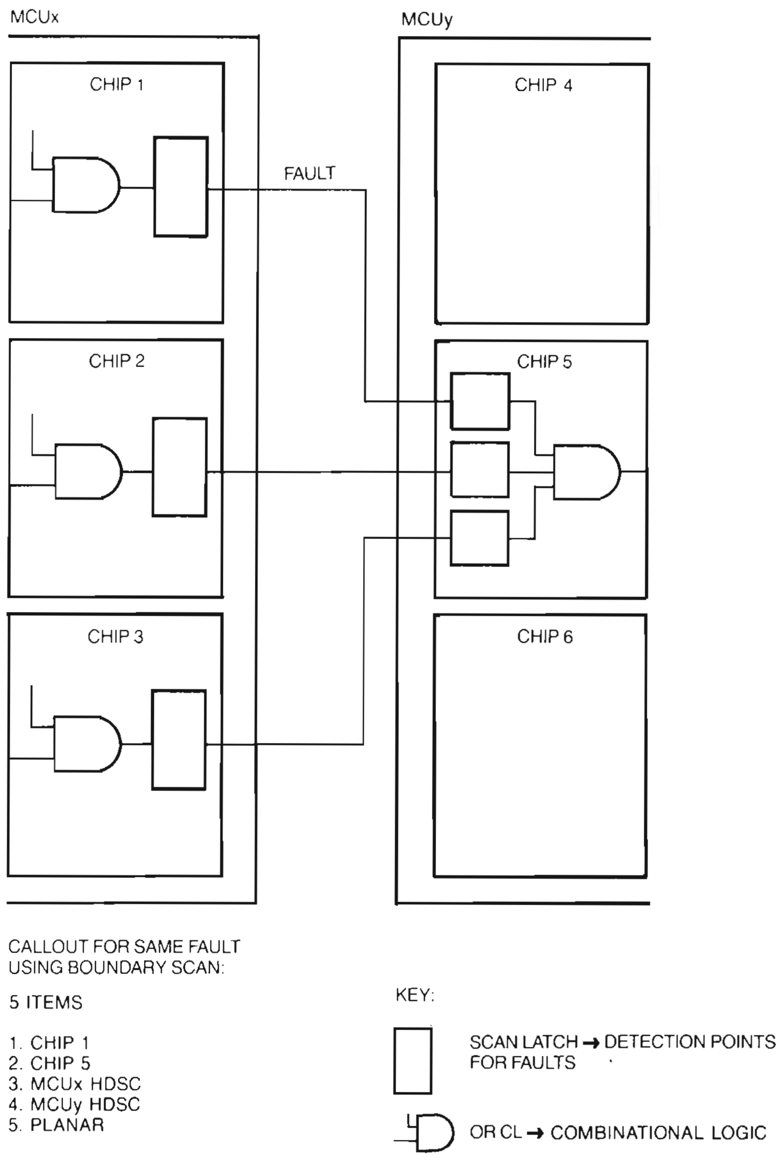


Figure 2 FanIN with Boundary Scan

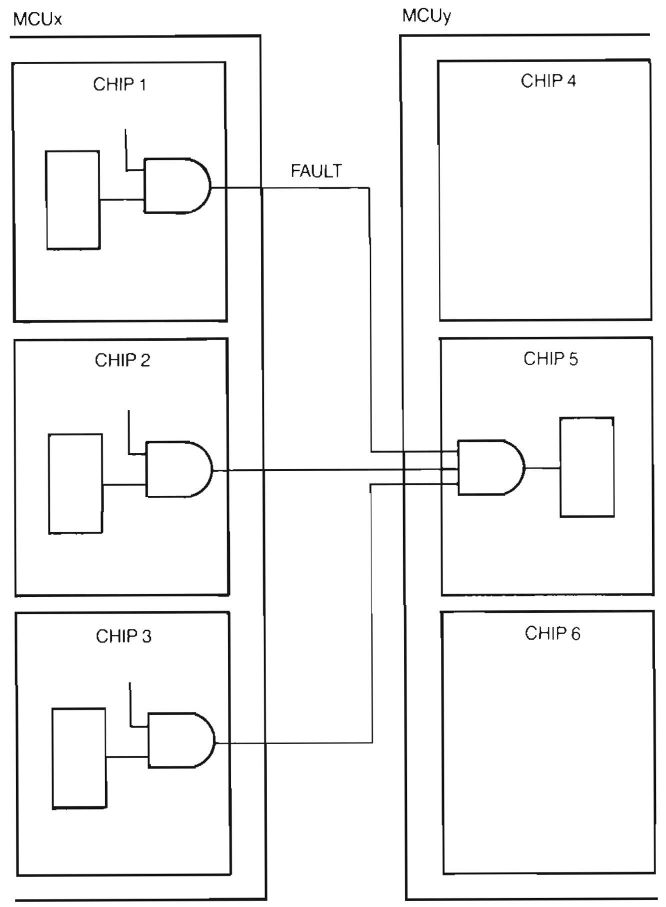
The interface supports flexibility for testing the scannable logic in the system control unit and one to four CPUs. The scan pattern data files contain the data required to test the hardware as discussed earlier in the Pattern Generation Process section.

The diagnostic packages the test data into scan operations that are submitted to the scan control module through the service processor unit system calls. The scan pattern diagnostic checks the returned status; and if faults are detected, saves the physical location of each fault in an internal database.

SCEPTER provides isolation maps, which are lists of the components that may be responsible for the fault detected by a given scan latch. There is one isolation map for each scan latch involved in the testing. When a scan latch detects a failure, the scan pattern diagnostic uses its physical location to access the isolation map provided by SCEPTER. The contents of the maps are used in the isolation callout.

Proper Niche for Structural Testing

We did not design the structural test process for the



CALLOUT WITHOUT BOUNDARY SCAN:

11 ITEMS

1. CHIP 1
2. CHIP 2
3. CHIP 3
4. CHIP 5
5. CHIP 1 CL
6. CHIP 2 CL
7. CHIP 3 CL
8. CHIP 5 CL
9. MCUx HDSC
10. PLANAR
11. MCUy HDSC

KEY:

▯ SCAN LATCH → DETECTION POINT FOR FAULTS

D OR CL → COMBINATIONAL LOGIC

Figure 3 FanIN without Boundary Scan

VAX 9000 system to cover every test problem. Instead, we designed the process to ensure that the hardware can physically operate as described by the design data. Structural testing cannot be used to determine if the VAX 9000 system is operating as a VAX system, that is the job of the functional diagnostics. Further, structural testing cannot be used to determine if the system is robust enough to support multiuser traffic; the User Environment Test Package (UETP) exercises the system hardware and operating system. The structural test process also cannot be used to find problems with the design. Architectural verification tests perform that function. Finally, structural testing cannot be used to detect intermittent faults. Because this type of fault requires the presence of special conditions which the test data may not provide, on-line error detectors and symptom-directed diagnosis are more effective alternatives.

Structural testing must be performed at a low level. It should be done when power failures or power surges occur, when multichip units on the

CPU or system control unit planar are swapped, or when signal-carrying cables are installed. The scan patterns also should be run if the system crashes or applications begin to behave erratically for no apparent reason. Initially, when a system problem occurs, the cause must be isolated to either the software or hardware to initiate the correct remedial action. If the hardware appears to be the cause, then the hardware diagnostic strategy must be followed to obtain optimal fault isolation in a minimal amount of time.

Structural Test Process

The VAX 9000 system's structural test process shows that, given the proper pattern data files, faults can be detected and isolated faster than with traditional methods that use the symptoms from the functional diagnostics. Structural testing not only fills a gap in Digital's test hierarchy, but also preserves the benefits derived from the functional diagnostics. As a result, logic designers and manufacturing engineers can concentrate on higher level problems, and field engineers can repair and bring a broken machine back on-line faster.

The structural test process has also produced an automatic test data generator. This generator is flexible enough to support testing for Digital's future processor designs, which include a scan system. This tool will prove to be essential in bringing the next innovative complex design to market on time. It makes design testing, prototype debugging, and repair more thorough and efficient.

Structural testing cannot address the unique problems presented by intermittent faults. These faults require constant monitoring and a mechanism to log the symptoms and isolate over time.

Symptom-directed Diagnosis

As computer systems have become more complex, the occurrence of intermittent faults has increased dramatically. This phenomenon results mainly from the increasing densities of chips and interconnections. Traditional test-directed diagnostics are ineffective in isolating intermittent faults because they rely on the ability to re-create the failure condition, which is seldom possible to do. Intermittent faults are usually as a result of marginal components and may only occur when certain conditions are met, such as the specific workload on the system. In contrast to test-directed diagnostics, symptom-directed diagnosis uses symptom information saved at the time of the failure to isolate the fault. Symptom information includes useful machine states, such as error detector states, multiplexer select values, memory addresses, and register values.

The VAX 9000 symptom-directed diagnosis strategy is composed of four major components. First, on-line hardware error detectors are used to achieve maximum coverage and an error-reporting process logs the necessary symptom information when errors are detected. Second, hardware error detectors and secondary syndromes are used to build symptom-directed diagnosis fault isolation rules that achieve the minimum possible callout of faulty field replaceable units. Third, symptom-directed diagnosis CAD tools calculate the coverage provided by on-line error detection and evaluate the quality of fault isolation provided by these detectors. Fourth, on-line, symptom-directed diagnosis software performs fault isolation for both single-error and multiple-error events.

Fault Detection Coverage and Error Logging

On-line hardware error detection is essential for detecting intermittent faults. On high availability mainframes such as the VAX 9000 system, it is essential to detect or "cover" a high percentage of intermittent faults. This section discusses the coverage measurement of on-line error detection for the VAX 9000 system. The VAX 9000 system error-logging process is also discussed.

On-line Error Detection

Hardware components are subject to temporary failures because of signal noise, environmental deviations, marginal devices, and other factors. To compensate for these inevitabilities, the VAX 9000 kernel includes over 450 error latches, which store the results from hardware error-detection circuits, such as parity and error-correcting code checkers. The detection of faults is critical to an orderly and predictable error-handling process. Error-detection circuits not only ensure the data integrity of the system, but also provide information that can be used for symptom-directed diagnosis fault isolation.

The placement of error-detection hardware is critical to the effectiveness of the process. The goals for error detector placement on the VAX 9000 system included:

- Maximizing coverage of higher failure components
- Minimizing the callout of faulty field replaceable units

- Minimizing pin use and cell count
- Minimizing effects on system performance

Coverage Calculation

One of the purposes of hardware error detection is to ensure that the VAX 9000 system behaves in a predictable manner when a fault occurs. Therefore, a high percentage of errors must be covered by on-line error detection. If a fault is not detected, then the machine may operate or fail in an unpredictable manner. Undetected faults complicate the error reporting and recovery processes and limit the quality of the symptom information available for symptom-directed diagnosis fault isolation.

Reliability Weighted Coverage The coverage provided by on-line error detectors is measured in terms of the reliability of the various components in the design. In other words, the coverage calculation is weighted according to the probability of failure of each device in the logic.

Reliability weighting is performed by first assigning a relative failure weight to each primitive physical element. Examples of primitive physical elements are gate array cells, self-timed RAM cells, the high-density signal carrier, multichip unit flex connectors, and planar module etch. A weight of one is assigned to the most reliable primitive physical element and all others are scaled proportionally upward.

Each signal in the machine is then assigned a failure weight by calculating the sum of the weights of each of the primitive elements that compose the signal. For example, a multichip unit interconnect signal is composed of two multichip unit flex connector primitive elements and one planar module etch primitive element. Therefore, the weight of this signal would be two times the multichip unit flex connector weight plus one times the planar module etch failure weight.

Probability of Detection The second aspect of the coverage calculation is the probability that a fault on a given signal will be detected by an on-line error detector. This aspect is called the signal probability of detection and is calculated by computing an error domain for each on-line error detector in the system. The error domain of a given detector is the sum of all of the signals in the design that have a greater than zero probability of being detected if they are faulted. The detector covers each signal in its error domain.

Computation of the error domains for each on-line error detector in the design results in a signal probability of detection for each covered signal. Uncovered signals are assigned a zero signal probability of detection.

Coverage Formula The signal probability of detection data and the signal reliability weight calculations are used to determine the system on-line error detector coverage. The formula for this calculation is

$$C = \frac{\sum_{i=1}^{n}\left(P_i \times RW_i\right)}{\sum_{i=1}^{n}\left(100_i \times RW_i\right)}$$

where n equals the number of signals in the system, p equals the signal probability of detection, RW equals the signal reliability weight, and C equals the system coverage.

A symptom-directed diagnosis CAD tool, called the hardware isolation domain evaluator (HIDE), was developed to automate the process of determining the signal reliability weights, probabilities of detection, and overall system coverage. HIDE is discussed in more detail in the CAD Tools and Processes section of this paper.

VAX 9000 Error-reporting Process

The error-reporting process on the VAX 9000 system facilitates symptom-directed diagnosis by saving critical symptom information that can be used for fault isolation. The VAX 9000 service processor unit initiates and controls the error-reporting process. The service processor unit monitors each of the VAX 9000 subsystems and reports conditions that deviate from normal operation. The service processor unit recovers the failed status from the subsystem in error and generates an error log entry, which contains important machine-state symptom information saved at the time that the error was detected. This information is analyzed by symptom-directed diagnosis fault isolation tools to determine the source of the error.

Fault Isolation Rules

The symptom-directed diagnosis fault isolation tools use a knowledge base of fault isolation rules to determine how to analyze the data inside the error log entry. The fault isolation rules were designed by reliability engineering experts who understand the behavior of the machine when it fails.

There are two basic types of fault isolation rules, single event and multiple event. Single-event rules are used for analyzing single error events (i.e., one error log entry). Multiple-event rules are used for analyzing multiple error events that occur over a specified interval of time.

Single Event Fault Isolation Rules

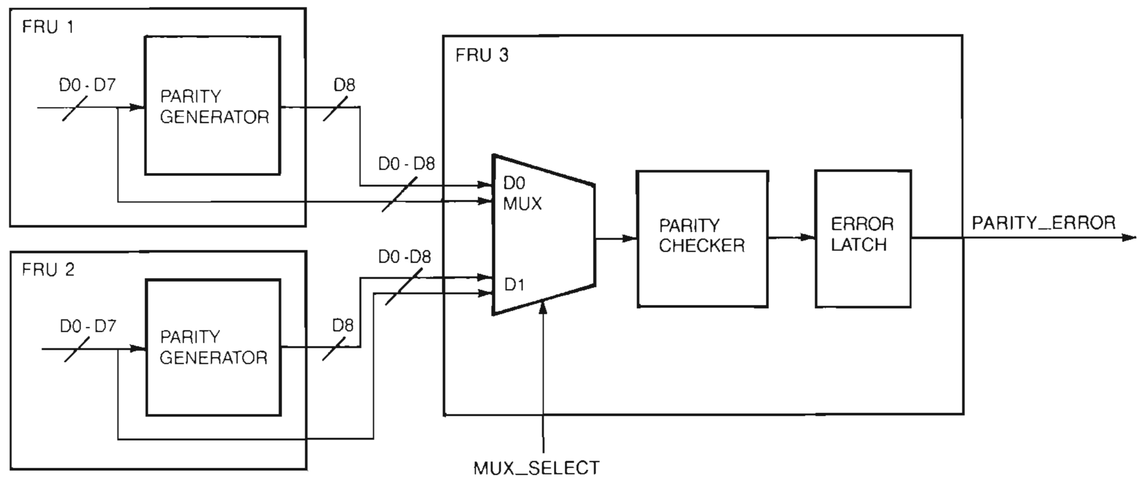
There are several categories of single event fault isolation rules. These rules are derived from the on-line error detection designed into the VAX 9000 system.

Primary Syndrome Fault Isolation Rules Primary syndromes are the error latches that detect and report error events. Each error latch stores the result of an on-line error detector. Each error detector covers a section of logic in the system. By mapping this logic to the physical partition (i.e., field replaceable units), the values of set error latches can be used as a first-pass fault isolation. In many instances, this analysis alone is sufficient to determine the faulty field replaceable unit.

Secondary Syndrome Fault Isolation Rules In some instances, the fault isolation provided by the primary syndromes may not localize the fault sufficiently. For example, if the primary syndrome field replaceable unit callout results in more than one field replaceable unit having a significant possibility of failure, then secondary syndromes must be used to reduce the callout. Secondary syndromes are key machine states, other than error latches, that are stored in the error log entry. Examples of secondary syndromes include multiplexer select lines, memory address values, and other path-sensitive control signals. These signal states are used to determine the specific path that was sensitized when an error occurred. The nonsensitized path(s) can then be removed from the callout. An example of how secondary syndromes are used for fault isolation is shown in Figure 4.

Fault Propagation Rules Sometimes a single-error event can trigger multiple error detectors because of fault propagation or domain intersection.

Fault propagation occurs when a fault in a given error domain (i.e., the propagation source) propagates into other error domains (i.e., the propagation destinations). To identify the real source of the error, the possible fault propagation paths must be found and the precedence of the error detectors in each propagation path must be identified. When multiple error latches are set, the propagation rules can then be applied to eliminate all propagation



PARITY_ERROR	MUX_SELECT	CALLOUT
1	UNKNOWN	FRU 1
		FRU 2
		FRU 3
1	0	FRU 1
		FRU 3
1	1	FRU 2
		FRU 3

Figure 4 Secondary Syndrome Example: MUX Select Used for Fault Isolation Refinement

destinations for each propagation source in the callout. An example of fault propagation is shown in Figure 5.

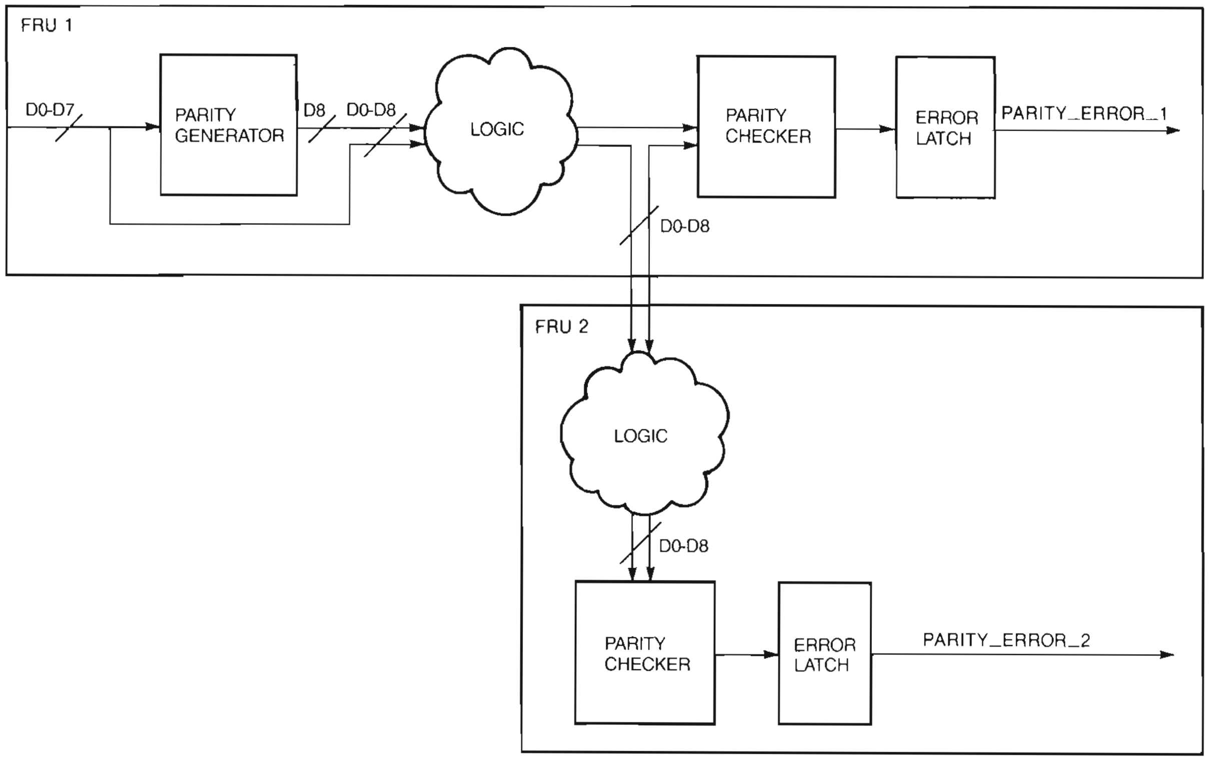
Domain Intersection Rules Domain intersection results when two or more error detectors cover a common piece of logic. This information is used to refine the callout when multiple error latches are set in the VAX 9000 system as shown in Figure 6.

Multiple Event Fault Isolation Rules

Multiple-event rules attempt to correlate separate error events to find a common problem. This type of analysis is beneficial when an intermittent or transient problem is not diagnosed sufficiently by single-event symptom-directed diagnosis rules.

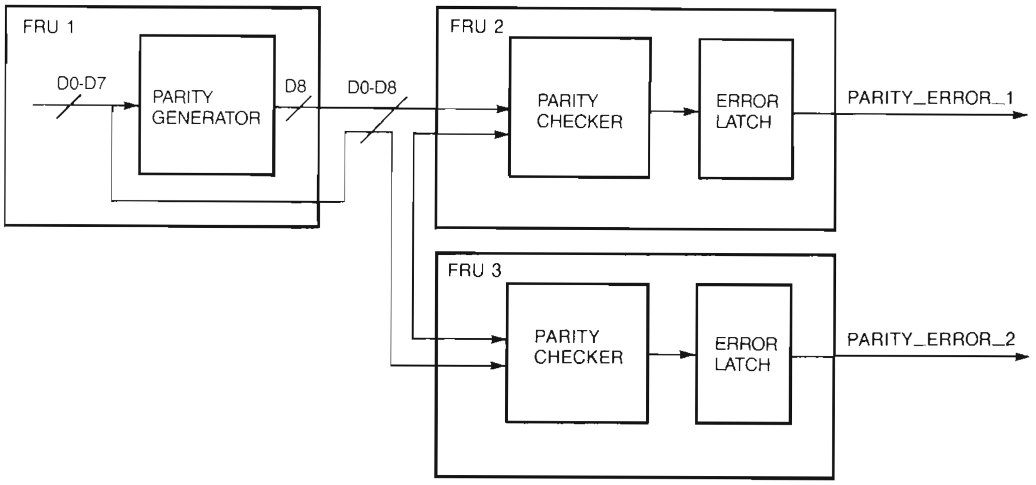
For example, if a logic fault were analyzed with single-event, symptom-directed diagnosis rules, an intermittent logic fault could be concluded as having occurred. Such an analysis would result in a callout of the faulty field replaceable unit. However, multiple-event rules include checking for certain environmental deviations in close proximity to a logic fault. In this case, multiple-event analysis would attempt to correlate the logic fault with the environmental deviations to determine if the fault is transient in nature. If this were the case, a callout would not be required.

Multiple-event rules can also be used to enforce the callout refinement provided by secondary syndromes, fault propagation, and domain intersection. For example, in a VAX 9000 system that repeatedly generates identical or similar error log entries, multiple event analysis can correlate these entries to a single intermittent fault. It can provide a scenario of which is the most likely secondary syndrome path to be sensitized and the most likely error domain to detect the error first. In this case,



	PARITY_ERROR_1	PARITY_ERROR_2	CALLOUT
NO PROPAGATION INFORMATION	1	1	FRU 1 FRU 2
WITH PROPAGATION INFORMATION	1	1	FRU 1

Figure 5 Fault Propagation Example



PARITY_ERROR_1	PARITY_ERROR_2	CALLOUT
1	0	FRU 1 FRU 2
0	1	FRU 1 FRU 3
1	1	FRU 1

Figure 6 Domain Intersection Example

multiple-event analysis can view these events as a single problem rather than seeing each error log entry in isolation.

CAD Tools and Processes

To ensure that the VAX 9000 symptom-directed diagnosis fault coverage and isolation goals were achieved, CAD tools were needed to measure the quality of the on-line error detection in the design. Tools also were needed to help develop symptom-directed diagnosis fault isolation rules and to facilitate the conversion of these rules into a format that could be used by the fault isolation software.

Some of the significant symptom-directed diagnosis CAD tools that were developed and used for the VAX 9000 system are discussed below.

Hardware Isolation Domain Evaluator

The hardware isolation domain evaluator (HIDE) CAD tool was developed to provide symptom-directed diagnosis fault coverage and isolation information to the VAX 9000 logic designers. HIDE also can generate simple symptom-directed diagnosis fault isolation rules for use in the system fault isolation matrices.

One of the goals for HIDE was to provide early feedback to logic designers on the quality of on-line error detection in designs. Early feedback gave designers time to make design changes if coverage or isolation goals were not achieved. Further, the information provided by HIDE helps designers select locations for error detectors and gave designers quick feedback on the implications of detector placement and design changes.

Symptom Diagnosis Information Language

The symptom-directed diagnosis fault isolation rules for the VAX 9000 system were coded into a set of system fault isolation matrix files, called symptom diagnosis information files. Symptom diagnosis information is a language that is designed to express both single-event and multiple-event, symptom-directed diagnosis fault isolation rules in an objective and consistent manner.

In earlier VAX systems, new fault isolation tools were needed for each new computer system. In the VAX 9000 system, the symptom diagnosis information language provides a general-purpose means to specify symptom-directed diagnosis fault isolation rules. The files are used as the rule base for the symptom-directed diagnosis fault isolation tools, which means that the tools can be used for future computer system designs.

On-line Fault Isolation Software

The VAX 9000 system contains on-line symptom-directed diagnosis software that automatically diagnoses faults as they occur. The software produces an isolation callout of the possible faulty field replaceable units that is automatically received by Digital customer service centers through a symptom-directed diagnosis reporting process. This process is designed to minimize the repair time for VAX 9000 systems. It automatically notifies Digital of problems and provides a repair plan to Customer Services before personnel are sent to the customer's site.

Service Processor Diagnostic

The VAX 9000 service processor unit contains a symptom-directed diagnosis fault isolation process that performs single-event analysis. This process runs in the background waiting for error log entries. When an error log entry is generated, the process analyzes the error log entry and produces an encoded callout of possible faulty field replaceable units.

The symptom-directed diagnosis fault isolation algorithm is performed by a general-purpose diagnostic engine. This engine uses a binary version of the symptom diagnosis information file, i.e., binary-coded matrix, as a rule base for its analysis. The diagnostic engine can analyze any error log entry that has a valid corresponding binary-coded matrix file.

In addition to the encoded callout, the single-event fault isolation process produces status information from each error event that is used for multiple-event analysis.

VAXsimPLUS

The VAXsimPLUS tool runs on the VAX 9000 CPU and performs symptom-directed diagnosis multiple-event analysis. The tool analyzes information generated by the single-event, symptom-directed diagnosis process using multiple-event, binary-coded matrix files. The VAXsimPLUS tool uses the same general-purpose diagnostic engine as the single-event, symptom-directed diagnosis process. The output of the VAXsimPLUS tool is a syndrome entry that collapses several error events into a single error analysis theory.

Summary

A complete test and diagnosis strategy for a large computer system, such as the VAX 9000 system, requires off-line testing and its on-line counterpart, symptom-directed diagnosis. Off-line testing provides a hierarchical mechanism for testing each component before it is assembled into the next level. In off-line testing, the use of the scan system provides high coverage and accurate fault isolation. Scan testing also has proven effective during all phases of the VAX 9000 system product development: design, manufacturing, prototype debug, and customer support.

Symptom-directed diagnosis is a sophisticated tool that provides detection and isolation of intermittent faults. Intermittent faults have been a significant problem in the past because of the difficulty to re-create the conditions that lead to such faults. Symptom-directed diagnosis solves the problem of intermittent faults by analyzing symptom information generated by on-line error handlers rather than by attempting to re-create the fault. Thus, the use of symptom-directed diagnosis provides greater machine availability for the VAX 9000 system.

Acknowledgments

The implementation of the VAX 9000 fault detection and isolation strategy would have been impossible if not for the perseverance and dedication to high quality shown by the following people: Jeff Barry, Dominic Carr, Steve Conway, Ed Crowley, Betty Daley, Tony Dancona, Dave D'Antonio, Chris Demos, Sue DesMarais, Paul Dormitzer, Rick Dusek, Mike Evans, Skip Gaede, Mike Gavronsky, Philippe Girard, Matt Goldman, Francis Gravel, Chris Joseph, Dale Keck, Tom Krehel, Charlie Kretz, Butch Leitz, Helen Lenane, Paul Leveille, Keith Mayhue, Chris McCabe, Robert Nobrega, Mike Newman, Paul Paternoster, Brian Rost, Dan Schullman, Scott Sitterly, Norm Sozio, Tamar Wexler, Tom Winter, Ted Wojcik, Richard Wood, Eugene Xia, and the members of the MCU tester and MCA3 test development teams.

General References

A. Miczo, *Digital Logic Testing* (New York: Harper and Row Publishers, Inc., 1986).

N. Tendoikar and R. Swann, "Automated Diagnostic Methodology for the IBM 3081 Processor Complex," *IBM Journal of Research and Development*, vol. 26, no. 1 (January 1982): 78–88.

H. Tanaka et al., "System Level Fault Dictionary Generation," *IEEE International Test Conference Proceedings* (New York, 1988): 884–887.

M. Goldman et al., "The VAX 9000 Service Processor Unit," *Digital Technical Journal*, vol. 2, no. 4 (Fall 1990, this issue): 90–101.

Further Readings

The Digital Technical Journal *publishes papers that explore the technological foundations of Digital's major products. Each Journal focuses on at least one product area and presents a compilation of papers written by the engineers who developed the product. The content for the Journal is selected by the Journal Advisory Board, which includes four vice presidents and five senior engineering managers.*

Topics covered in previous issues of the *Digital Technical Journal* are as follows:

DECwindows Program
Vol. 2, No. 3 Summer 1990
An overview and descriptions of the enhancements Digital's engineers have made to MIT's X Window System in such areas as the server, toolkit, interface language, and graphics, as well as contributions made to related industry standards

VAX 6000 Model 400 System
Vol. 2, No. 2, Spring 1990
The highly expandable and configurable midrange family of VAX systems that includes a vector processor, a high-performance scalar processor, and advances in chip design and physical technology

Compound Document Architecture
Vol. 2, No. 1, Winter 1990
The CDA family of architectures and services that support the creation, interchange, and processing of compound documents in a heterogeneous network environment

Distributed Systems
Vol. 1, No. 9, June 1989
Products that allow system resource sharing throughout a network, the methods and tools to evaluate product and system performance

Storage Technology
Vol. 1, No. 8, February 1989
Engineering technologies used in the design, manufacture, and maintenance of Digital's storage and information management products

CVAX-based Systems
Vol. 1, No. 7, August 1988
CVAX chip set design and multiprocessing architecture of the midrange VAX 6200 family of systems and the MicroVAX 3500/3600 systems

Software Productivity Tools
Vol. 1, No. 6, February 1988
Tools that assist programmers in the development of high-quality, reliable software

VAXcluster Systems
Vol. 1, No. 5, September 1987
System communication architecture, design and implementation of a distributed lock manager, and performance measurements

VAX 8800 Family
Vol. 1, No. 4, February 1987
The microarchitecture, internal boxes, VAXBI bus, and VMS support for the VAX 8800 high-end multiprocessor, simulation, and CAD methodology

Networking Products
Vol. 1, No. 3, September 1986
The Digital Network Architecture (DNA), network performance, LANbridge 100), DECnet-ULTRIX and DECnet-DOS, monitor design

MicroVAX II System
Vol. 1, No. 2, March 1986
The implementation of the microprocessor and floating point chips, CAD suite, MicroVAX workstation, disk controllers, and TK50 tape drive

VAX 8600 Processor
Vol. 1, No. 1, August 1985
The system design with pipelined architecture, the I-box, F-box, packaging considerations, signal integrity, and design for reliability

Subscriptions to the *Digital Technical Journal* are available on a yearly, prepaid basis. The subscription rate is $40.00 per year (four issues). Requests should be sent to Cathy Phillips, Digital Equipment Corporation, MLO1-3/B68, 146 Main Street, Maynard, MA 01754, U.S.A. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

Single copies and past issues of the *Digital Technical Journal* can be ordered from Digital Press at a cost of $16.00 per copy.

Technical Papers and Books by Digital Authors

Dileep Bhandarkar and Richard Brunner, "VAX Vector Architecture," *Proceedings of 17th Annual International Symposium on Computer Architecture* (IEEE, May 1990): 204–215.

David G. Shurtleff and Colin Strutt, "Extensibility of an Enterprise Management Director," in *Network Management and Control,* edited by A. Kershenbaum, M. Malek, and M. Wall (New York: Plenum Press, 1990): 129–141.

Xi-Ren Cao, "System Representations and Performance Sensitivity Estimates of Discrete Event Systems," *Mathematics and Computers in Simulation,* vol. 31 (1989): 113–122.

Xi-Ren Cao, "On a Sample Performance Function of Jackson Queueing Networks," *Operations Research,* vol. 36, no. 1 (1988): 128–136.

Xi-Ren Cao and Y. C. Ho, "Estimating Sojourn Time Sensitivity in Queueing Networks Using Perturbation Analysis," *Journal of Optimization Theory and Applications,* vol. 53, no. 3 (1987): 353–375.

Xi-Ren Cao, "First-Order Perturbation Analysis of a Single Multi-Class Finite Source Queue," *Performance Evaluation,* vol. 7 (1987): 31–41.

D. Lomet and B. Salzberg, "Access Method for Multiversion Data," *Proceedings of ACM SIGMOD Conference* (May 1989): 315–324.

D. Lomet and B. Salzberg, "A Robust Multi-attribute Search Structure," *Proceedings of International Conference on Data Engineering* (February 1989): 296–304.

D. Lomet, "A Simple Bounded Disorder File Organization with Good Performance," *ACM Transactions on Database Systems 13,* vol. 4 (December 1988): 525–551.

P. Bernstein and D. Lomet, "CASE Requirements for Extensible Database Systems," *Data Engineering,* vol. 10, no. 2 (June 1987): 2–9.

W. Linwin and D. Lomet, "A New Method for Fast Data Search with Keys," *IEEE Software,* vol. 4, no. 2 (March 1987): 16–24.

D. Lomet, "Partial Expansions for File Organizations with an Index," *ACM Transactions on Database Systems,* vol. 12, no. 1 (March 1987): 65–84.

Digital Press

Digital Press is the book publishing group of Digital Equipment Corporation. Digital Press publishes books internationally for computer professionals, specializing in the areas of networking and data communication, artificial intelligence, computer integrated manufacturing, windowing systems, and the VMS operating system. Digital Press welcomes proposals and ideas in these and related areas.

VAX/VMS: Writing Real Programs in DCL
Paul C. Anagnostopoulos, 1989, softbound, 409 pages ($29.95)

X WINDOW SYSTEM TOOLKIT: The Complete Programmer's Guide and Specification
Paul J. Asente and Ralph R. Swick, 1990, softbound, 1,000 pages ($44.95)

UNIX FOR VMS USERS
Philip E. Bourne, 1990, softbound, 368 pages ($28.95)

INFORMATION TECHNOLOGY STANDARDIZATION: Theory, Practice, and Organizations
Carl F. Cargill, 1989, softbound, 252 pages ($24.95)

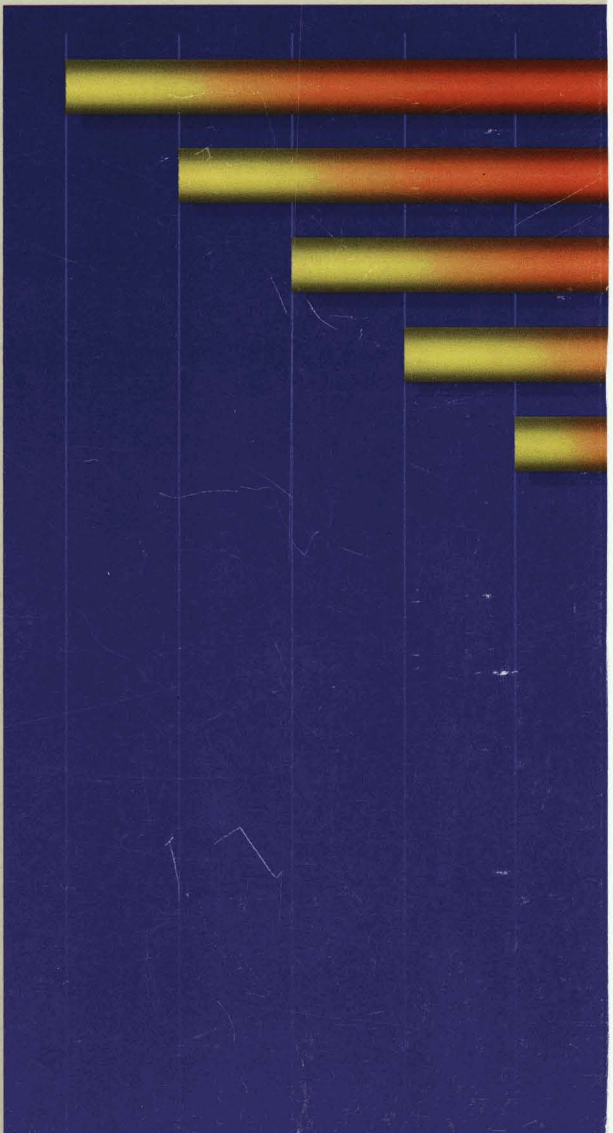
THE DIGITAL GUIDE TO SOFTWARE DEVELOPMENT
Corporate User Publication Group of Digital Equipment Corporation, 1990, softbound, 239 pages ($27.95)

VMS INTERNALS AND DATA STRUCTURES: Version 5 Update Xpress, Volumes 1,2,3,4,5
Ruth E. Goldenberg and Lawrence J. Kenah, 1989, 1990, 1991, all softbound ($27.95)

VAX/VMS INTERNALS AND DATA STRUCTURES: Version 4.4
Lawrence J. Kenah, Ruth E. Goldenberg, and Simon F. Bate, 1988, softbound, 979 pages ($75.00)

THE USER'S DIRECTORY OF COMPUTER NETWORKS
Tracy L. LaQuey, 1990, softbound, 630 pages ($34.95)

COMPUTER PROGRAMMING AND ARCHITECTURE: The VAX, Second Edition
Henry M. Levy and Richard H. Eckhouse, Jr., 1989, hardbound, 444 pages ($38.00)



ISSN 0898-901X