*Alpha AXP Architecture and Systems*

# Digital Technical Journal

Digital Equipment Corporation

Volume 4 Number 4
Special Issue 1992

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, 1 Burlington Woods Drive, Burlington, MA 01830-4597.

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, site location code, and address. 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-J886E-DP



Book production was done by Quantic Communications, Inc.

***Cover Design***

*The DECchip 21064, the first implementation of Digital's Alpha AXP computer architecture, is the world's fastest single-chip microprocessor. Represented on our cover by the AXP logo, the DECchip takes its place among symbols of other devices from computing history, including the vacuum tube, a punch card, sketches of Babbage's Analytical Engine, a wheel from the Pascaline, and an abacus.*

*The cover was designed by Deborah Falck of Digital's Corporate Human Factors Group with the help of Kaza Design.*

# Contents

# Editor's Introduction

**Jane C. Blake**
*Editor*

This special issue of the *Digital Technical Journal* presents the computer architecture that Digital believes will become the universal platform for computing over the next 25 years. A significant milestone in the company's history, the Alpha AXP architecture arises out of Digital's extensive engineering experience and puts into place a cohesive, flexible framework for high-performance 64-bit RISC computing. This issue contains papers representative of the scope of the program across Digital's Engineering organization, including hardware systems, an operating system, compilers, binary translators, network and database software, and simulators.

The results of the engineering efforts discussed in these papers reflect three primary goals for the Alpha AXP architecture: high performance, longevity, and easy migration from the 32-bit VAX VMS computer line. Dick Sites, one of the chief Alpha AXP architects, has written a definitive paper that explains how key architectural decisions were made relative to the goals. He reviews the similarities and differences between the AXP architecture and other RISC architectures, and then presents details of the design, including data and instruction formats. In his conclusion, he projects evolutionary changes in the architecture and the resulting performance increases of a thousandfold over the next 25 years.

The first implementation of the Alpha AXP architecture is the DECchip 21064 microprocessor, which can execute up to 400 million operations per second. Dan Dobberpuhl and members of the Alpha chip team offer an overview of the CMOS process technology, the chip microarchitecture, and the external interface. They then detail the circuit implementation and explain the design choices directed toward meeting architectural performance requirements and to allow application flexibility. The result of their design efforts is a microprocessor that operates at speeds up to 200 MHz—the fastest commercially available chip in the industry.

Early implementations of this chip became part of a prototype system, the Alpha Demonstration Unit. As Chuck Thacker, Dave Conroy, and Larry Stewart explain in their paper, the prototype served the overall Alpha AXP program by giving software developers early access (ten months) to AXP-compliant hardware. Because of the architectural emphasis on multiple processors, prototype designers focused on delivering a robust multiprocessing system. The authors discuss the significance of the choice of a backplane interconnect for a multiprocessor, compare different approaches to cache coherence, and describe the system modules and packaging.

With constraints different from those of the prototype, the hardware product projects are represented here by three different implementations: desktop, departmental, and data center systems. In the desktop area, the DEC 3000 AXP family of workstations are balanced uniprocessor systems. Todd Dutton, Dan Eiref, Hugh Kurth, Jim Reisert, and Robin Stewart review the decision to replace the traditional common system bus with a crossbar system interconnect constructed of ASICs. This new interconnect allowed the designers to meet the goals of low memory latency, high memory bandwidth, and minimal CPU–I/O memory contention in a cost-competitive manner.

The DEC 4000 AXP system is a departmental server that implements the IEEE Futurebus+ standard. Barry Maskas, Stephen Shirron, and Nick Warchol present the reasoning behind the system architecture and technology decisions that resulted in the achievement of optimized uniprocessor performance, dual-processor symmetric multiprocessing, and balanced I/O throughput. Details of the subsystems that make up this expandable modular system are also provided.

The DEC 7000 and DEC 10000 systems are powerful mid-range and mainframe platforms intended for large commercial applications and designed to utilize multiple future generations of the DECchip. Described by Brian Allison and Catharine van Ingen, the heart of these systems is a high-performance interconnect that allows communications between multiple processors, memory arrays, and I/O subsystems. The authors review each of the modules and the I/O subsystem design, which includes interfaces for XMI and Futurebus. Notably, a 32-bit VAX CPU module has been designed to the

requirements of the high-performance system interconnect. Users who wish to migrate from the VAX system to Alpha AXP need only swap module boards.

Migration to Alpha AXP from other architectures, in particular from VAX VMS, is one of the major goals set by the Alpha architects. Existing software—operating systems, languages, programs—must be adapted to run effectively on 64-bit RISC systems. A paper by Nancy Kronenberg, Tom Benson, Wayne Cardoza, Ravindran Jagannathan, and Ben Thomas addresses the challenges of porting the OpenVMS operating system—originally developed specifically for 32-bit VAX systems—to Alpha AXP systems. To deal with the huge amount of code, the project team developed a compiler that treats VAX assembly language (VAX MACRO-32) as a source language to be compiled. The authors also discuss the major architectural differences in the kernel, performance, and some future directions for the system.

The GEM compiler system is the technology Digital is using to build state-of-the-art compiler products. GEM is described here by David Blickstein, Peter Craig, Caroline Davidson, Neil Faiman, Kent Glossop, Rich Grove, Steve Hobbs, and Bill Noyce. A significant achievement in the development of this compiler is that a single optimizer is used for all languages and platforms. Developers of compilers will find in-depth information in the authors' discussions of optimization techniques, code generation, compiler engineering, and future enhancements.

Binary translation is another means of moving complex software applications from one architecture and operating system to another architecture and operating system. Two binary translators are the subject of a paper by Dick Sites, Anton Chernoff, Matthew Kirk, Maurice Marks, and Scott Robinson. The authors discuss the alternatives to translators, performance issues, and the development of the translators, VEST and mx, and the complementary run-time environments. VEST translates OpenVMS VAX images to OpenVMS AXP images, and mx translates ULTRIX/MIPS images to DEC OSF/1 AXP images.

An easy migration path to Alpha AXP for two database management systems used in large commercial applications is the subject of a paper by Jeff Coffler, Zia Mohamed, and Peter Spiro. The authors define the issues involved in porting the complex VAX DBMS and Rdb/VMS products to the AXP platform. Adding to the challenge but balanced by its advantages was the decision to have a common source, or single code, base. The authors review this design approach and provide details of the individual porting efforts.

The process of porting DECnet-VAX to the OpenVMS AXP operating system is described by Jim Colombo, Pam Rickard, and Paul Benoit. They discuss the DECnet features supported in the operating system, the software techniques used, and the importance of the decision to build common code for the VAX and Alpha AXP systems. The authors share details of the port and lessons learned that can be applied to future porting efforts.

Complementary to the previously mentioned prototype hardware system are four software simulators that enabled engineers to develop software for Alpha AXP concurrently with hardware development. Described by George Darcy, Ron Brender, Steve Morris, and Mike Iles, the Mannequin simulator was used by the OpenVMS group to boot the entire operating system and debug utilities; the ISP simulator was used by the DEC OSF/1 group with similar success. A major section of the paper focuses on the Alpha User-mode Debugging Environment in which user-mode code being developed for Alpha AXP platforms can be compiled and executed as Alpha AXP code.

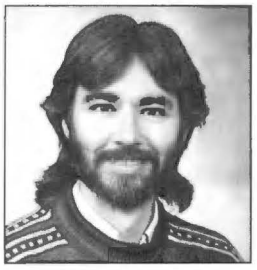
The closing paper is an unusual one for the *Journal* because it addresses engineering management, not strictly technical issues. Peter Conklin offers insights into the reasons for the success of one of the largest engineering programs undertaken in the industry. He defines the enrollment management model used for the Alpha AXP program and explains key concepts, including the program office and project "cusps."

The editors are very grateful for the help of Bob Supnik, Vice President and Corporate Consultant, in planning this special issue and for writing its Foreword.

We are also pleased to note that four papers in this issue are being copublished with the *Communications of the ACM*, including those on the Alpha AXP architecture, the Alpha Demonstration Unit, OpenVMS AXP, and binary translation. Barbara Watterson from Digital's semiconductor organization; Diane Crawford, Executive Editor of the CACM; the DTJ editors; and the authors cooperated so that these informative papers could be made available to a broad technical audience.

Jane Blake

# Biographies



**Brian R. Allison** Brian Allison is a senior consultant engineer for Digital's mid-range VAX/Alpha AXP systems group and is the system architect responsible for the coordination of the VAX and DEC 7000 and 10000 system definition and design. Prior to this work, he served as system architect for the VAX 6000 product. Brian holds a B.S.E.E. and a B.S.C.S. from Worcester Polytechnic Institute (1977).

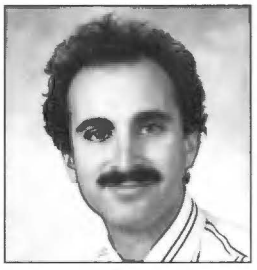
**Randy Allmon** After receiving a B.S. degree in electrical engineering from the University of Cincinnati, Randy Allmon joined Digital in 1981. As a circuit designer in the Semiconductor Engineering Group, he has contributed to the development of numerous high-performance CMOS processors. Currently, Randy is responsible for the technical design and management of a next-generation processor based on the Alpha AXP architecture. He is the coauthor of four high-performance processor papers given at ISSCC and has one patent pending.

**Robert Anglin** Robert Anglin received S.B. and S.M. degrees in electrical engineering in 1989 from the Massachusetts Institute of Technology. In the same year, he joined Digital's Semiconductor Engineering Group, where he has worked on the design of high-performance microprocessors. Robert is a member of Sigma Xi. He is currently pursuing an M.B.A. degree at Harvard University.

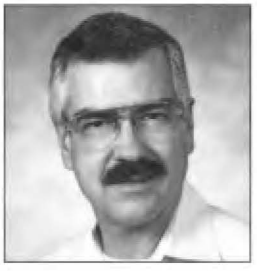
**Paul Benoit** Paul Benoit is a principal software engineer in the Networks and Communications Group. He is the project/technical leader for the DECnet for OpenVMS AXP project; the team received an Alpha Achievement Award for early completion of project commitments. Previous to this, Paul led the DECnet-VAX Phase IV effort. He holds an M.S.S.E. (1991) from Boston University and a B.S.C.S. (1986) from the University of Lowell. Paul is a member of ACM and IEEE Computer Society.

**Thomas R. Benson** A consulting engineer in the OpenVMS AXP Group, Tom Benson was the project leader and principal designer of the VAX MACRO-32 compiler. Prior to his Alpha AXP contributions, he led the VMS DECwindows FileView and Session Manager projects and brought the Xlib graphics library to the VMS operating system. Earlier, he supported an optimizing compiler shell used by several VAX compilers. Tom joined Digital's VAX Basic project in 1979, after receiving B.S. and M.S. degrees in computer science from Syracuse University. He has applied for four patents related to his Alpha AXP work.

**David Bertucci** David Bertucci received a B.S.E.E. degree in 1982 from Wayne State University and an M.S.E.E. degree in 1988 from Michigan State University. He joined Digital's Semiconductor Engineering Group in 1989 and worked on advanced CMOS microprocessor design. Currently, he is employed at Sun Microsystems, Inc.

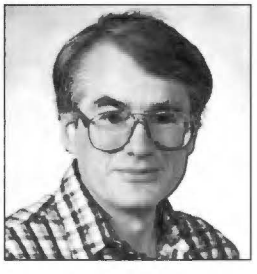


**David S. Blickstein** Principal software engineer David Blickstein has worked on optimizations for the GEM compiler system since the project began in 1985. During that time, he designed various optimization techniques, including induction variables, loop unrolling, code motions, common subexpressions, base binding, and binary shadowing. Prior to this, David worked on Digital's PDP-11 and VAX APL implementations and led the VAX-11 PL/I project. He received a B.A. (1980) in mathematics from Rutgers College, Rutgers University, and holds one patent on side effects analysis and another on induction variable analysis.



**Ronald F. Brender** Ron Brender is a senior consultant software engineer, contributing to the GEM compiler back-end project in the Software Development Technologies Group. He has worked on compilers and programming language definition for Alpha AXP, VAX, PDP-11, and PDP-10 systems, including Ada, FORTRAN and BLISS. A member of various standards committees since the mid-1970s, Ron is now responsible for VAX and Alpha AXP calling standards. He joined Digital in 1970, after receiving a Ph.D. in computer and communication sciences at the University of Michigan.

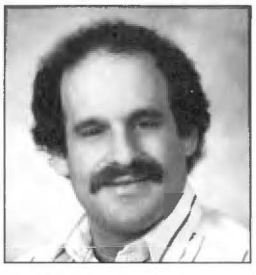
**Sharon Britton** Sharon Britton received a B.S.E.E. degree from Boston University in 1983 and an M.S.E.E. degree from the Massachusetts Institute of Technology in 1990. She joined Digital in 1983 to work on the design and development of 80186-based controllers for read-only and write-once optical disk drives. Sharon's graduate research involved the development of an integrated content addressable memory system with error detection capability. Currently a member of the Semiconductor Engineering Group, she is involved in the design and implementation of high-performance CMOS microprocessors.



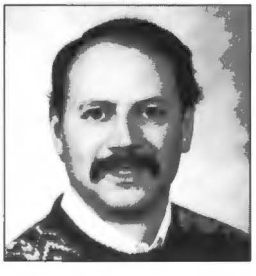
**Wayne M. Cardoza** Wayne Cardoza is a senior consultant engineer in the OpenVMS AXP Group. Since joining Digital in 1979, he has worked in various areas of the OpenVMS kernel. Wayne was also one of the architects of PRISM, an earlier Digital RISC architecture; he holds several patents for this work. More recently, Wayne participated in the design of the Alpha AXP architecture and was a member of the initial design team for the OpenVMS port. Before coming to Digital, Wayne was employed by Bell Laboratories. Wayne received a B.S.E.E. from Southeastern Massachusetts University and an M.S.E.E. from MIT.

**Linda Chao** Linda Chao received a B.S.E.E. degree from the Massachusetts Institute of Technology in 1987. Since joining Digital in the Semiconductor Engineering Group/Advanced Development in 1987, Linda has been engaged in the design of microprocessors based on the VAX and Alpha AXP architectures. She is currently pursuing master's degrees in electrical engineering and management through the MIT Leaders for Manufacturing Program.

**Anton Chernoff** Anton Chernoff is a member of the technical staff at Digital Equipment Corporation, working in the Alpha AXP Migration Tools Group. He joined Digital in 1991, but also worked at Digital between 1973 and 1981 as project leader and developer of the RT-11 and RSTS/E operating systems. Anton spent 1982 through 1991 at Liant Software Corporation as a senior consulting engineer in compiler and debugger development.



**Jeffrey A. Coffler** A principal software engineer in the Database Systems Engineering Group, Jeff Coffler led the effort to port DBMS to the Alpha AXP platform. Prior to this, Jeff worked on the DBMS and Rdb backup/restore facility and on new DBMS features and maintenance. He is currently working on the project to port Rdb for OpenVMS to operating systems such as Windows NT and OSF/1. He has also contributed to the RSTS/E operating system, WPS-PLUS porting, and workflow management projects. Jeff joined Digital in 1984 and holds a B.S.C.S. (1983) from California State University at Northridge.



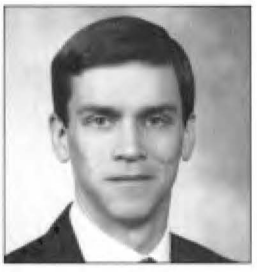
**James V. Colombo** Project/technical leader James Colombo is currently responsible for the next release of DECnet/OSI for OpenVMS for the VAX and Alpha AXP computing environments. Prior to this, he led the port of DECnet-VAX Phase IV to the OpenVMS AXP operating system; the team received an Alpha Achievement Award for early completion of the project. Jim also led the DECnet for OS/2 V1.0 and various PATHWORKS product efforts. Before coming to Digital in 1983, Jim worked at Prime Computer, Inc. and Computer Devices, Inc. He holds a B.S.C.S. from Boston University and is a member of IEEE.

**Peter F. Conklin** Peter Conklin is director of Alpha AXP Systems Development. Since joining Digital in 1969, he has held engineering management positions in large and small systems and terminals groups, direct hardware and software engineering, product management, base product marketing, quality management, and advanced development. Peter was the first software engineer on the VMS project in 1975, ran the VAX architecture team, and was instrumental in developing the key architectures and products for the VAX VMS layered product set. Peter received an A.B. in mathematics from Harvard University in 1963.

**Robert A. Conrad** Robert Conrad received a B.S. degree in electrical and computer engineering from the University of Cincinnati in 1984 and an M.S. degree in electrical and computer engineering from the University of Massachusetts in 1992. In 1981 he joined Digital's Semiconductor Engineering Group, where he worked as a co-op student in the Architecturally Focused Logic Group. Since 1984 Rob has been engaged in the research and development of VLSI microprocessors, including the MicroVAX CPU, a 50-MHz RISC CPU, and most recently the DECchip 21064 microprocessor.

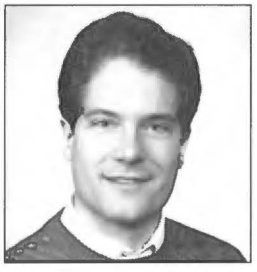
**David G. Conroy** Dave Conroy received a B.A.Sc. degree in electrical engineering from the University of Waterloo, Canada, in 1977. After working briefly in industrial automation, Dave moved to the United States in 1980. He cofounded the Mark Williams Company and built a successful copy of the UNIX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

**Peter W. Craig** Peter Craig is a principal software engineer in the Software Development Technologies Group. He is currently responsible for the design and implementation of a dependence analyzer for use in future compiler products. Peter was a project leader for the VAX Code Generator used in the VAX C and VAX PL/I compilers, and prior to this, he developed CPU performance simulation software in the VAX Architecture Group. He received a B.S.E.E. (magna cum laude, 1982) from the University of Connecticut and joined Digital in 1983.



**George A. Darcy III** As a senior software engineer in the Alpha Migration Tools Group, George Darcy has worked on the Mannequin Alpha AXP simulator, the VEST binary translator, and the Translated Image Environment (TIE) run-time library. In his ten years at Digital, he has also developed a virtual disk driver for the OpenVMS V5.0 SMP operating system, software behavioral models of a high-end VAX processor, and various simulation and CAD software tools. George received a B.S.C.E. (cum laude, 1984) from Boston University, where he was an Engineering Merit Scholar and a member of Tau Beta Pi.

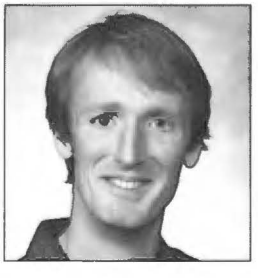
**Caroline S. Davidson** Since joining Digital in 1981, Caroline Davidson has contributed to several software projects, primarily related to code generation. Currently a principal software engineer, she is working on the GEM compiler generator project and is responsible for the areas of lifetimes, storage allocation, and entry-exit calls. Caroline is also a project leader for the Intel code generation effort. Her prior work involved the VAX FORTRAN for ULTRIX, VAX Code Generator, and FORTRAN IV software products. Caroline has a B.S.C.S. from the State University of New York at Stony Brook.



**Daniel E. Dever** Dan Dever received a B.S.E.E. degree in 1988 from the University of Cincinnati. He joined Digital's Semiconductor Engineering Group in 1988, where he worked on the design and logic verification of CMOS VAX microprocessors. Since 1990 he has been involved in the design of RISC architecture microprocessors, including the floating-point unit of the DECchip 21064 microprocessor. Dan is currently involved in the design of integer arithmetic logic for the next-generation processor based on the Alpha AXP architecture.

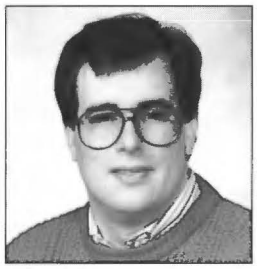
**Daniel W. Dobberpuhl** Dan Dobberpuhl received a B.S.E.E. degree from the University of Illinois in 1967. Subsequent to positions with the Department of Defense and General Electric Company, he joined Digital's Semiconductor Engineering Group in 1976. Since that time, he has been active in the design of four generations of microprocessors, including the first single-chip PDP-11 and the first single-chip VAX. Most recently, Dan was the project leader for the first VLSI implementation of Digital's new 64-bit Alpha AXP computing architecture. He is coauthor of the text, *The Design and Analysis of VLSI Circuits.*

**Todd A. Dutton** A principal hardware engineer, Todd Dutton was responsible for the overall design integration and timing verification of the DEC 3000 AXP Model 500. Prior to this, he led a team in developing vector processor hardware in the Advanced VAX Development Group. Todd joined Digital in 1987. Previously, he was employed at Numerix Corporation and at Signal Processing Systems, Inc. Todd has a B.S. degree in computer science from the Massachusetts Institute of Technology and was elected to Tau Beta Pi. He holds a patent on vector processor technology and has published two papers on vector processors.

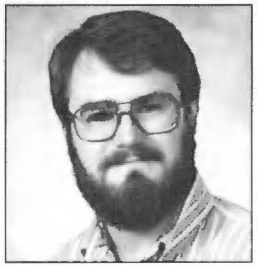


**Daniel Eiref** Dan Eiref joined Digital in 1987 after receiving B.S. and M.S. degrees in electrical engineering from Columbia University. At Columbia he was elected to Tau Beta Pi and was awarded the Steven Abbey Outstanding Student-athlete Award. He is currently attending Harvard Business School. A principal hardware engineer, Dan was responsible for the design of the memory and clock systems of the DEC 3000 AXP Model 500. He also designed the workstation's SLICE and ADDR ASICs. Prior to this project, he worked as an ECL hardware designer in the Advanced VAX Development Group.

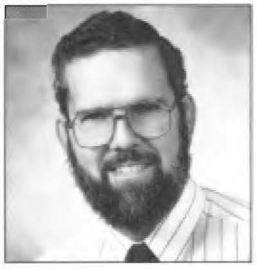
**R. Neil Faiman, Jr.** Neil Faiman is a consultant software engineer in the Software Development Technologies Group. He was the primary architect of the GEM intermediate language and a project leader for the GEM compiler optimizer. Prior to this work, he led the BLISS compiler project. Neil came to Digital in 1983 from MDSI (now Schlumberger/Applicon). He has B.S. (1974) and M.S. (1975) degrees in computer science, both from Michigan State University. Neil is a member of Tau Beta Pi and ACM, and an affiliate member of the IEEE Computer Society.



**Bruce Gieseke** Bruce Gieseke received a B.S. degree in electrical engineering from the University of Cincinnati in 1984, and an M.S. degree in electrical engineering from North Carolina State University in 1985. In 1986 he joined Digital's Semiconductor Engineering Group, where he has been engaged in the implementation and circuit design of RISC microprocessors.

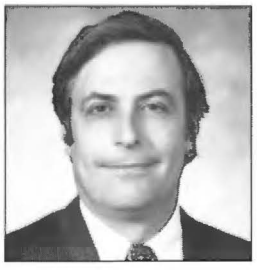


**Kent D. Glossop** Kent Glossop is a principal engineer in the Software Development Technologies Group. Since 1987 he has worked on the GEM compiler system, focusing on code generation and instruction-level transformations. Prior to this, Kent was the project leader for a release of the VAX PL/I compiler and contributed to version 1 of the VAX Performance and Coverage Analyzer. Kent joined Digital in 1983 after receiving a B.S. in computer science from the University of Michigan. He is a member of IEEE.



**Richard B. Grove** Senior consultant software engineer Rich Grove joined Digital in 1971 and is currently in the Software Development Technologies Group. He has led the GEM compiler project since the effort began in 1985, contributing to the code generation phases. Prior to this work, Rich was the project leader for the PDP-11 and VAX FORTRAN compilers, worked on VAX Ada V1, and was a member of the ANSI X3J3 FORTRAN Committee. He is presently a member of the design team for Alpha AXP calling standards and architecture. Rich has B.S. and M.S. degrees in mathematics from Carnegie-Mellon University.

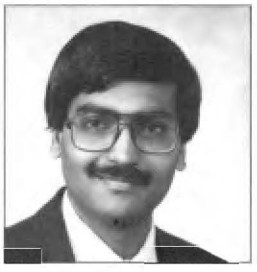
**Soha M.N. Hassoun** Soha Hassoun received a B.S.E.E. degree from South Dakota State University in 1986, and an S.M.E.E. degree from the Massachusetts Institute of Technology in 1988. From August 1988 to August 1991 she was employed at Digital as a custom design engineer in the Semiconductor Engineering Group. She contributed to the design of the floating-point unit of the DECchip 21064 processor. Soha was the recipient of a Digital Minority and Women's Scholarship in 1991 and is pursuing a Ph.D. degree at the University of Washington, Seattle, Computer Systems Engineering Department.



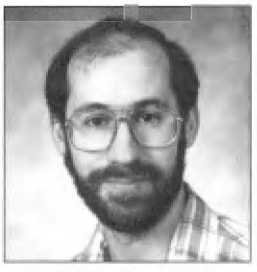
**Steven O. Hobbs** A member of the Software Development Technologies Group, Steven Hobbs is working on the GEM compiler project. In prior contributions at Digital, he was the project leader for VAX Pascal, the lead designer for the global optimizer in VAX FORTRAN, and a member of the Alpha AXP architecture design team. Steve received his A.B. (1969) in mathematics at Dartmouth College and while there, helped develop the original BASIC time-sharing system. He has an M.A. (1972) in mathematics from the University of Michigan and has done additional graduate work in computer science at Carnegie-Mellon University.

**Gregory W. Hoeppner** Gregory Hoeppner graduated with distinction from Purdue University in 1979. His research topic was ion-implanted optical waveguides. In 1980 he worked at General Telephone and Electronics Research Laboratory, where he performed basic properties research on GaAs for fabrication of submicrometer FETs. From 1981 to 1992 he held a number of positions at Digital Equipment Corporation's Hudson, MA site, including co-implementation leader of Digital's DECchip 21064. He is currently employed as a senior engineer at IBM, Advanced Workstation Division.

**Michael V. Iles** Michael Iles is a senior technology consultant at the UK Alpha AXP Migration Centre. Since joining Digital in 1975, Mike has worked in various field positions, in Advanced VAX development as a microcoder, and for VMS engineering as a software engineer. He worked on the migration of OpenVMS VAX to the Alpha AXP platform, designing and implementing a user-mode simulation environment that became AUD. Mike has a B.Sc. in electrical engineering (honors, 1973) from City University, London, and holds a patent for digital speech synthesis techniques. He has several patents pending for AUD.



**Ravindran Jagannathan** Ravindran Jagannathan is a principal software engineer in the OpenVMS Performance Group currently investigating OpenVMS AXP multiprocessing performance. Since 1986, he has worked on performance analysis and characterization, and algorithm design in the areas of OpenVMS services, SMP, VAXcluster systems, and host-based volume shadowing. Ravindran received a B.E. (honors, 1983) from the University of Madras, India, and M.S. degrees (1986) in operations research and statistics and in computer and systems engineering from Rensselaer Polytechnic Institute.



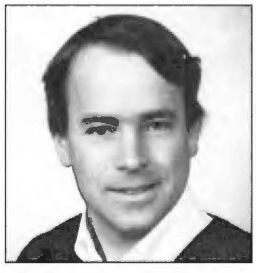
**Matthew B. Kirk** Matthew Kirk is a senior software engineer in the SEG/AD AXP Migration Tools Group, where he works on binary translator development, testing, and support. He joined Digital in 1986 and has also designed and developed automated architectural test software for pipelined VAX hardware and the CI computer interconnect. Matthew holds a B.S. in computer science (1986) from the University of Massachusetts.

**Nancy P. Kronenberg** Nancy Kronenberg joined Digital in 1978 and has developed VMS support for several VAX systems. She designed and wrote the VMS CI port driver and part of the VMScluster System Communications Services. In 1988, Nancy joined the team that investigated alternatives to the VAX architecture and drafted the proposal for the Alpha AXP architecture and for porting the OpenVMS operating system to it. Nancy is a senior consulting software engineer and technical director for the OpenVMS AXP Group. She holds an A.B. degree in physics from Cornell University.

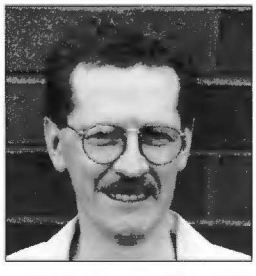
**Kathryn Kuchler** Kathryn Kuchler received a B.S. degree in electrical engineering from Cornell University in 1990. Upon graduation, she joined Digital's Semiconductor Engineering Group, where she worked on the first implementation of a RISC microprocessor based on the Alpha AXP architecture.

**Hugh R. Kurth** Hugh Kurth joined Digital in 1986 after receiving a B.S. degree in electrical engineering, computer engineering, and mathematics from Carnegie-Mellon University. At Carnegie-Mellon, he was elected to Eta Kappa Nu and was awarded the David Tuma Undergraduate Laboratory Project Award. A senior hardware engineer, Hugh designed the TCDS ASIC and SCSI subsystem for the DEC 3000 AXP Model 500. Prior to this work, he designed floating-point hardware for two projects in the Advanced VAX Development Group.

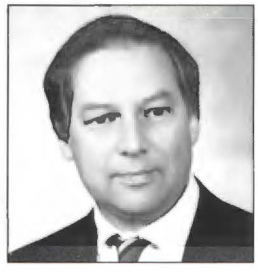
**Maureen Ladd** Maureen Ladd received a B.S. degree in computer engineering from the University of Illinois in 1986. She then joined the Semiconductor Engineering Group within Digital and worked on a 32-bit RISC microprocessor. Maureen received an M.S.E. degree in electrical engineering from the University of Michigan in 1990 through Digital's Graduate Engineering Education Program. Upon her return to Digital, she worked on the implementation of the first microprocessor based on the Alpha AXP architecture.



**Burton M. Leary** Mike Leary is currently a consulting engineer in the Semiconductor Engineering Group/Advanced Development Memory Group. He designed the instruction and data caches for the DECchip 21064 CPU and is currently working on the design of advanced memory products. Mike joined Digital in 1980 after receiving a B.S.E.E. degree from the University of Massachusetts.

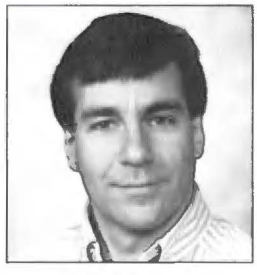


**Liam Madden** Liam Madden joined Digital in 1984 and has since designed both CISC and RISC microprocessors and contributed in the area of CMOS process development. He is currently a consultant engineer in Digital's CPU Advanced Development Group and his interests include circuit design and CMOS technology development. Prior to joining Digital, Liam designed industrial microcontrollers for Mahon and McPhillips, Ireland, and worked for Harris Semiconductor. He received a B.S. degree from University College Dublin in 1979 and an M.E. degree from Cornell University in 1990.

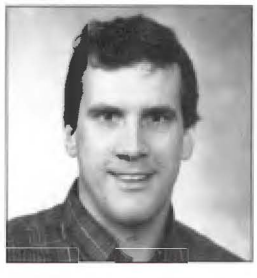


**Maurice P. Marks** Maurice Marks is a senior engineering manager in the Semiconductor Engineering Advanced Development Group. He currently manages the AXP Migration Tools Group and contributed to the design and implementation of the translators. In Maurice's twenty years with Digital, he has led compiler, operating system, hardware and software tools, CAD, system, and chip projects. He holds B.Sc. and B.E. degrees from the University of New South Wales and has published papers on transaction processing, software portability, and CAD technology. Maurice is a member of the Australian Computer Society.

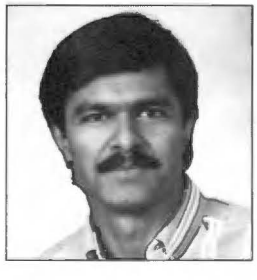
**Barry A. Maskas** Barry Maskas is the project leader responsible for architecture, semiconductor technology, and development of the DEC 4000 AXP system buses, processors, and memories. He is a consulting engineer with the Entry Systems Business Group. In previous work, he was responsible for the architecture and development of custom VLSI peripheral chips for VAX 4000 and MicroVAX systems. Prior to that work, he was a codesigner of the MicroVAX II CPU and memory modules. He joined Digital in 1979, after receiving a B.S.E.E. from Pennsylvania State University. He holds three patents and has eleven patent applications.



**Edward J. McLellan** Ed McLellan is a principal engineer in the Semiconductor Engineering Group. He has contributed to the design of several processor chips. Ed joined Digital in 1980 after receiving a B.S. degree in computer and systems engineering from Rensselaer Polytechnic Institute, where he was elected to Eta Kappa Nu. He holds three patents in computer design and has one application pending.

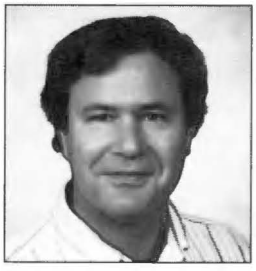


**Derrick R. Meyer** Dirk Meyer joined Digital's Semiconductor Engineering Group in 1986. He was initially involved in the design of the cache and memory systems for a chilled CMOS VAX processor. He has since been involved in the development of microprocessors based on the Alpha AXP architecture. Prior to joining Digital, he was employed at Intel Corporation, where he was involved in the design of various CMOS microcontrollers, including the 80C51 and 80C196. Dirk received a B.S. degree in computer engineering from the University of Illinois in 1983.

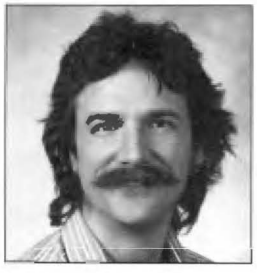


**Zia Mohamed** Zia Mohamed has been a member of the Database Systems Group since joining Digital in 1989. He works in the area of query optimization for the DEC Rdb for OpenVMS products; his contributions involve cost-based optimization of database queries and algorithms for execution of optimized query plans. He has developed dynamic OR optimization techniques, refinement of cost-model, and algorithms for better access plans for views. Zia holds a B.S. degree in electrical engineering from Bangalore University, India, and an M.S. degree in computer science from Texas Tech University.

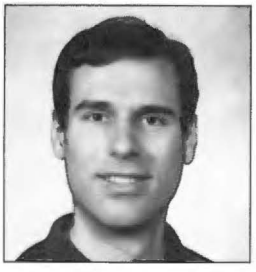
**James Montanaro** James Montanaro received B.S.E.E. and M.S.E.E. degrees from the Massachusetts Institute of Technology in 1980. He joined Digital Equipment Corporation in 1982. He was a circuit designer on the floating-point chip for the LSI 11/74 and a MicroVAX peripheral chip. He led the physical implementation of the uPRISM CPU, a 70-MHz prototype RISC CPU completed in 1988. James also led the physical implementation of the first CPU chip based on the Alpha AXP architecture and then contributed as a circuit designer for the DECchip 21064 CPU. He is currently with Apple Computer, Inc.



**Stephen J. Morris** Stephen Morris is a consultant software engineer in the Semiconductor Engineering Advanced Development Group. In addition to writing the Alpha ISP simulator, he wrote the OpenVMS and OSF PALcode for the Alpha AXP program. In previous work, Stephen designed the control sections of the instruction prefetch and translation look-aside buffer for an experimental Digital RISC chip. He also worked on the MicroVAX chip team, doing console and debug work, and in the RSTS/E operating system group. Stephen joined Digital after receiving a B.A. in biology from the University of Rochester in 1977.



**William B. Noyce** Senior consultant software engineer William Noyce is a member of the Software Development Technologies Group. He has developed several GEM compiler optimizations, including those that eliminate branches. In prior positions at Digital, Bill implemented support for new disks and processors on the RSTS/E project, led the development of VAX DBMS V1 and VAX Rdb/VMS V1, and designed and implemented automatic parallel processing for VAX FORTRAN/HPO. Bill received a B.A. (1976) in mathematics from Dartmouth College, where he implemented enhancements to the time-sharing system.

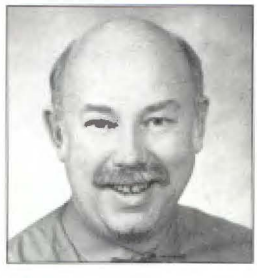


**Donald A. Priore** After receiving an S.M. degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Donald Priore joined Digital in 1984. Initially, he worked on device characterization, yield enhancement, and yield modeling of NMOS and CMOS processes in manufacturing. Subsequently, he joined a CMOS design group, working first with low-temperature CMOS technology and later with conventional CMOS in high-performance microprocessor design. His interests include signal, clock, and power integrity in the on-chip environment.

**Vidya Rajagopalan** Vidya Rajagopalan received a B.E. degree in electronics engineering from Visvesvaraya Regional College of Engineering, Nagpur, India, in 1986, and an M.S. degree in electrical engineering from the University of Maryland in 1989. She was with Norsk Data India Ltd. from 1986 to 1987 as a systems design engineer. In 1989 she joined Digital's Semiconductor Engineering Group and was a member of the design team of the DECchip 21064 RISC microprocessor. Vidya is currently involved in the design of high-performance microprocessors.

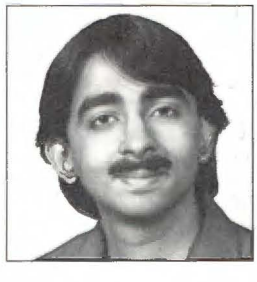
**James J. Reisert** A senior hardware engineer, Jim Reisert designed the TC ASIC for the DEC 3000 AXP Model 500. Prior to this project work, he designed instruction parsers/decoders for two VAX implementations. Jim holds a patent for his design of a method for replaying instructions after a microtrap. Before joining Digital in 1986, he received an S.B. in electrical engineering from the Massachusetts Institute of Technology. He is currently in charge of timing verification for another AXP workstation.

**Pamela J. Rickard** Principal software engineer Pam Rickard is a member of the team porting DECnet/OSI for OpenVMS to the Alpha AXP platform. As the initial member of the DECnet for OpenVMS AXP porting team, Pam took responsibility for creating an effective team, ported NETDRIVER and other MACRO-32 code, and debugged major portions of the ported product. Since joining Digital in 1978, she has contributed to PATHWORKS for OS/2 and led the console, microcode, and system test activities of the VAX-11/785 project. Pam received a B.S. (1970) in mathematics and computer science from the University of Denver.

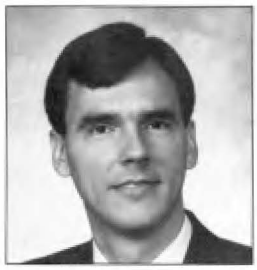


**Scott G. Robinson** Scott Robinson is a software engineering manager in the AXP Migration Tools Group. He contributed to the design and implementation of the binary translators, particularly the VAX translated image environment. Scott has also developed implementations of DECnet and CAD/CAM systems to design VAX processors. Prior to joining Digital in 1978, Scott worked on a variety of Digital hardware and software implementations. He holds a B.S. in electrical engineering from the University of Arizona and is a member of IEEE.

**Sridhar Samudrala** Sridhar Samudrala is a consulting hardware engineer in the Semiconductor Engineering Group, where he is currently working on a new CPU chip. He joined Digital in 1977. Since that time, he has worked on the design and verification of PDP-11/23 chips, VAX 8200 microcode development, and on the architecture and design of floating-point chips. He holds two patents and has three patent applications pending, all on floating-point design. Sridhar received an M.Sc. (Tech) degree from Andhra University, India, and an M.S.E.E. degree from the University of Wisconsin.

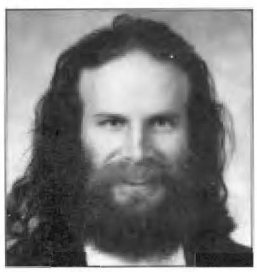


**Sribalan Santhanam** Sri Santhanam received a B.E. degree in electrical engineering from Anna University, Madras, India, in 1987, and an M.S.E. degree in computer science and engineering from the University of Michigan in 1989. Upon graduation, he joined Digital as a design engineer for the Semiconductor Engineering Group, responsible for the full-custom design and development of high-performance CMOS VLSI processors. Sri worked on the design of the floating-point unit of the DECchip 21064 CPU. He is currently involved in the design of another high-performance microprocessor.



**Stephen F. Shirron** Stephen Shirron is a consulting software engineer in the Entry Systems Business Group and is responsible for OpenVMS support of new systems. He contributed to many areas of the DEC 4000, including PALcode, console, and OpenVMS support. Stephen joined Digital in 1981 after completing B.S. and M.S. degrees (summa cum laude) at Catholic University. In previous work, he developed an interpreter for VAX/Smalltalk-80 and wrote the firmware for the RQDX3 disk controller. Stephen has two patent applications and has written a chapter in *Smalltalk-80: Bits of History, Words of Advice.*

**Richard L. Sites** Dick Sites is a senior consultant engineer in the Semiconductor Engineering Group, where he is working on binary translators and the Alpha AXP architecture. He joined Digital in 1980 and has contributed to various VAX implementations. Previously, he was employed by IBM, Hewlett-Packard, and Burroughs, and taught at the University of California. Dick received a B.S. in mathematics from MIT and a Ph.D. in computer science from Stanford University. He also studied computer architecture at the University of North Carolina. He holds a number of patents on computer hardware and software.

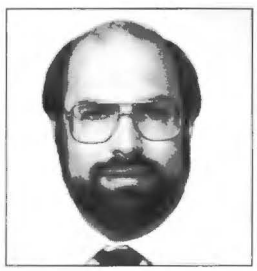


**Peter M. Spiro** Peter Spiro, a consulting software engineer, is presently the technical director for the Rdb and DBMS software products. Peter's current focus is database performance for Alpha AXP systems and very large database issues. Peter joined Digital in 1985, after receiving M.S. degrees in forest science and computer science from the University of Wisconsin–Madison. He has four patents related to database journaling and recovery, and he has authored two papers for earlier issues of the *Digital Technical Journal.*

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

**Robin L. Stewart** Robin Stewart joined Digital in 1986 after receiving a B.S. in electrical engineering from the University of Vermont. She is in the process of obtaining an M.B.A. degree from Boston College. A senior technology (hardware) engineer, Robin had responsibility for the integrated circuit technology in the DEC 3000 AXP Model 500 workstation. Prior to this project work, she was a component engineer in Digital's Semiconductor Business Organization.

**Charles P. Thacker** Chuck Thacker has been with Digital's Systems Research Center since 1983. Before joining Digital, he was a senior research fellow at the Xerox Palo Alto Research Center. His research interests include computer architecture, computer networking, and computer-aided design. He holds several patents in the area of computer organization and is coinventor of the Ethernet local area network. In 1984, Chuck was the recipient (with B. Lampson and R. Taylor) of the ACM Software System Award. He received an A.B. degree in physics from the University of California in 1967. He is a member of ACM and IEEE.



**Benjamin J. Thomas III** Benjamin Thomas joined the OpenVMS AXP project in 1989 as project leader for I/O subsystem design and porting. In this role, he has also contributed to the I/O architecture of current and future AXP systems. Ben joined Digital in 1982 and has worked in the VMS group since 1984. In prior work, he was the director of software engineering at a microcomputer firm. Ben is a consulting engineer and has a B.S. (1978) in physics from the University of New Hampshire and an M.S.C.S. (1990) from Worcester Polytechnic Institute.

**Catharine van Ingen** A consulting software engineer, Catharine van Ingen was co-system architect for the VAX and DEC 7000 products. Catharine is currently on leave from Digital and is working on engineering document management in large heterogeneous systems. Before joining Digital in 1987, she worked on data acquisition systems for two large physics detectors at the Fermi National Accelerator Laboratory and Stanford Linear Accelerator Center. She holds several degrees in civil engineering, including a B.S. and an M.S. from the University of California and a Ph.D. from the California Institute of Technology.

**Nicholas A. Warchol** Nick Warchol, a consulting engineer in the Entry Systems Business Group, is the project leader responsible for I/O architecture and I/O module development for the DEC 4000 AXP systems. In previous work, he contributed to the development of VAX 4000 systems. He was also a designer of the MicroVAX 3300 and 3400 processor modules and the RQDX3 disk controller. Nick joined Digital in 1977 after receiving a B.S.E.E. (cum laude) from the New Jersey Institute of Technology. In 1984 he received an M.S.E.E. from Worcester Polytechnic Institute. He has four patent applications.

**Richard T. Witek** Rich Witek joined Digital in 1977 to work on DECnet network architecture during Phase II. In 1982 he joined Digital's Semiconductor Engineering Group where he worked on CAD development, MicroVAX VLSI chips, and a variety of internal RISC projects. Rich was a codesigner of the Alpha AXP architecture and the principal microarchitect of the DECchip 21064 CPU chip. He received a B.A. degree in computer science from Aurora College. Rich is currently employed by Apple Computer, Inc.

# Foreword

**Robert M. Supnik**
*Corporate Consultant,*
*Vice President*
*Technical Director,*
*Engineering*

It all started with eight people in a conference room.*

The time was the summer of 1988. Digital Equipment Corporation had just closed the best fiscal year in its history, with record revenues and profits. Digital's VAX systems were the most widely used timesharing systems in the industry and were the "blue-ribbon standard" for mid-range computing. Digital was the second-largest workstation vendor. The company had just introduced the VAX 6000 system, its first expandable multiprocessor, was developing a true VAX mainframe, and had decided on a rapid thrust into RISC workstations to capitalize on that growing market. What could possibly go wrong?

Nonetheless, senior managers and engineers saw trouble ahead. Workstations had displaced VAX VMS from its original technical market. Networks of personal computers were replacing timesharing. Application investment was moving to standard, high-volume computers. Microprocessors had surpassed the performance of traditional mid-range computers and were closing in on mainframes. And advances in RISC technology threatened to aggravate all of these trends. Accordingly, the Executive Committee asked Engineering to develop a long-term strategy for keeping Digital's systems competitive. Engineering convened a task force to study the problem.

The task force looked at a wide range of potential solutions, from the application of advanced pipelining techniques in VAX systems to the deployment of a new architecture. A basic constraint was that the proposed solution had to provide strong compatibility with current products. After several months of study, the team concluded that only a new RISC architecture could meet the stated objective of long-term competitiveness, and that only the existing VMS and UNIX environments could meet the stated constraint of strong compatibility. Thus, the challenge posed by the task force was to design the most competitive RISC systems that would run the current software environments.

Key groups in Engineering responded to this challenge. A cross-functional team from hardware and software defined the basic architecture. Advanced development teams began work on the knotty engineering problems: in the semiconductor group, the specification and design of a fast microprocessor, and the automatic translation of executable binary images; in the operating systems groups, on the porting of ULTRIX and of VMS (which was not portable!); and in the compiler group, on superscalar code generation. In the fall of 1989, Alpha became an officially sanctioned advanced development program.† In the summer of 1990, it transitioned to product development.

From the original core in semiconductors, operating systems, and compilers, work expanded throughout Engineering. The server and workstation hardware groups specified and started designing a family of systems, from desktop to data center. The networks group began porting DECnet, TCP/IP, X.25, LAT, and the many other networking products. The layered software group inventoried the existing portfolio of products and prioritized the order and importance of delivery. The research group pitched in by designing an experimental multiprocessor as a software development testbed.

In parallel with the engineering work, marketing, sales, and service teams worked closely with business partners and customers to shape the deliverables and messages to meet external requirements. These teams briefed key customers and partners early in the development process and

*The Corona Borealis conference room in the LTN1 facility in Littleton, Mass. LTN1 was chosen because it was the geographic epicenter of the arc of Digital engineering facilities on Massachusetts Route 495, the Corona Borealis because it was the only conference room with windows.

†After going through more than one name change. The original study team was called the "RISCy VAX Task Force." The advanced development work was labeled "EVAX." When the program was approved, the Executive Committee demanded a neutral code name, hence "Alpha."

incorporated their advice into the development program. Ongoing partner and customer advisory boards provided regular feedback on all aspects of the program and helped shape two critical extensions of the original concept: the open licensing of Alpha technology, and the porting of Windows NT.

Taken together, the scope of the Engineering effort, the need for Marketing, Field, and Service involvement, and the high degree of customer and business partner participation, posed unique management challenges. Rather than organize a large-scale hierarchical project, the company chose to manage Alpha as a distributed program. A small program team used enrollment management practices and strict operational discipline to coordinate and inspect activities across the company. This networked approach to management gave the program both flexibility and resiliency in the face of rapidly changing business and organizational conditions.

The work of Engineering, Manufacturing, Marketing, Sales, and Service came together in November 1992 with the announcement of the Alpha AXP systems family: seven systems, three operating systems, six languages, multiple networks, migration tools, open licensing of technology, hardware and software partnerships, and more than 2000 committed applications. Today, Alpha AXP embodies a fundamental repositioning of Digital Equipment Corporation to be the technology and solutions leader in twenty-first century computing: a company dedicated to meeting customers' needs with the best computing, business, and service technology available. The delivery of Alpha AXP required the largest engineering program in Digital's history, spanning more than twenty Engineering groups worldwide. This issue of the *Digital Technical Journal* documents just a few of the hundreds of projects involved in bringing Alpha to fruition; future issues will continue the story.

**Richard L. Sites** |

# Alpha AXP Architecture

*The Alpha AXP 64-bit computer architecture is designed for high performance and longevity. Because of the focus on multiple instruction issue, the architecture does not contain facilities such as branch delay slots, byte writes, and precise arithmetic exceptions. Because of the focus on multiple processors, the architecture does contain a careful shared-memory model, atomic-update primitive instructions, and relaxed read/write ordering. The first implementation of the Alpha AXP architecture is the world's fastest single-chip microprocessor. The DECchip 21064 runs multiple operating systems and runs native-compiled programs that were translated from the VAX and MIPS architectures.*

> Thus in all these cases the Romans did what all wise princes ought to do; namely, not only to look to all present troubles, but also to those in the future, against which they provided with the utmost prudence.
>
> —Niccolo Machiavelli, *The Prince*

## Historical Context

The Alpha AXP architecture grew out of a small task force chartered in 1988 to explore ways to preserve the VAX VMS customer base through the 1990s. This group eventually came to the conclusion that a new reduced instruction set computer (RISC) architecture would be needed before the turn of the century, primarily because 32-bit architectures will run out of address bits. Once we made the decision to pursue a new architecture, we shaped it to do much more than just preserve the VMS customer base.

This paper discusses the architecture from a number of points of view. It begins by making the distinction between architecture and implementation. The paper then states the overriding architectural goals and discusses a number of key architectural decisions that were derived directly from these goals. The key decisions distinguish the Alpha AXP architecture from other architectures. The remaining sections of the paper discuss the architecture in more detail, from data and instruction formats through the detailed instruction set. The paper concludes with a discussion of the designed-in future growth of the architecture. An Appendix explains some of the key technical terms used in this paper. These terms are highlighted with an asterisk in the text.

## Architecture Distinct from Implementations

From the beginning of the Alpha AXP design, we distinguished the architecture from the implementations, following the distinction made by the IBM System/360 architects:

> Computer architecture is defined as the attributes and behavior of a computer as seen by a machine-language programmer. This definition includes the instruction set, instruction formats, operation codes, addressing modes, and all registers and memory locations that may be directly manipulated by a machine-language programmer.
>
> Implementation is defined as the actual hardware structure, logic design, and data-path organization of a particular embodiment of the architecture.[1]

Thus, the architecture is a document that describes the behavior of all possible implementations; an implementation is typically a single computer chip.[2] The architecture and software written to the architecture are intended to last several decades, while individual implementations will have much shorter lifetimes. The architecture must therefore carefully describe the behavior that a machine-language programmer sees, but must not describe the means by which a particular implementation achieves that behavior.

A similar approach has been used with much success in specifying the PDP-11 and VAX families of computers. An alternate approach is to design and build a fast RISC* chip, then wait to see if it is successful in the marketplace. If so, successive implementations are often forced to reproduce accidents of the initial design, or to introduce slight software incompatibilities. This approach works, but with varying success.

## Architectural Goals

When we started the detailed design of the Alpha AXP architecture, we had a short list of goals:

1. High performance
2. Longevity
3. Capability to run both VMS and UNIX operating systems
4. Easy migration from VAX and MIPS architectures

These goals directly influenced our key decisions in designing the architecture.

In considering performance and longevity, we set a 15- to 25-year design horizon and tried to avoid any design elements that we thought could become limitations during this time. In current architectures, a primary limitation is the 32-bit memory address. Thus we adopted a full 64-bit architecture, with a minimal number of 32-bit operations for backward compatibility.

We also considered how implementation performance should scale over 25 years. During the past 25 years, computers have become about 1,000 times faster. Therefore we focused our design decisions on allowing Alpha AXP system implementations to become 1,000 times faster over the coming 25 years. In our projections of future performance, we reasoned that raw clock rates would improve by a factor of 10 over that time, and that other design dimensions would have to provide two more factors of 10.

If the clock cannot be made faster, then more work must be done per clock tick. We therefore designed the Alpha AXP architecture to encourage multiple instruction issue* implementations that will eventually sustain about ten new instructions starting every clock cycle. This aggressive technique of starting multiple instructions distinguishes the Alpha AXP architecture from many other RISC architectures.

The remaining factor of 10 will come from multiple processors. A single system will contain perhaps ten processors and share memory. We therefore designed a multiprocessor memory model and matching instructions from the beginning. This early accommodation for multiple processors also distinguishes the Alpha AXP architecture from many other RISC architectures, which try to add the proper primitives later.

To run the OpenVMS AXP and the DEC OSF/1 AXP—and now the Microsoft Windows NT—operating systems, we adopted an idea from a previous Digital RISC design called PRISM.[3] We placed the underpinnings for interrupt delivery and return, exceptions, context switching, memory management, and error handling in a set of privileged software subroutines called PALcode. These subroutines have controlled entry points, run with interrupts turned off, and have access to real hardware (implementation) registers. By including different sets of PALcode for different operating systems, neither the hardware nor the operating system is burdened with a bad interface match, and the architecture itself is not biased toward a particular computing style.

To run existing VAX and MIPS binary images, we adopted the idea of binary translation,* as described in a companion paper.[4,5,6] The combination of PALcode and binary translation gave us the luxury of designing a new architecture. Other than the fundamental integer and floating-point data types, there are no specific VAX or MIPS features carried directly into the Alpha AXP instruction-set architecture for compatibility reasons.

## Key Design Decisions

This section presents the design decisions that distinguish the Alpha AXP architecture from others.

### RISC

The Alpha AXP architecture is a traditional RISC load/store architecture. All data is moved between registers and memory without computation, and all computation is done between values in registers. Little-endian byte addressing and both VAX and IEEE floating-point operations* are carried over from the VAX and MIPS architectures.[7] We assumed that most implementations would pipeline instructions, i.e., they would start execution of a second, third, etc. instruction before the execution of a first instruction completes. We assumed that the implementation latency of many operations would be important. Latency is the number of cycles a program must wait to use the result of a preceding instruction. We assumed that the vast majority of memory operands would be aligned. An aligned operand of size 2**$N$ bytes* has an address with $N$ low-order zeros. Other memory operands are termed unaligned.

### Full 64-bit Design

The Alpha AXP architecture uses a linear* 64-bit virtual address space. Registers, addresses, integers, floating-point numbers, and character strings are

all operated on as full 64-bit quantities. There are no segmented addresses.*

### Register File

In choosing the register file design, we considered both a single combined register file and split integer and floating-point register files. We chose a split register file to support aggressive multiple issue. A combined file is somewhat more flexible, especially for programs that are heavily skewed toward integer-only or floating-point-only computation. A combined file also makes it easier to pass a mixture of integer and floating-point subroutine parameters in registers. However, split files allow graceful two-chip implementations and smaller integer-only implementations. They also need fewer read/write ports per file to sustain a given amount of multiple instruction issue.

We also considered whether each file should contain 32 or 64 registers. We chose 32, largely because

1. Thirty-two registers in each file are enough to support at least eight-way multiple issue.
2. Two valuable instruction bits are better used to make a 16-bit displacement field in memory-format instructions.

More registers might seem better, but excess registers consume chip area and access time, save/restore speed across subroutines and context switches, and instruction bits that might be put to better use. Compilers can deliver substantial performance gains when given 32 registers instead of 16, but there is no clear evidence of similar gains with 64 registers. Demand for registers is likely to increase slowly in the future, but a number of implementation techniques, such as short latency pipelines and register renaming, should satisfy this demand.

### Multiple Instruction Issue

Our design sought to eliminate any mechanism that would hinder aggressive multiple instruction issue implementations. Therefore we tried to avoid all special or hidden processor resources.[8] Thus, the Alpha AXP architecture has no condition codes, no global exception enables, no multiplier-quotient or string registers, no branch delay slots, no suppressed instructions or skips, no precise arithmetic exceptions, and no single-byte writes to memory. All of these features, found in some RISC architectures, have the effect of hindering multiple instruction issue, or hindering pipelining of multiple instances of the same instruction. For example, a dedicated string register makes it hard to have three unrelated string operations in the pipeline at once.

To illustrate the performance loss associated with special or hidden processor resources, consider a dual-issue implementation with a four-cycle-deep pipeline. At the beginning of each cycle, up to six prior instructions are partially executed and two more are about to be issued. Six prior instructions can have six pending writes to result registers, plus six sets of side effects on special or hidden processor resources. The next two instructions can specify a total of four operand registers, two more result registers, and two more sets of side effects on special or hidden resources. The decision to issue 0, 1, or 2 of the next instructions involves 36 simple comparisons of pairs of register numbers and 12 complex comparisons of sets of side effects. The number of such comparisons increases as a function of the issue width, the pipeline depth, and the number of special or hidden processor resources. The complexity of these comparisons can limit the clock rate. The register-number comparisons are unavoidable, therefore we tried to limit special or hidden processor resources.

*Branch Delay Slots* The Alpha AXP architecture has no branch delay slots. The branch delay slots found in some RISC architectures require exactly one following instruction to be executed after a conditional branch. In 1988 this was, perhaps, a good idea for overlapping branch latency in a single-issue chip with a one-cycle instruction cache. In 1995, however, it will not scale well to a four-way issue chip with a two-cycle instruction cache. Instead of one instruction, up to eight instructions would be needed in the delay slot. Branch delay slots also introduce a restart problem if the instruction in the delay slot faults: one restart program counter is needed for the delay slot and another one for the actual branch target.

*Suppressed Instructions* The Alpha AXP architecture has no suppressed instructions, whereby the execution of one instruction conditionally suppresses a following one. Suppressed (or skipped) instructions are found in other RISC architectures. The suppression bit(s) represent nonreplicated hidden state, so multiple instruction issue is difficult for more than one potential suppressor. If an interrupt is taken between a suppressor and suppressee, or if the suppressee takes a restartable exception (e.g., page fault), the correct version of

the suppression state must be saved and restored. There are also definitional problems with this approach: Are exceptions ever reported for suppressed instructions? What happens if the suppressed instruction suppresses a third instruction?

*Byte Load or Store Instructions* The Alpha AXP architecture has no byte load or store instructions and no implicit unaligned accesses. There also are no partial-register writes. The byte load/store instructions and unaligned accesses found in some RISC architectures can be a performance bottleneck. They require an extra byte shifter in the speed-critical load and store paths, and they force a hard choice in fast cache design. The partial-register writes found in other RISC architectures can also be a performance bottleneck because they require masking and shifting in the fundamental operation of accessing a register.

On a previous project involving a MIPS implementation, we found the shifter for the load-left/load-right instructions to be a direct cycle-time bottleneck. Also, the VAX 8700 implementation (circa 1986) removed the byte shifter in the load/store hardware in favor of a faster microcycle, with 2 cycles for a byte load and 6 cycles for an unaligned 32-bit access. This decision achieved a net performance gain. Our experience encouraged us to avoid byte load/store.

An additional problem with byte stores is that an implementer may easily choose only two of the three design features: fast write-back cache, single-bit error correction code (ECC), or byte stores.

Byte stores are straightforward in simple byte-parity write-through cache implementations. Except for the expensive design of four or five ECC bits for every eight bits of data, a byte store to a fast ECC write-back cache involves

1. Reading an entire cache word*
2. Checking the ECC bits and correcting any single-bit error
3. Modifying the byte
4. Calculating the new ECC bits
5. Writing the entire cache word

This read-modify-write sequence requires hidden sequencer hardware and hidden state to hold the cache word temporarily. The sequencer tends to slow down ordinary full-cache-word stores. The need for byte stores tends to ripple throughout the memory subsystem design, making each piece a little more complicated and a little slower. With nonreplicated hidden state, it is difficult to issue another byte store until the first one finishes. Finally, the existence of a byte store instruction has led to programs and library routines for other RISC implementations with single-byte move and compare loops. String manipulation on Alpha AXP implementations is up to eight times faster by processing eight bytes at a time.[9]

Instead of including byte load/store, we followed the RISC philosophy of exposing hidden computation as a sequence of many simple, fast instructions. In the Alpha AXP architecture, a byte load is done as an explicit load/shift sequence; a byte store as an explicit load/modify/store sequence. We tuned the instruction set to keep these sequences short. The instructions in these sequences can be intermixed, scheduled, and issued as multiples with other computation, as can the rest of the instructions in the architecture. Table 1 gives a summary of the Alpha AXP instruction set.

*Arithmetic Exceptions* The Alpha AXP architecture has no precise arithmetic exceptions. Reporting an arithmetic exception (e.g., overflow, underflow) precisely means that instructions subsequent to the one causing the exception must not be executed. This is straightforward in a slow implementation that runs a single instruction to completion before starting the next one, but becomes substantially more difficult to do quickly in a pipelined four-way issue implementation. There are standard techniques available for delivering precise exceptions while running quickly (checking exponents, suppressing register writes, exception silos and backout), but these techniques consume substantial design time and can cost some performance. They appear not to scale well with wider multiple issue or faster clocks.

Exceptional cases are just that—exceptional, or rare, events. Based partly on customer requests, we decided to emphasize the performance of normal operations at the expense of exceptional cases. Rather than an implicit exception ordering between every pair of instructions, we adopted the Cray-1 model of arithmetic exceptions—in which exceptions are reported eventually—plus an explicit trap barrier (TRAPB) instruction that can be used to make exception reporting as precise as desired.[10] We also documented a code-generation design that needs one trap barrier per branch (at most) to give precise reporting. Using TRAPB

**Table 1 Alpha AXP Architecture Instruction Set Summary**

| Load/Store, Byte Manipulation | |
|---|---|
| LDA | Load address |
| LDAH | Load address high |
| LDL | Load sign-extended longword |
| LDQ | Load quadword |
| LDQ_U | Load unaligned quadword |
| LDL_L | Load sign-extended longword, locked |
| LDQ_L | Load quadword locked |
| STL_C | Store longword, conditional |
| STQ_C | Store quadword, conditional |
| STL | Store longword |
| STQ | Store quadword |
| STQ_U | Store unaligned quadword |
| EXTBL | Extract byte low |
| EXTWL | Extract word low |
| EXTLL | Extract longword low |
| EXTQL | Extract quadword low |
| EXTWH | Extract word high |
| EXTLH | Extract longword high |
| EXTQH | Extract quadword high |
| INSBL | Insert byte low |
| INSWL | Insert word low |
| INSLL | Insert longword low |
| INSQL | Insert quadword low |
| INSWH | Insert word high |
| INSLH | Insert longword high |
| INSQH | Insert quadword high |
| MSKBL | Mask byte low |
| MSKWL | Mask word low |
| MSKLL | Mask longword low |
| MSKQL | Mask quadword low |
| MSKWH | Mask word high |
| MSKLH | Mask longword high |
| MSKQH | Mask quadword high |

| Floating Point Load/Store | |
|---|---|
| LDF | Load F format (VAX single) |
| LDG | Load G format (VAX double) |
| LDS | Load S format (IEEE single) |
| LDT | Load T format (IEEE double) |
| STF | Store F format (VAX single) |
| STG | Store G format (VAX double) |
| STS | Store S format (IEEE single) |
| STT | Store T format (IEEE double) |

| Address/Constant | |
|---|---|
| LDA | Load address |
| LDAH | Load address high |

| Integer Computation and Conditional Move | |
|---|---|
| ADDL | Add longword |
| S4ADDL | Add longword, scale by 4 |
| S8ADDL | Add longword, scale by 8 |
| ADDQ | Add quadword |
| S4ADDQ | Add quadword, scale by 4 |
| S8ADDQ | Add quadword, scale by 8 |
| CMPEQ | Compare signed quadword = |
| CMPLT | Compare signed quadword < |
| CMPLE | Compare signed quadword ≤ |
| CMPULT | Compare unsigned quadword < |
| CMPULE | Compare unsigned quadword ≤ |
| MULL | Multiply longword |
| MULQ | Multiply quadword |
| UMULH | Multiply quadword high, unsigned |
| SUBL | Subtract longword |
| S4SUBL | Subtract longword, scale by 4 |
| S8SUBL | Subtract longword, scale by 8 |
| SUBQ | Subtract quadword |
| S4SUBQ | Subtract quadword, scale by 4 |
| S8SUBQ | Subtract quadword, scale by 8 |
| AND | AND logical |
| BIS | OR logical |
| XOR | XOR logical |
| BIC | AND-NOT logical |
| ORNOT | OR-NOT logical |
| EQV | XOR-NOT logical |
| SLL | Shift left, logical |
| SRL | Shift right, logical |
| SRA | Shift right, arithmetic |
| CMOVEQ | Conditional move if reg = 0 |
| CMOVNE | Conditional move if reg ≠ 0 |
| CMOVLT | Conditional move if reg < 0 |
| CMOVLE | Conditional move if reg ≤ 0 |
| CMOVGT | Conditional move if reg > 0 |
| CMOVGE | Conditional move if reg ≥ 0 |
| CMOVLBC | Conditional move if reg low bit clear |
| CMOVLBS | Conditional move if reg low bit set |
| CMPBGE | Compare bytes, unsigned |
| ZAP | Clear selected bytes |
| ZAPNOT | Clear unselected bytes |

| Integer Branch | |
|---|---|
| BEQ | Branch if reg = 0 |
| BNE | Branch if reg ≠ 0 |
| BLT | Branch if reg < 0 |
| BLE | Branch if reg ≤ 0 |
| BGT | Branch if reg > 0 |
| BGE | Branch if reg ≥ 0 |
| BLBC | Branch if low bit clear |
| BLBS | Branch if low bit set |
| BR | Branch |
| BSR | Branch to subroutine |
| JMP | Jump |
| JSR | Jump to subroutine |
| RET | Return from subroutine |
| JSR_COROUTINE | Jump to subroutine, return |

| Floating Point Branch | |
|---|---|
| FBEQ | FP branch if = 0 |
| FBNE | FP branch if ≠ 0 |
| FBLT | FP branch if < 0 |
| FBLE | FP branch if ≤ 0 |
| FBGT | FP branch if > 0 |
| FBGE | FP branch if ≥ 0 |

continued on next page

**Table 1 Alpha AXP Architecture Instruction Set Summary (continued)**

| Floating Point Computation and Conditional Move | |
|---|---|
| CPYS | Copy sign |
| CPYSN | Copy sign, negate |
| CPYSE | Copy sign and exponent |
| CVTQL | Convert quadword to longword |
| CVTLQ | Convert longword to quadword |
| FCMOVEQ | FP conditional move if reg = 0 |
| FCMOVNE | FP conditional move if reg ≠ 0 |
| FCMOVLT | FP conditional move if reg < 0 |
| FCMOVLE | FP conditional move if reg ≤ 0 |
| FCMOVGT | FP conditional move if reg > 0 |
| FCMOVGE | FP conditional move if reg ≥ 0 |
| MF_FPCR | Move from FP control register |
| MT_FPCR | Move to FP control register |
| ADDF | Add F format (VAX single) |
| ADDG | Add G format (VAX double) |
| ADDS | Add S format (IEEE single) |
| ADDT | Add T format (IEEE double) |
| CMPGEQ | Compare G format = (VAX double) |
| CMPGLT | Compare G format < (VAX double) |
| CMPGLE | Compare G format ≤ (VAX double) |
| CMPTEQ | Compare T format = (IEEE double) |
| CMPTLT | Compare T format < (IEEE double) |
| CMPTLE | Compare T format ≤ (IEEE double) |
| CMPTUN | Compare T format unordered (IEEE double) |
| CVTGQ | Convert G format to quadword (VAX double) |
| CVTQF | Convert quadword to F format (VAX single) |
| CVTQG | Convert quadword to G format (VAX double) |
| CVTDG | Convert D to G format (VAX double/double) |
| CVTGD | Convert G to D format (VAX double/double) |
| CVTGF | Convert G to F format (VAX double/single) |
| CVTTQ | Convert T format to quadword (IEEE double) |
| CVTQS | Convert quadword to S format (IEEE single) |
| CVTQT | Convert quadword to T format (IEEE double) |
| CVTTS | Convert T to S format (IEEE double/single) |
| CVTST | Convert S to T format (IEEE single/double) |
| DIVF | Divide F format (VAX single) |
| DIVG | Divide G format (VAX double) |
| DIVS | Divide S format (IEEE single) |
| DIVT | Divide T format (IEEE double) |
| MULF | Multiply F format (VAX single) |
| MULG | Multiply G format (VAX double) |
| MULS | Multiply S format (IEEE single) |
| MULT | Multiply T format (IEEE double) |
| SUBF | Subtract F format (VAX single) |
| SUBG | Subtract G format (VAX double) |
| SUBS | Subtract S format (IEEE single) |
| SUBT | Subtract T format (IEEE double) |
| **System** | |
| CALL_PAL | Call privileged architecture library |
| TRAPB | Trap barrier (precise exception) |
| FETCH | Prefetch (cache) date hint |
| FETCH_M | Prefetch (cache) data, modify hint |
| MB | Memory barrier (serialize) |
| WMB | Memory barrier (serialize) write |
| RPCC | Read process cycle counter |
| RC | Read and clear |
| RS | Read and set |
| PALRES0 | PALcode reserved opcode 0 |
| PALRES1 | PALcode reserved opcode 1 |
| PALRES2 | PALcode reserved opcode 2 |
| PALRES3 | PALcode reserved opcode 3 |
| PALRES4 | PALcode reserved opcode 4 |

instructions in the first Alpha AXP implementation lowers performance 3 percent to 25 percent in real floating-point programs and less than 1 percent in integer programs, but improves cycle time approximately 10 percent.

In contrast to arithmetic exceptions, memory management exceptions, such as page faults, are reported precisely. This is not as much of a burden on implementers as precise arithmetic exceptions would be, and lack of precise memory management faults would be a severe burden on software writers.

### *Shared-memory Multiprocessing*

The Alpha AXP architecture's shared-memory multiprocessing model is an integral part of the design. It is not the add-on found in other RISC architectures.

The underlying primitive for safe updating of a multiprocessor-shared memory location is a sequence of RISC instructions: load-locked, in-register modify, store-conditional, test. If this sequence completes with no interrupts, no exceptions, and no interfering write from another processor, then the store-conditional stores the modified result,

and the test indicates success: an atomic update was in fact performed.

If anything goes wrong, the store-conditional does not store a result, and the test indicates failure. The program must then retry the sequence until it succeeds. We chose this primitive sequence (quite similar to the MIPS R4000 chip design[5]) because it can be implemented in a way that scales up with processor performance. In the absence of an interfering write, the entire sequence can be done in an on-chip write-back cache, and hundreds of chips can do noninterfering sequences simultaneously. The sequence can also be used to achieve byte granularity* of writes in shared memory.[6]

The Alpha AXP architecture has no strict multiprocessor read/write ordering, whereby the sequence of reads and writes issued by one processor in a multiprocessor configuration is delivered to all other processors in exactly the order issued. Strict order is simple, but has a problem similar to that of byte stores. An implementer may easily choose only two of the three design features: pipelined writes, bus retry, or strict read/write ordering.

If one processor starts a write to location A and a write to location B, then discovers that the write to A has failed (bus parity error, etc.) and retries it successfully, then a second processor will observe the writes out of order: B, then A.

Before Alpha AXP implementations, many VAX implementations avoided pipelined writes to main memory, multibank caches, write-buffer bypassing, routing networks, crossbar memory interconnect, etc., to preserve strict read/write ordering. The Alpha AXP architecture's shared-memory model instead specifies no implicit ordering between the reads and writes issued on one processor, as viewed by a different processor. This programming model is an enabling technology for a wide variety of high-performance implementation techniques. Strict ordering can be specified when needed by insertion of explicit memory barrier (MB) instructions, quite similar to the IBM System/370 serialization design.[11]

## Data Representation and Processor State

This section describes the fundamental Alpha AXP data types and their representation in memory and registers. It also describes the complete hardware register state for each processor and outlines the additional state maintained by operating-system-specific PALcode routines. The Alpha AXP architecture differs from other RISC architectures by carefully specifying a canonical form for 32-bit data in 64-bit registers. A canonical form is a standardized choice of data representation for redundantly encoded values. Since 32-bit operations assume canonical operands and give canonical results, very few explicit conversions between 32- and 64-bit representations are needed.

The fundamental unit of data in the Alpha AXP architecture is a 64-bit quadword.* As shown in Figure 1, quadwords may reside in memory or registers. For backwards compatibility, 32-bit longwords* may also be stored in memory.

There are three fundamental data types: integer, IEEE floating point, and VAX floating point; each is available in 32-bit and 64-bit forms.[4,12] VAX floating-point values in memory have 16-bit words swapped, for compatibility with VAX (and PDP-11) formats. The VAX floating-point load and store instructions do word swapping* to give a common register order. The 32-bit load instructions expand values to 64-bit canonical form, and the 32-bit store instructions contract 64-bit values back to 32.[13] All register-to-register operations are thus done on full 64-bit values in a common integer or floating-point format. No partial-register reads or writes are done.

The canonical form of a 32-bit value in a 64-bit integer register has the most significant 33 bits all equal to bit<31>. In essence, bit<31> is kept as a "fat bit." This allows signed integer values to be used directly in 64-bit arithmetic and branches. This canonical form is maintained as a closed system (even for 32-bit data considered to be "unsigned") by using a combination of 64-bit operates, 32-bit add/subtract/multiply, and two-instruction sequences for shifts.

The canonical form of a 32-bit value in a 64-bit floating-point register has the 8-bit exponent field expanded to 11 bits and the 23-bit mantissa field expanded to 52 bits. Except for IEEE denormals,* this allows single-precision floating-point values to be used directly in double-precision arithmetic and branches. This canonical form is maintained as a closed system by using single-precision instructions.

Bytes and words (16-bit quantities) are not fundamental data types. They may be transferred between memory and registers with short sequences of instructions and manipulated in registers using normal arithmetic and the byte-manipulation instructions described in the Operate Instructions section.

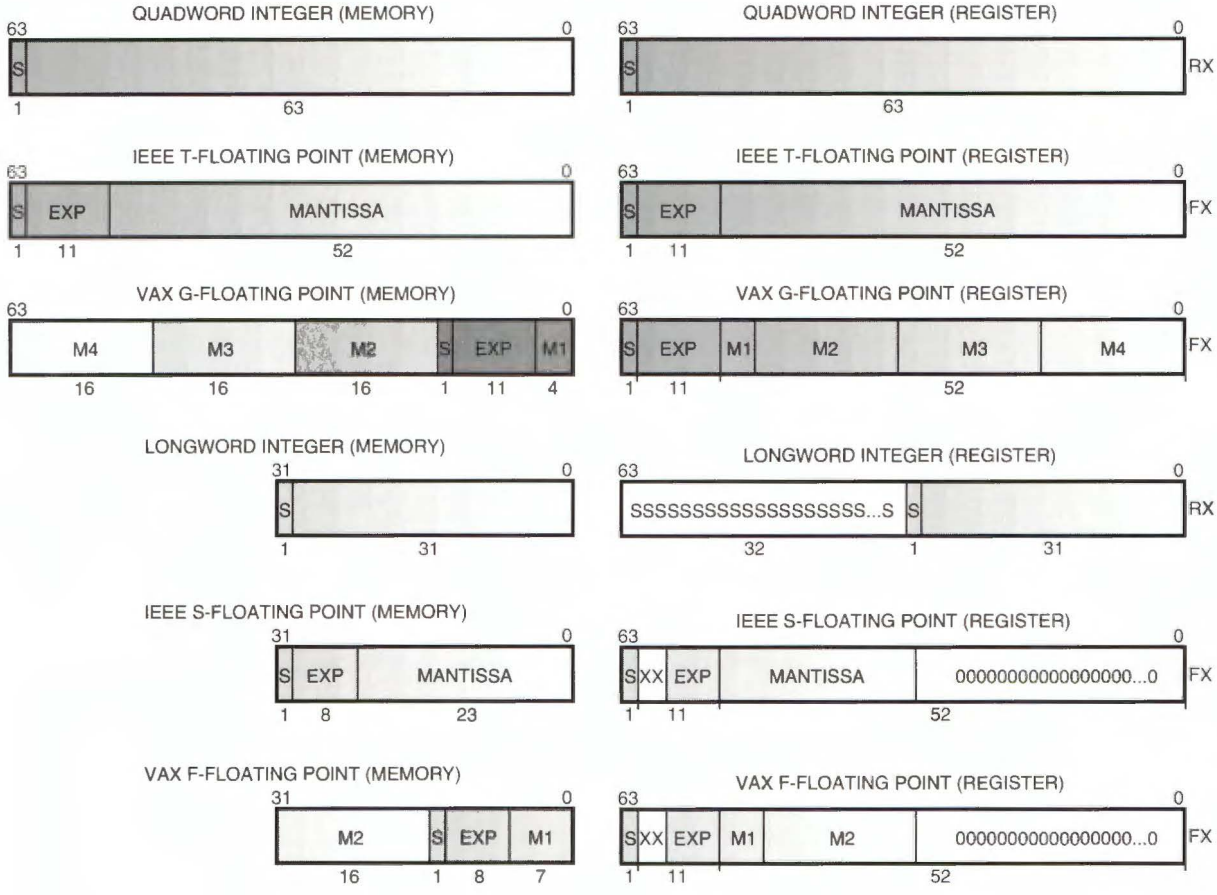



*Figure 1 Data Representation*

The hardware processor state, shown in Figure 2, includes 32 integer registers R0..R31 of 64 bits each; R31 is always zero. There are also 32 floating-point registers F0..F31 of 64 bits each; F31 is always zero. Writes to R31 and F31 are ignored.

A 64-bit program counter (PC) contains a longword-aligned virtual byte address (i.e., the low 2 bits of the PC are always zero). The VAX architecture keeps the PC in general register 15, where it is directly used for PC-relative memory addressing. In the Alpha AXP architecture, however, code and data pages are usually separated by 64 kilobytes (KB) or more to allow separate memory protection, but the 16-bit displacement in load/store instructions cannot span more than 64KB.

The hardware processor state includes a lock flag and a locked physical address for the load-locked/store-conditional sequence. It also has a floating-point control register containing the IEEE dynamic rounding mode.*

Hardware implementations may optionally include a pair of state registers for memory prefetching (FETCH/FETCH_M instructions), and an optional interrupt flag for use only by translated VAX OpenVMS AXP programs that reproduce complex instruction set computer (CISC*) instruction atomicity using a sequence of RISC instructions.[6]

In addition to the above hardware state, the privileged architecture library routines for the various operating systems implement additional state. This state may be maintained by hardware or (PALcode) software, at the option of the implementer, and it varies from one operating system to another. Typical PALcode state includes a processor status (PS) word, kernel and user stack pointers, a process control block base for context switching, a process-unique value for threads, and a processor number for multiprocessor dispatching. Additional PALcode state may include a floating-point enable bit, interrupt priority level, and translation look-aside

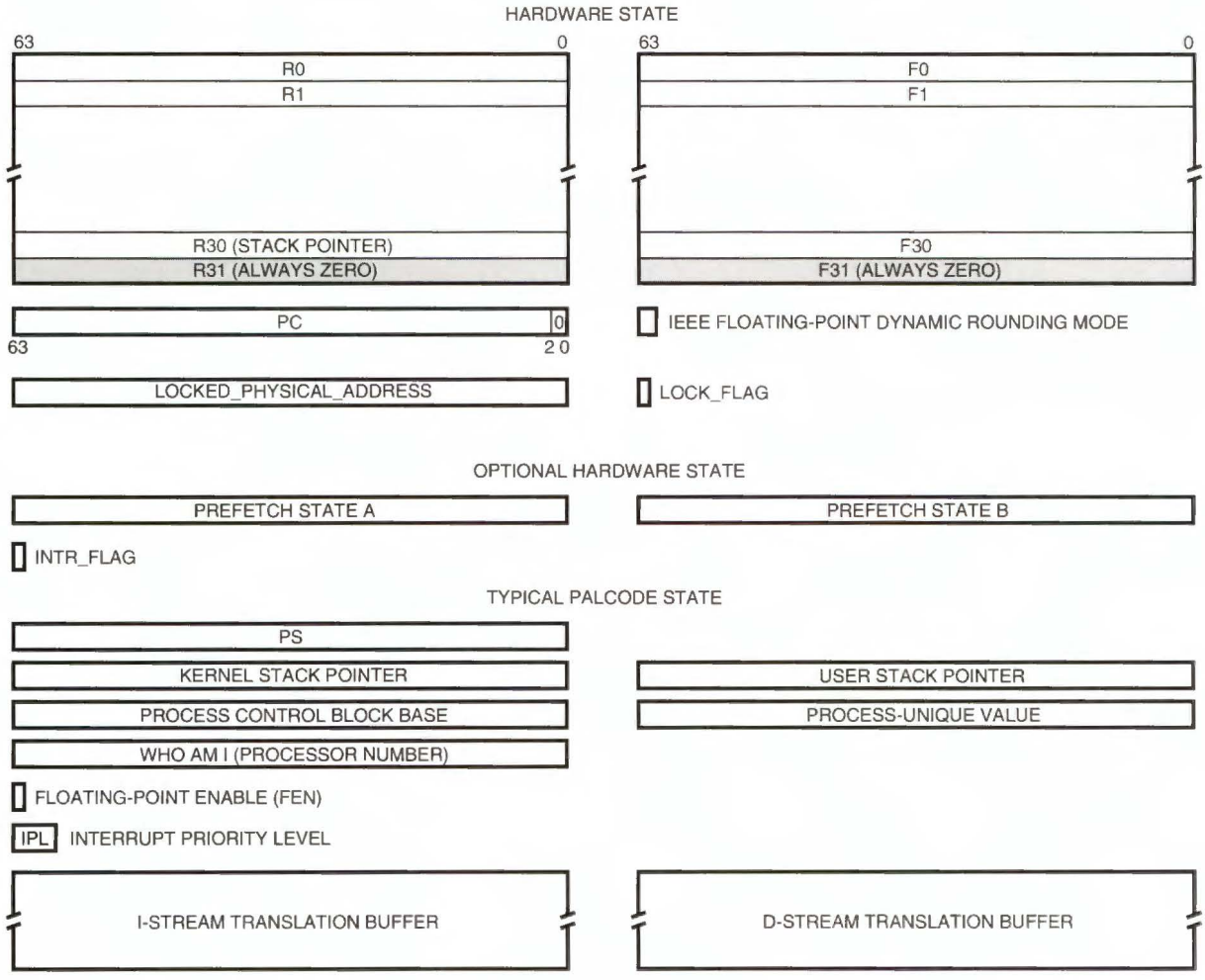



*Figure 2 Per-processor State*

buffers for mapping instruction-stream and data-stream virtual addresses. All of this state is soft in the sense that it is defined only in relationship to the PALcode routines for a specific operating system. In a multiprocessor implementation, all of the above state is replicated for each processor.

## Memory Access

Alpha AXP memory is byte addressed, using the lowest-numbered byte of a datum. Only aligned longwords or quadwords may be accessed: an aligned longword is a four-byte datum whose address is a multiple of four; an aligned quadword is an eight-byte datum whose address is a multiple of eight. Normal load or store instructions that specify an unaligned address take a precise data alignment trap to PALcode (which may do the access using two aligned accesses or report a fatal error, depending on the operating system design).

Alpha AXP implementations allow data to be accessed using either a little-endian* view (byte 0 is the low byte of an integer), or a big-endian* view (byte 0 is the high byte of an integer). As described in the Load/Store Instructions section, there is a one-instruction bias in the sequences for little- and big-endian byte manipulation.

Virtual addresses are a full 64 bits; implementations may restrict addresses to have some number of identical high-order bits, but must always distinguish at least 43 bits. Virtual addresses are mapped in an operating-specific way to physical addresses, using fixed-size pages. Memory protection is done on a per-page basis. Address mapping errors (e.g., protection, page faults) take precise traps to

PALcode. Each page may also be marked to provide a fault on each read, write, or instruction-fetch.

Virtual addresses may be further qualified by address space numbers (ASNs), to allow multiple disjoint addresses spaces. The choice of disjoint or common mapping across all processes is done on a per-page basis.

The virtual- to physical-address mapping is done on a per-page basis. Each implementation may have a page size of 8KB, 16KB, 32KB, or 64KB. The 64KB upper bound allows a linker to allocate blocks of memory with differing protection or ASN properties far enough apart to work on all implementations. The virtual- to physical-address mapping can be many to one, i.e., synonyms are allowed. In a multiprocessor implementation, shared main memory locations have the same physical address on all processors. Per-processor unshared locations are also allowed.

Memory has longword granularity: two processors may simultaneously access adjacent longwords without mutual interference. The load-locked/store-conditional sequence discussed previously can be used to achieve multiprocessor byte granularity.

Input/output is memory mapped: some physical memory addresses may refer to I/O device registers whose access triggers side effects (such as the transfer of data). Side effects on reads are discouraged.

## Instruction Formats

Four fundamental instruction formats—operate, memory, branch, and CALL_PAL—are shown in Figure 3. All instructions are 32 bits wide and reside in memory at aligned longword addresses. Each instruction contains a 6-bit opcode field and zero to three 5-bit register-number fields, RA, RB, and RC. The remaining bits contain function (opcode extension), literal, or displacement fields. To minimize register file ports in fast implementations, RB is never written, and RC is never read.

All the operate instructions are three-operand register-to-register, calculating *RC = RA operate RB*. In integer operates, the opcode and a 7-bit function field specify the exact operation. Integer operates may have an 8-bit zero-extended literal instead of RB. In floating-point operates, the opcode and an 11-bit function field specify the exact operation. There are no floating-point literals.

Memory format instructions are used for loads, stores, and a few miscellaneous operations. Loads and stores are two-operand instructions, specifying a register RA and a base-displacement virtual byte address. The effective address calculation sign extends the 16-bit displacement to 64 bits and adds the 64-bit RB base register (ignoring overflow). The resulting virtual byte address is mapped to a physical address. The miscellaneous instructions make other uses of the RA, RB, and displacement fields.

Branch format instructions specify a single register RA and a signed PC-relative longword displacement. The branch target calculation shifts the 21-bit displacement left by 2 bits to make it a longword (not byte) displacement, then sign extends it and adds it to the updated PC. Conditional branch instructions test register RA, and unconditional branches write the updated PC to RA for subroutine linkage. The large longword displacement allows a range of ±4MB, substantially reducing the need for branches around or to other branches.

The CALL_PAL instruction has only a 6-bit opcode and a 26-bit function field. The function field is a small integer specifying one of a few dozen privileged architecture library subroutines.

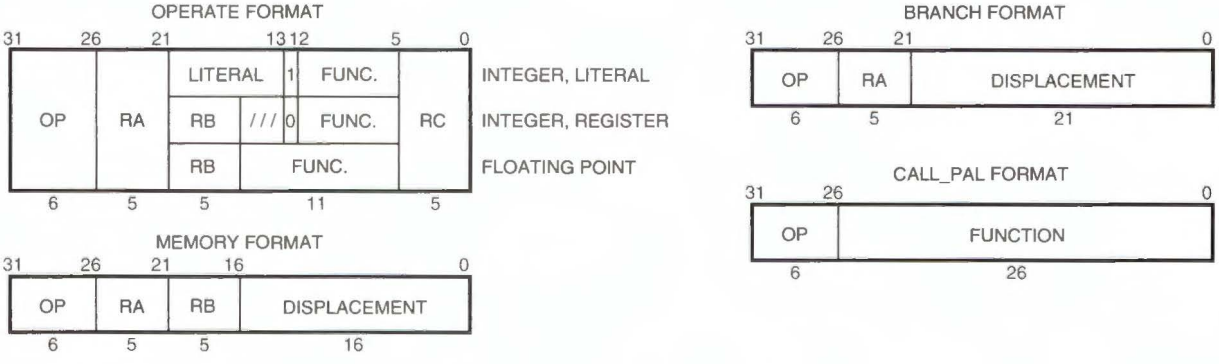



*Figure 3 Instruction Formats*

### Operate Instructions

There are five groups of register-to-register operate instructions: integer arithmetic, logical, byte-manipulation, floating-point, and miscellaneous. All instructions operate on 64-bit quadwords unless otherwise specified.

*Integer Arithmetic Instructions* The integer arithmetic instructions are add, subtract, multiply, and compare. Add, subtract, and multiply have variants that enable arithmetic overflow traps. They also have longword variants that check for 32-bit overflow (instead of 64) and force the high 33 bits of the result to all equal bit<31>. Add and subtract also have scaled variants that shift the first operand left by 2 or 3 bits (with no overflow checking) to speed up simple subscripted address arithmetic. The UMULH instruction (from PRISM) gives the high 64 bits of an unsigned 128-bit product and may be used for dividing by a constant. There is no integer divide instruction; a software subroutine is used to divide by a nonconstant. The compare instructions are signed or unsigned and write a Boolean result (0 or 1) to the target register.

*Logical Instructions* The logical instructions are AND, OR, and XOR, with the second operand optionally complemented (ANDNOT, ORNOT, XORNOT). The shifts are shift left logical, shift right logical, and shift right arithmetic. The 6-bit shift count is given by RB or a literal. The conditional move instructions test RA (same tests as the branching instructions) and conditionally move RB to RC. These can be used to eliminate branches in short sequences such as MIN(a,b).

*Byte-manipulation Instructions* The byte-manipulation instructions are used with the load and store unaligned instructions to manipulate short unaligned strings of bytes. Long strings should be manipulated in groups of eight (aligned quadwords) whenever possible. The byte-manipulation instructions are fundamentally masked shifts. They differ from normal shifts by having a byte count (0..7) instead of a bit count (0..63), and by zeroing some bytes of the result, based on the data size given in the function field.

The extract (EXTxx) instructions extract part of a 1-, 2-, 4-, or 8-byte field from a quadword and place the resulting bytes in a field of zeros. A single EXTxL instruction can perform byte or word loads, pulling the datum out of a quadword and placing it in the low end of a register with high-order zeros. A pair of EXTxL/EXTxH instructions can perform unaligned loads, pulling the two parts of an unaligned datum out of two quadwords and placing the parts in result registers. A simple OR operation can then combine the two parts into the full datum.

The insert (INSxx) and mask (MSKxx) instructions position new data and zero out old data in registers for storing bytes, words, and unaligned data. If the Alpha AXP architecture were a four-operand one, inserting and masking could have been combined into a single instruction.

The compare-byte instruction allows character-string search and compare to be done eight bytes at a time. The ZAP instructions allow zeroing of arbitrary patterns of bytes in a register. These instructions allow very fast implementations of the C language string routines, among other uses.

*Floating-point Arithmetic Instructions* The floating-point arithmetic instructions are add, subtract, multiply, divide, compare, and convert. The first four have variants for IEEE and VAX floating-point, and single- and double-precision data types. They also have variants that enable combinations of arithmetic traps and that specify the rounding mode. The single-precision instructions write canonical 64-bit results, but do exponent checking and rounding to single-precision ranges. The compare instructions write a Boolean result (0 or nonzero) to the target register. The convert instructions transfer between single and double, floating-point and integer, and two forms of VAX double (D-float and G-float). A combination of hardware and software provides full IEEE arithmetic. Operations on VAX reserved operands,* dirty zeros,* IEEE denormals, infinities,* and not-a-numbers* are done in software.

There are also a few floating-point instructions that move data without applying any interpretation to it. These include a complete set of conditional move instructions similar to the integer conditional moves.

*Miscellaneous Instructions* The miscellaneous instructions include: memory prefetching instructions to help decrease memory latency, a read cycle counter instruction for performance measurement, a trap barrier instruction for forcing precise arithmetic traps, and memory barrier instructions for forcing multiprocessor read/write ordering.

## *Load/Store Instructions*

The load and store instructions only move data. They never apply an interpretation to the data and therefore never take any data-dependent traps. This design allows moving completely arbitrary bit patterns in and out of registers and allows completely transparent saving/restoring of registers.

The integer load and store quadword unaligned (LDQ_U, STQ_U) instructions ignore the low three bits of the byte address and always transfer an aligned quadword. These instructions are used with the in-register byte manipulation instructions to operate on byte, word, and unaligned data by short sequences of RISC instructions.

Example 1 in Figure 4 shows a two-instruction sequence for loading a byte into the low end of a register, using little-endian byte numbering. Example 2 shows a similar sequence for loading a byte into the high end of a register, using big-endian byte numbering. Example 3 shows a sequence for storing a byte (the first two and last two instructions might issue simultaneously on the first Alpha AXP implementation). Example 4 shows a sequence for an explicit unaligned load quadword (no data alignment trap).

The integer load-locked and store-conditional (LDQ_L, LDL_L, STQ_C, STL_C) instructions are included in the architecture to facilitate atomic updates of multiprocessor-shared data. As described above, they can be used in short sequences of RISC instructions to do atomic read-modify-writes. Example 5 shows a sequence for doing a multiprocessor test-and-set. Note that changing the LDQ_U/STQ_U in Example 3 to AND/LDQ_L/STQ_C/BEQ gives a byte-store sequence that is safe to use with multiprocessor-shared data.

There are two related load address instructions. LDA calculates the effective address and writes it into RC. LDAH first shifts the displacement left 16 bits, then calculates the effective address and writes it into RC. LDAH is included to give a simple way of creating most 32-bit constants in a pair of instructions. (Because LDA sign-extends the displacement, some values in the range 000000007FFF8000 .. 000000007FFFFFFF require three instructions.) Constants of 64 bits are loaded with LDQ instructions.

## *Branching Instructions*

The branch instructions include conditional branches, unconditional branches, and calculated jumps. In addition to the previously described conditional moves, the architecture contains hints to improve branching performance.

The integer conditional branches test register RA for an opcode-specified condition (>0 >=0 =0 !=0 <=0 <0 even odd) and either branch to the target address or fall through to the updated PC address. The floating-point conditional branches are the same, except they do not include even/odd tests. Arbitrary testing (and faulting on VAX or IEEE nonfinite values) can be done by sequences of compare instructions and branch instructions. Logical or arithmetic instructions can combine compare results without using branches.

Unconditional branches write the updated PC to RA for subroutine linkage and branch to the target address. RA = R31 may be used if no linkage is needed.

Calculated jumps write the updated PC to RA and jump to the target address in RB. Calculated jumps are used for subroutine call, return, CASE (or SWITCH) statements, and coroutine linkage.

The architecture specifies three kinds of branching hints in instructions. The hints need not be correct, but to the extent that they are, implementations may perform faster.

The first form of hint is an architected static branch prediction rule: forward conditional branches are predicted not-taken, and backward ones taken. To the extent that compilers and hardware implementers follow this rule, programs can run more quickly with little hardware cost. This hint does not eliminate the use of dynamic branch prediction in an implementation, but it may reduce the need to use it.

The second form describes computed jump targets. Unused instruction bits are defined to give the low bits of the most likely target, using the same target calculation as unconditional branches. The 14 bits provided are enough to specify the instruction offset within a page, which is often enough to start a fastest-level instruction-cache read many cycles before the actual target value is known.

The third form describes subroutine and coroutine returns. By marking each branch and jump as call, return, or neither, the architecture provides enough information to maintain a small stack of likely subroutine return addresses within an implementation. This implementation stack can be used to prefetch subroutine returns quickly.

The conditional move instructions (discussed previously in the Logical Instructions section and the Floating-point Arithmetic Instructions section)

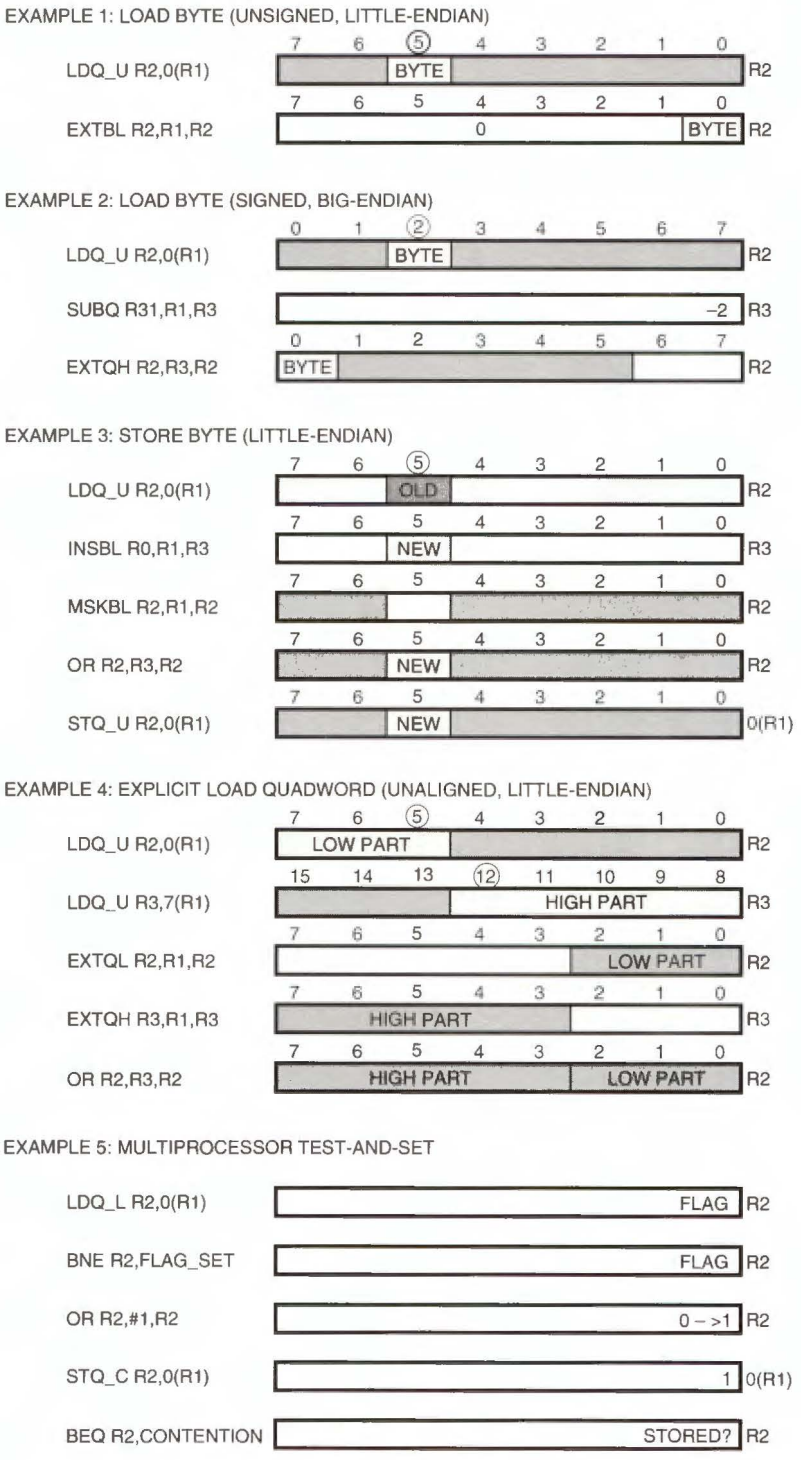



*Figure 4 Load/Store Instructions*

and the branching hints eliminate some branches and speed up the remaining ones without compromising multiple instruction issue.

## Supervision

The actions underpinning an operating system are performed in PALcode subroutines and are a flexible part of the architecture. All asynchronous events, such as interrupts, exceptions, and machine errors, are mediated by PALcode routines. PALcode establishes the initial state of the machine before execution of the first software instruction. PALcode routines mediate all accesses to physical hardware resources, including physical main memory and memory-mapped I/O device registers.

This design allows implementers to craft a set of PALcode routines that closely match an operating system design, not only for traditional operating systems, but also for specialized environments such as real-time or highly secure computing. As new computing paradigms are adopted and new operating systems are created, the Alpha AXP architecture may well prove flexible enough to accommodate them efficiently.

## Future Changes

The Alpha AXP architecture will surely change during its lifetime. In addition to the PALcode flexibility discussed above, explicit performance flexibility and instruction-set flexibility exist in the architecture.

Architectural fields that are too small can limit performance. The Alpha AXP architecture therefore has many fields deliberately sized for later expansion.

Although initial implementations use only 43 bits of virtual address, they check the remaining 21 bits, so that software can run unmodified on later implementations that use (up to) all 64 bits. Furthermore, although initial implementations use only 34 bits of physical address, the architected page table entry (PTE) formats and page-size choices allow growth to 48 bits. By expanding into a 16-bit PTE field that is not currently used by mapping hardware, another 16 bits of physical address growth can be achieved, if ever needed.

Initial implementations also use only 8KB pages, but the design accommodates limited growth to 64KB pages. Beyond that, page table granularity hints allow groups of 8, 64, or 512 pages to be treated as a single large page, thus effectively extending the page-size range by a factor of over 1,000. Each architected PTE format also has one bit reserved for future expansion.

Several other soft PALcode registers, such as the PS or ASN, that need only a few bits today are allocated a full 64 bits for future expansion.

Exception processing can limit performance. PALcode routines deliver exceptions to an operating system, so the design can be gradually improved. In fact, PALcode routines for the data alignment have been improved in the OpenVMS AXP and DEC OSF/1 AXP operating systems. Some currently specified software exceptions (such as IEEE denormal arithmetic) could be moved into PALcode or hardware.

There are a number of areas of instruction-set flexibility designed into the architecture. Four of the 6-bit opcodes are nominally reserved for adding integer and floating-point aligned octaword* (128-bit) load/store instructions.[14] Nine more 6-bit opcodes remain for other expansion. Within each opcode, the function field contains room for further expansion. For example, the scaled add/subtract functions were added between prototype chip and product chip. The fact that the function fields are not fully policed is a mistake.

Within the IEEE floating-point function field, code points are nominally reserved for double-extended* precision (128-bit) arithmetic. Within the memory barrier instruction group, three code points were reserved for subset barriers. One of these has already been redefined as a write-write barrier.

Not all changes involve growth. There are subsetting rules defined for removing either one or both (IEEE and VAX) floating-point data types. If both are removed, the floating-point registers can also be removed. The AMOVxx PALcode routines and RS/RC instructions are defined as optional and can be deleted when the transition of translated VAX code is completed. Other unneeded PALcode routines can also be removed eventually.

## Summary

The goals that shaped the Alpha AXP architecture design have largely been realized. For high performance, the first implementation (the DECchip 21064 microprocessor) is listed in the October 1992 *Guinness Book of Records* as the world's fastest single-chip microprocessor. It is too early to measure longevity, but the fact that we had designed-in flexibility in places that changed during development is at least encouraging. OpenVMS AXP, DEC OSF/1 AXP,

and Windows NT operating systems all run on Alpha AXP implementations today. Programs from the VAX and MIPS architectures transport easily to Alpha AXP implementations and run quickly. Many of the ideas in the Alpha AXP design are now being adopted by other architectures in the industry.

## Appendix

*Binary translation*—A software technique to change an executable program written for one architecture/operating-system pair into an equivalent program for a different architecture/operating-system pair.

*Big-endian memory addressing*—A view of memory in which byte 0 of an operand contains the most significant (sign) bit of an integer. Compare little-endian memory addressing.

*Byte*—An 8-bit datum.

*Byte granularity*—The appearance that two processors can update adjacent bytes in memory without interfering with each other.

*CISC*—Complex instruction set computer, characterized by variable-length instructions, a wide variety of memory addressing modes, and instructions that combine one or more memory accesses with arithmetic. CISC designs express computation as a few complex steps.

*IEEE denormalized number (denormal)*—A floating-point number with magnitude between zero and the smallest representable normalized number. Numbers in this range are typically not representable in other floating-point arithmetic systems; such systems might signal an underflow exception or force a result to zero instead.

*IEEE double-extended format*—A loosely specifed floating-point format with at least 64 significant bits of precision and at least 15 bits of exponent width; typically implemented using a total of 80 or 128 bits.

*IEEE dynamic rounding mode*—One of four different rounding rules.

*IEEE floating-point*—A form of computer arithmetic specified by IEEE standard 754.[12] IEEE arithmetic includes rules for denormalized numbers, infinities, and not-a-numbers. It also specifies four different modes for rounding results.

*IEEE infinity*—An operand with an arbitrarily large magnitude.

*IEEE not-a-number (NaN)*—A symbolic entity encoded in a floating-point format. The IEEE standard specifies some exceptional results (e.g., 0/0) to be NaNs.

*Linear addressing*—A memory addressing technique in which all addresses form a single range, from 0 to the largest possible address. Subscript calculations can create any address in the entire range.

*Little-endian memory addressing*—A view of memory in which byte 0 of an operand contains the least significant bit of an integer. The terms little-endian and big-endian are borrowed from *Gulliver's Travels* in which religious wars were waged over which end of an egg to break.

*Longword*—A 32-bit datum.

*Multiple instruction issue*—A high-performance computer implementation technique of starting more than one instruction at once. An implementation that starts (up to) two instructions at once is called dual-issue; four instructions, quad-issue or four-way issue; etc.

*Octaword*—A 128-bit datum.

*Quadword*—A 64-bit datum.

*RISC*—Reduced instruction set computer, characterized by fixed-length instructions, simple memory addressing modes, and a strict decoupling of load/store memory access instructions from register-to-register arithmetic instructions. RISC designs express computation as many simple steps.

*Segmented addressing*—A memory addressing technique in which addresses are broken into two or more parts (segments). Subscript calculations can only be done within a single segment, and elaborate software techniques are needed to extend addressing beyond a single segment.

*VAX dirty zero*—A zero value represented with a non-zero faction; must be converted to a true zero result.

*VAX floating-point*—A form of computer arithmetic specified by the VAX architecture manual.[4] VAX arithmetic includes rules for reserved operands and dirty zeros.

*VAX reserved operand*—A non-number that signals an exception when used as an operand in VAX floating-point arithmetic.

*VAX word swapping*—The rearrangement needed for the 16-bit pieces of a VAX floating-point number

to put the fields in a more usual order; this is an artifact of the PDP-11 16-bit architecture.

*Word*—A 16-bit datum.

## Acknowledgments

Hundreds of people have worked on the Alpha AXP architecture, hardware, and software. Many Alpha AXP architectural ideas came from the PRISM design, most notably the PALcode idea.[3] The architecture work was done in the rich environment of dozens and later hundreds of bright, thoughtful, and outspoken professional peers. Ellen Batbouta, Dileep Bhandarkar, Richard Brunner, Wayne Cardoza, Dave Cutler, Daniel Dobberpuhl, Robert Giggi, Henry Grieb, Richard Grove, Robert Halstead, Jr., Michael Harvey, Nancy Kronenberg, Raymond Lanza, Stephen Morris, William Noyce, Charles Nylander, Dave Orbits, Mary Payne, Audrey Reith, Robert Supnik, Benjamin Thomas, Catharine van Ingen, and Rich Witek all contributed directly to the written specification. Rich Witek is co-architect and is the other half of the term "we" used in this paper.

## References and Notes

1. G. Amdahl, G. Blaauw, and F. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development,* vol. 8, no. 2 (April 1967): 87–101.

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

3. R. Conrad et al., "A 50 MIPS (Peak) 32/64b Microprocessor," *ISSCC Digest of Technical Papers* (February 1989): 76–77.

4. R. Brunner, ed., *VAX Architecture Reference Manual* Second Edition (Bedford, MA: Digital Press, 1991).

5. G. Kane and J. Heinrich, *MIPS RISC Architecture* (Englewood Cliffs, NJ: Prentice-Hall, 1992).

6. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 137–152.

7. The little-endian bias is very slight; both big- and little-endian Alpha AXP systems and software are in fact being built.

8. There are two special-resource anomalies in the architecture that we were unable to avoid: the dedicated state for the load-locked instruction and the dynamic rounding-mode register required for full IEEE conformance.

9. This is borne out in a large customer's recent C string manipulation benchmark result, running 3 to 6 times faster than the customer's expectation (which was based solely on clock rate ratios).

10. *Cray-1 Computer System Reference Manual,* Form 2240004 (Minneapolis: Cray Research, Inc., 1977).

11. *IBM System/370 Principles of Operation,* Form GA22-7000-4 (Armonk, NY: IBM Corporation, 1974): 28.

12. Institute of Electrical and Electronics Engineers, "Binary Floating-point Arithmetic for Microprocessor Systems," Standard Number IEEE-754 (New York, 1985).

13. The careful reader will notice that Alpha AXP implementations require a longword shifter in the load/store path for 32-bit operands. Although we briefly considered a design with no 32-bit operands, we decided to keep 32-bit load/store support for good business reasons. Similarly, Alpha AXP implementations require a word swapper in the load/store path for VAX floating-point operands. We decided to keep VAX floating-point support for good business reasons. Depending on market needs, VAX floating-point support can be removed in the future.

14. Many commercially successful architectures have grown to double-width memory implementations in mid-life: the IBM 709 series from 36 to 72 bits; the IBM System/360 series from 32 to 64 bits; the Digital PDP-11 series from 16 to 32 bits; and the Digital VAX series from 32 to 64 bits. This trend is likely to continue.

*Daniel W. Dobberpuhl*
*Richard T. Witek*
*Randy Allmon*
*Robert Anglin*
*David Bertucci*
*Sharon Britton*
*Linda Chao*
*Robert A. Conrad*
*Daniel E. Dever*
*Bruce Gieseke*
*Soha M.N. Hassoun*
*Gregory W. Hoeppner*
*Kathryn Kuchler*
*Maureen Ladd*
*Burton M. Leary*
*Liam Madden*
*Edward J. McLellan*
*Derrick R. Meyer*
*James Montanaro*
*Donald A. Priore*
*Vidya Rajagopalan*
*Sridhar Samudrala*
*Sribalan Santhanam*

# A 200-MHz 64-bit Dual-issue CMOS Microprocessor

*A 400-mips/200-MFLOPS (peak) custom 64-bit VLSI CPU chip is described. The chip is fabricated in a 0.75-μm CMOS technology utilizing three levels of metalization and optimized for 3.3-V operation. The die size is 16.8 mm × 13.9 mm and contains 1.68 million transistors. The chip includes separate 8KB instruction and data caches and a fully pipelined floating-point unit that can handle both IEEE and VAX standard floating-point data types. It is designed to execute two instructions per cycle among scoreboarded integer, floating-point, address, and branch execution units. Power dissipation is 30 W at 200-MHz operation.*

A reduced instruction set computer (RISC)-style microprocessor has been designed and tested that operates up to 200 megahertz (MHz). The chip implements a new 64-bit architecture, designed to provide a huge linear address space and to be devoid of bottlenecks that would impede highly concurrent implementations. Fully pipelined and capable of issuing two instructions per clock cycle, this implementation can execute up to 400 million operations per second. The chip includes an 8-kilobyte (KB) I-cache, 8KB D-cache and two associated translation buffers, a four-entry, 32-byte-per-entry write buffer, a pipelined 64-bit integer execution unit with a 32-entry register file, and a pipelined floating-point unit (FPU) with an additional 32 registers. The pin interface includes integral support for an external secondary cache. The package is a 431-pin pin grid array (PGA) with 140 pins dedicated to $V_{DD}/V_{SS}$ (power supply voltage/ground). The chip is fabricated in a 0.75-micrometer (μm) n-well complementary metal-oxide semiconductor (CMOS) process with three layers of metalization. The die measures 16.8 millimeters (mm) × 13.9 mm and contains 1.68 million transistors. Power dissipation is 30 watts (W) from a 3.3-volt (V) supply at 200 MHz.



## CMOS Process Technology

The chip is fabricated in a 0.75-μm, 3.3-V, n-well CMOS process optimized for high-performance microprocessor design. Process characteristics are shown in Table 1. The thin gate oxide and short transistor lengths result in the fast transistors required to operate at 200 MHz. There are no explicit bipolar devices in the process as the incremental process complexity and cost were deemed

**Table 1 Process Description**

| | |
|---|---|
| Feature size | 0.75 μm |
| Channel length | 0.5 μm |
| Gate oxide | 10.5 nm |
| $V_{tn}/V_{tp}$ | 0.5 V/−0.5 V |
| Power supply | 3.3 V |
| Substrate | P-epitaxial with n-well |
| Salicide | Cobalt-disilicide in diffusions and gates |
| Buried contact | Titanium nitride |
| Metal 1 | 0.75-μm AlCu, 2.25-μm pitch (contacted) |
| Metal 2 | 0.75-μm AlCu, 2.625-μm pitch (contacted) |
| Metal 3 | 2.0-μm AlCu, 7.5-μm pitch (contacted) |

too large in comparison to the benefits provided—principally more area-efficient large drivers such as clock and I/O.

The metal structure is designed to support the high operating frequency of the chip. Metal 3 is very thick and has a relatively large pitch. It is important at these speeds to have a low-resistance metal layer available for power and clock distribution. It is also used for a small set of special signal wires such as the data buses to the pins and the control wires for the two shifters. Metal 1 and metal 2 are maintained at close to their maximum thickness by planarization and by filling metal 1 and metal 2 contacts with tungsten plugs. This removes a potential weak spot in the electromigration characteristics of the process and allows more freedom in the design without compromising reliability.

## Alpha AXP Architecture

The computer architecture implemented is a 64-bit load/store RISC architecture with 168 instructions, all 32 bits wide.[1] Supported data types include 8-, 16-, 32-, and 64-bit integers and both Digital and IEEE 32- and 64-bit floating-point formats. Each of the two register files, integer and floating point, contains 32 entries of 64 bits with one entry in each being a hardwired zero. The program counter and virtual address are 64 bits. Implementations can subset the virtual address size, but are required to check the full 64-bit address for sign extension. This ensures that when later implementations choose to support a larger virtual address, programs will still run and not find addresses that have dirty bits in the previously "unused" bits.

The architecture is designed to support high-speed multi-issue implementations. To this end the architecture does not include condition codes, instructions with fixed source or destination registers, or byte writes of any kind (byte operations are supported by extract and merge instructions within the CPU itself). Also there are no first-generation artifacts that are optimized around today's technology, which would represent a long-term liability to the architecture.

## Chip Microarchitecture

The block diagram (Figure 1) shows the major functional blocks and their interconnecting buses, most of which are 64 bits wide. The chip implements four functional units: the integer unit (IRF plus E-box), the floating-point unit (FRF plus F-box), the load/store unit (A-box), and the branch unit (distributed). The bus interface unit (BIU), described in the next section, handles all communication between the chip and external components. The microphotograph (Figure 2) shows the boundaries of the major functional units. The dual-issue rules are a direct consequence of the register file ports, the functional units, and the I-cache interface. The integer register file (IRF) has two read ports and one write port dedicated to the integer unit, and two read and one write port shared between the branch unit and the load/store unit. The floating-point register file (FRF) has two read ports and one write port dedicated to the floating unit, and one read and one write port shared between the branch unit and the load/store unit. This leads to dual-issue rules that are quite general:

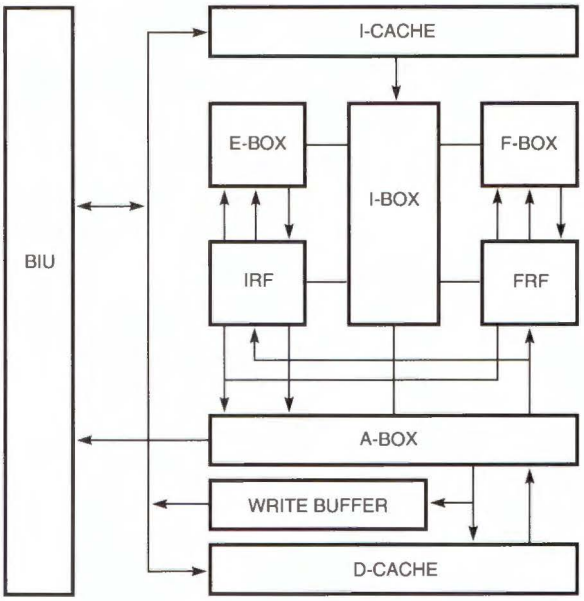



*Figure 1 CPU Chip Block Diagram*

- Any load/store in parallel with any operate
- An integer operate in parallel with a floating operate
- A floating operate and a floating branch
- An integer operate and an integer branch

except that integer store and floating operate and floating store and integer operate are disallowed as pairs.

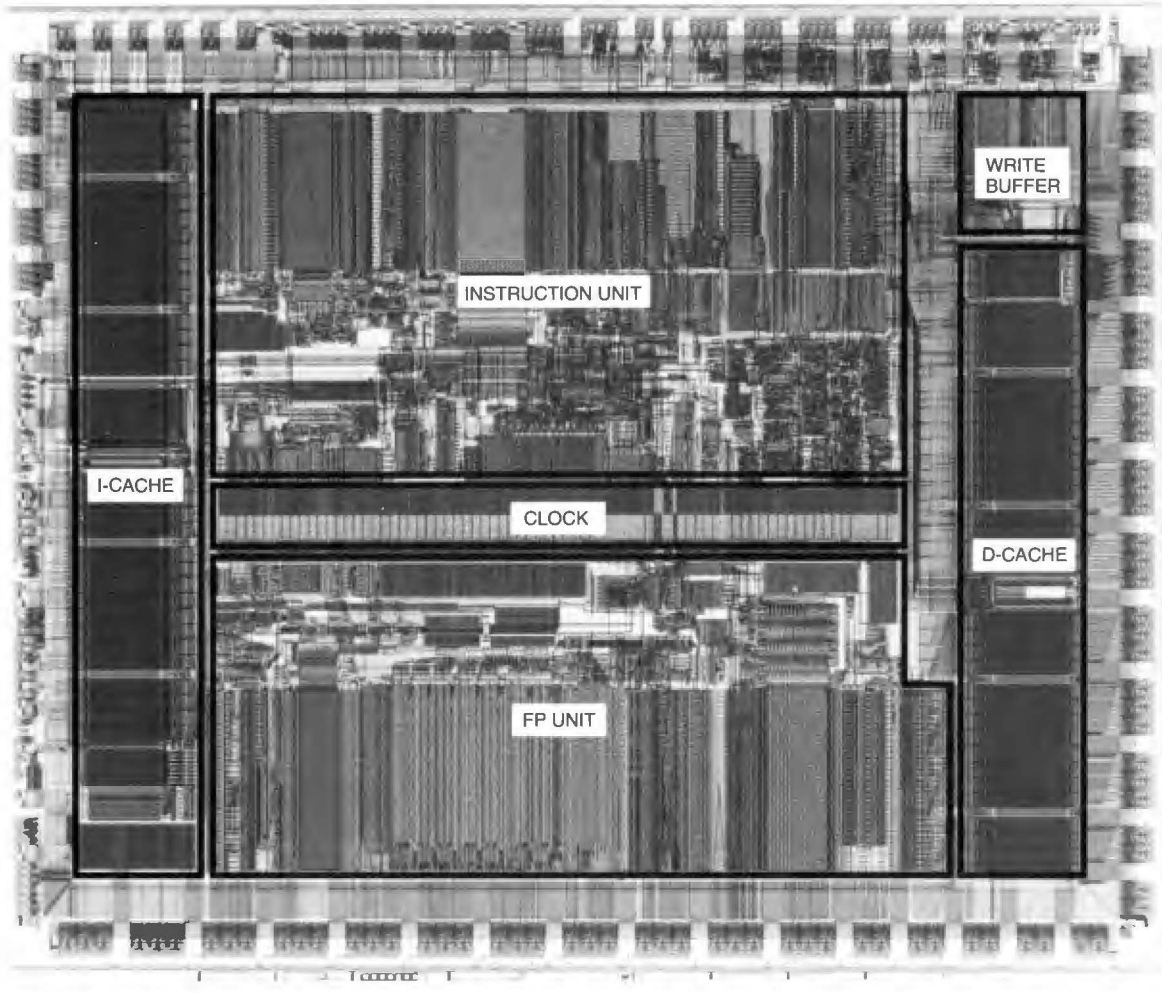
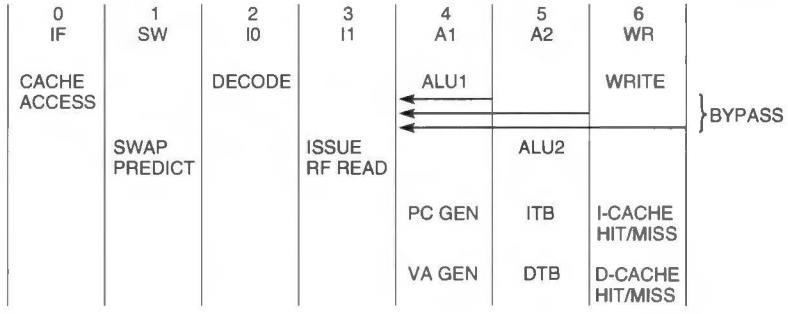



*Figure 2 Microphotograph of Chip*

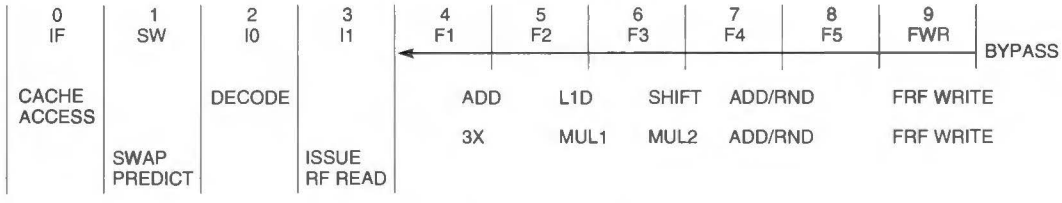
As shown in Figure 3a, the integer pipeline is 7 stages deep, where each stage is a 5-nanosecond (ns) clock cycle. The first four stages are associated with instruction fetching, decoding, and scoreboard checking of operands. Pipeline stages 0 through 3 can be stalled. Beyond 3, however, all pipeline stages advance every cycle. Most arithmetic and logic unit (ALU) operations complete in cycle 4, allowing single-cycle latency, with the shifter being the exception. Primary cache accesses complete in cycle 6, so cache latency is three cycles. The chip will do hits under misses to the primary D-cache.

The I-stream is based on autonomous prefetching in cycles 0 and 1 with the final resolution of I-cache hit not occurring until cycle 5. The prefetcher includes a branch history table and a subroutine return stack. The architecture provides a convention for compilers to predict branch decisions and destination addresses, including those for register indirect jumps. The penalty for branch mispredict is four cycles.

The floating-point unit is a fully pipelined 64-bit floating-point processor that supports both VAX standard and IEEE standard data types and rounding modes. It can generate a 64-bit result every cycle for all operations except divide. As shown in Figure 3b, the floating-point pipeline is identical and mostly shared with the integer pipeline in stages 0 through 3; however, the execution phase is three cycles longer. All operations, 32- and 64-bit (except divide) have the same timing. Divide is handled by a nonpipelined, single bit per cycle, dedicated divide unit.




*(a) Integer Unit Pipeline Timing*




*(b) Floating-point Unit Pipeline Timing*

KEY:

| | |
|---|---|
| PC GEN | GENERATE NEW PROGRAM COUNTER VALUE |
| VA GEN | GENERATE NEW VIRTUAL ADDRESS |
| ITB | INSTRUCTION TRANSLATION BUFFER |
| DTB | DATA TRANSLATION BUFFER |

*Figure 3 Pipeline Timing*

In cycle 4, the register file data is formatted to fraction, exponent, and sign. In the first-stage adder, exponent difference is calculated and a 3 × multiplicand is generated for multiplies. In addition, a predictive leading 1 or 0 detector using the input operands is initiated for use in result normalization. In cycles 5 and 6, for add/subtract, alignment or normalization shift and sticky-bit calculation are performed. For both single- and double-precision multiplication, the multiply is done in a radix-8 pipelined array multiplier. In cycles 7 and 8, the final addition and rounding are performed in parallel and the final result is selected and driven back to the register file in cycle 9. With an allowed bypass of the register write data, floating-point latency is six cycles.

The CPU contains all the hardware necessary to support a demand paged virtual memory system. It includes two translation buffers to cache virtual-to-physical address translation. The instruction translation buffer contains 12 entries, 8 that map 8KB pages and 4 that map 4-megabyte (MB) pages. The data translation buffer contains 32 entries that can map 8KB, 64KB, 512KB, or 4MB pages.

The CPU supports performance measurement with two counters that accumulate system events on the chip such as dual-issue cycles and cache misses or external events through two dedicated pins that are sampled at the selected system clock speed.

## External Interface

The external interface (Figure 4) is designed to directly support an off-chip backup cache that can range in size from 128KB to 8MB and can be constructed from ordinary SRAMs. For most operations, the CPU chip accesses the cache directly in a combinatorial loop by presenting an address and waiting *N* CPU cycles for control, tag, and data to appear, where *N* is a mode-programmable number between 3 and 16 set at power-up time. For writes, both the total number of cycles and the

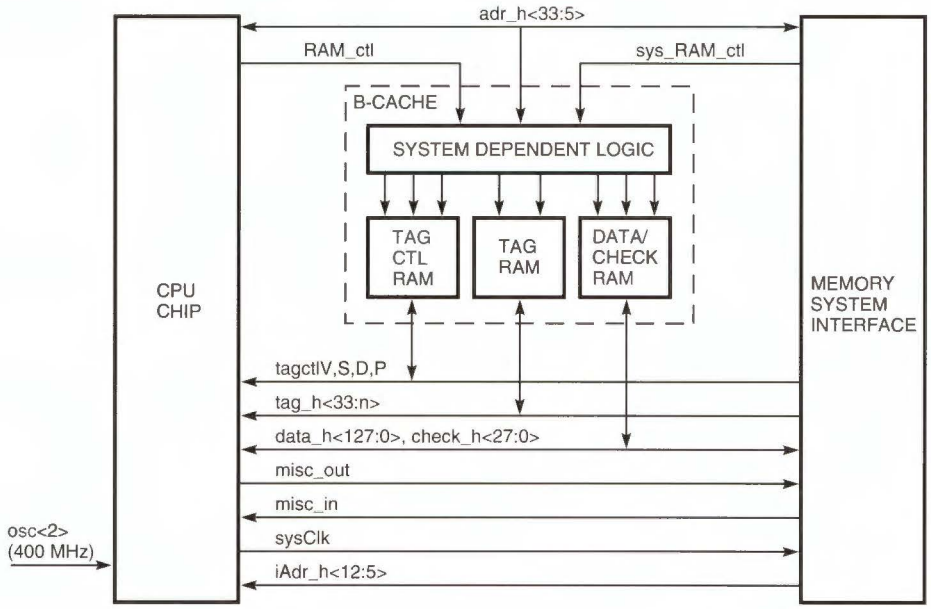



*Figure 4 CPU External Interface*

duration and position of the write signal are programmable in units of CPU cycles. This allows the module designer to select the size and access time of the SRAMs to match the desired price/performance point.

The interface is designed to allow all cache policy decisions to be controlled by logic external to the CPU chip. There are three control bits associated with each backup cache (B-cache) line: valid, shared, and dirty. The chip completes a B-cache read as long as valid is true. A write is processed by the CPU only if valid is true and shared is false. When a write is performed, the dirty bit is set to true. In all other cases, the chip defers to an external state machine to complete the transaction. This state machine operates synchronously with the SYS_CLK output of the chip, which is a mode-controlled submultiple of the CPU clock rate ranging from divide by 2 to divide by 8. It is also possible to operate without a backup cache.

As shown in the diagram, the external cache is connected between the CPU chip and the system memory interface. The combinatorial cache access begins with the desired address delivered on the adr_h lines and results in ctl, tag, data, and check bits appearing at the chip receivers within the prescribed access time. In 128-bit mode, B-cache accesses require two external data cycles to transfer the 32-byte cache line across the 16-byte pin bus. In 64-bit mode, it is four cycles. This yields a maximum backup cache read bandwidth of 1.2 gigabytes per second (GB/s) and a write bandwidth of 711MB/s. Internal cache lines can be invalidated at the rate of one line per cycle using the dedicated invalidate address pins, iAdr_h<12:5>.

In the event external intervention is required, a request code is presented by the CPU chip to the external state machine in the time domain of the SYS_CLK as described previously. Figure 5 shows the read miss timing where each cycle is a SYS_CLK cycle. The external transaction starts with the address, the quadword within block and instruction/data indication supplied on the cWMask_h pins, and READ_BLOCK function supplied on the cReq_h pins. The external logic returns the first 16 bytes of data on the data_h and error correcting code (ECC) or parity on the check_h pins. The CPU latches the data based on receiving acknowledgment on rdAck_H. The diagram shows a stall cycle (cycle 4) between the request and the return data; this depends on the external logic and could range from zero to many cycles. The second 16 bytes of data are returned in the same way with rdAck_h signaling the return of the data and cAck_h signaling the completion of the transaction. cReq_h returns to idle and a new transaction can start at this time.

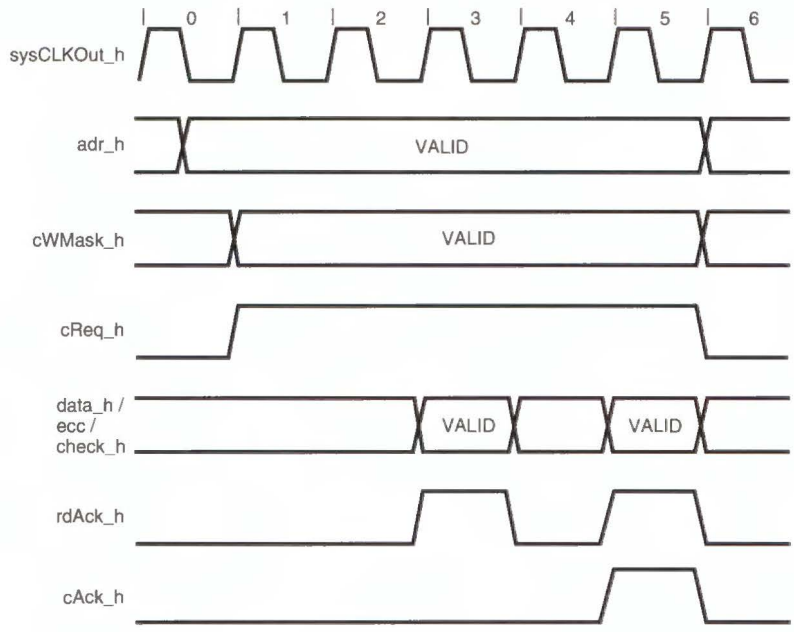



*Figure 5 CPU External Timing*

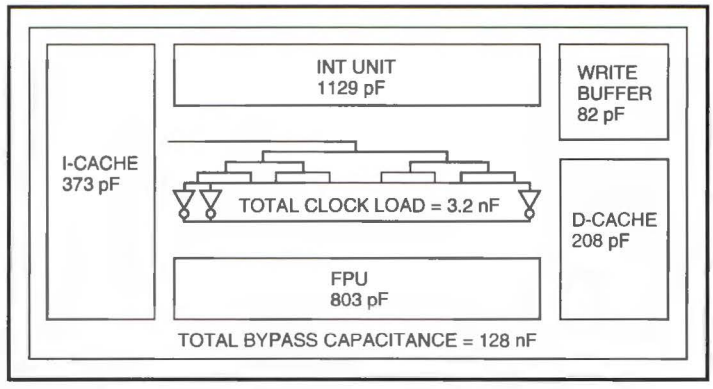
The chip implements a novel set of features supporting chip and module test. When the chip is reset, the first action is to read from a serial read-only memory (SROM) into the I-cache via a private three-wire port. The CPU is then enabled and the program counter (PC) is forced to 0. Thus with only three functional components (CPU chip, SROM, and clock input), a system is able to begin executing instructions. This initial set of instructions is used to write the bus control registers inside the CPU chip to set the cache timing and to test the chip and module from the CPU out. After the SROM loads the I-cache, the pins used for the SROM interface are enabled as serial in and out ports. These ports can be used to load more data or to return status of testing and setup.

## Circuit Implementation

Many novel circuit structures and detailed analysis techniques were developed to support the clock rate in conjunction with the complexity demanded by the concurrence and wide data paths. The clocking method is single wire level sensitive. The bus interface unit operates from a buffered version of the main clock. Signals that cross this interface are deskewed to eliminate races. This clocking method eliminates dead time between phases and requires only a single clock signal to be routed throughout the chip.

One difficulty inherent in this clocking method is the substantial load on the clock node, 3.25 nanofarad (nF) in our design. This load and the requirement for a fast clock edge led us to take particular care with clock routing and to do extensive analysis on the resulting grid. Figure 6 shows the distribution of clock load among the major functional units. The clock drives into a grid of vertical metal 3 and horizontal metal 2. Most of the loading occurs in the integer and floating-point units that are fed from the more robust metal 3 lines. To ensure the integrity of the clock grid across the chip, the grid was extracted from the layout and the resulting network, which contained 630,000 RC elements, was simulated using a circuit simulation program based on the AWEsim simulator from Carnegie-Mellon University. Figure 7 shows a three-dimensional representation of the output of this simulation and shows the clock delay from the driver to each of the 63,000 transistor gates connected to the clock grid.

The 200-MHz clock signal is fed to the driver through a binary fanning tree with five levels of buffering. There is a horizontal shorting bar at the input to the clock driver to help smooth out possible asymmetry in the incoming wave front. The driver itself consists of 145 separate elements, each of which contains four levels of prescaling into a final output stage that drives the clock grid.




Note: Total effective switching capacitance = 12.5 nF

*Figure 6 Clock Load Distribution*

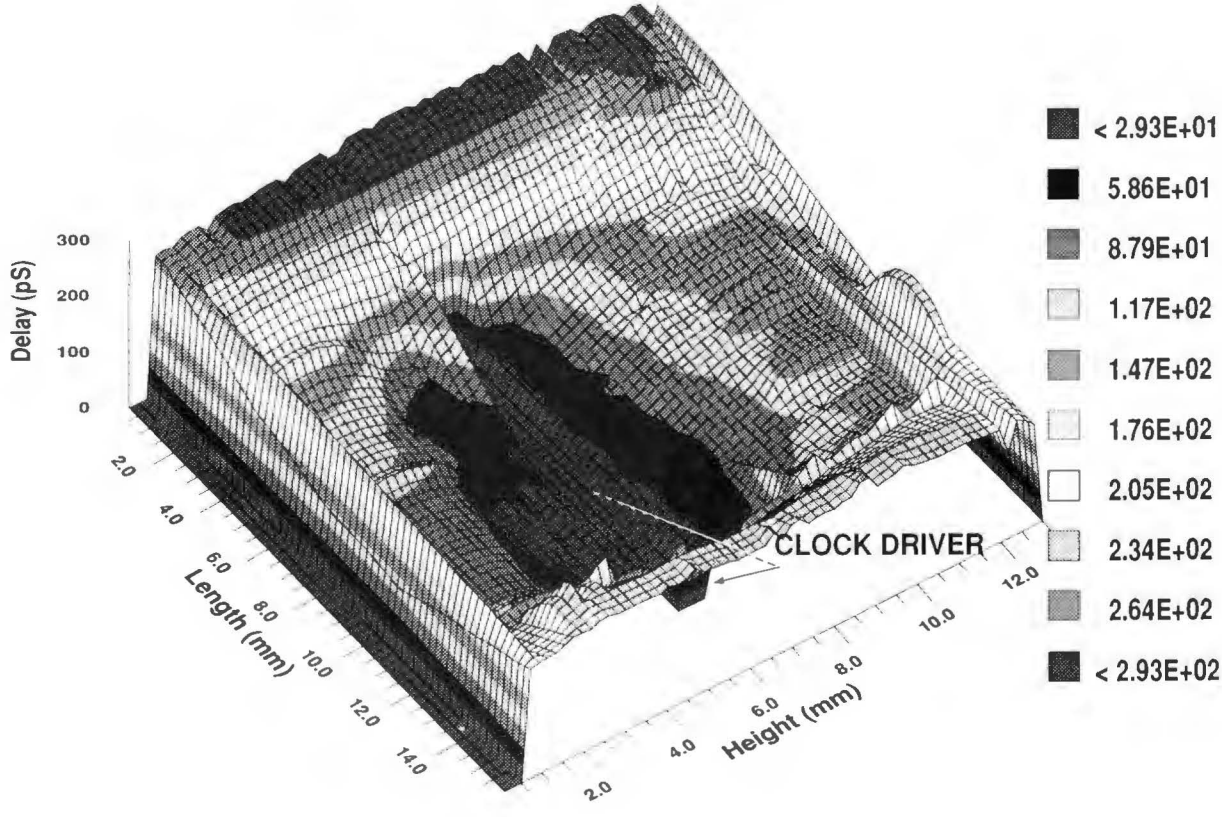


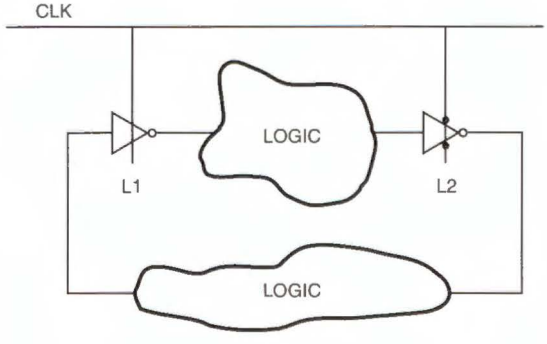

*Figure 7 CPU Clock Skew*

The clock driver and predriver represent about 40 percent of the total effective switching capacitance determined by power measurement to be 12.5 nF (worst case including output pins). To manage the problem of *di/dt* on the chip power pins, explicit decoupling capacitance is provided on-chip. This consists of thin oxide capacitance that is distributed around the chip, primarily under the data buses. In addition, there are horizontal metal 2 power and clock shorting straps adjacent to the clock generator, and the thin oxide decoupling cap under these lines supplies charge to

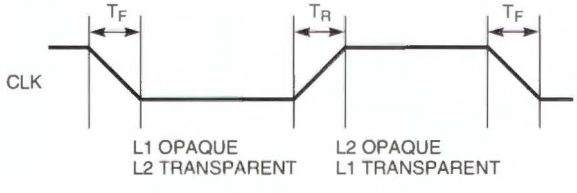
the clock driver. *di/dt* for the driver alone is about $2 \times 10^{11}$ amperes per second. The total decoupling capacitance as extracted from the layout measures 128 nF. Thus the ratio of decoupling capacitance to switching cap is about 10:1. With this capacitance ratio, the decoupling cap could supply all the charge associated with a complete CPU cycle with only a 10 percent reduction in the on-chip supply voltage.

## *Latches*

As previously described, the chip employs a single-phase approach, with nearly all latches in the core of the chip receiving the clock node, CLK, directly. A representative example is illustrated in Figure 8. Notice that L1 and L2 are transparent latches separated by random logic and are not simultaneously active; L1 is active when CLK is high and L2 is active when CLK is low. The minimum number of delays between latches is zero and the maximum number of delays is constrained only by the cycle time and the details of any relevant critical paths. The bus interface unit, many data-path structures, and some critical paths deviate from this approach and use buffered versions and/or conditionally buffered versions of CLK. The resulting clock skew is managed or eliminated with special latch structures.




*(a) Latching Schema*




*(b) Latch Timing*

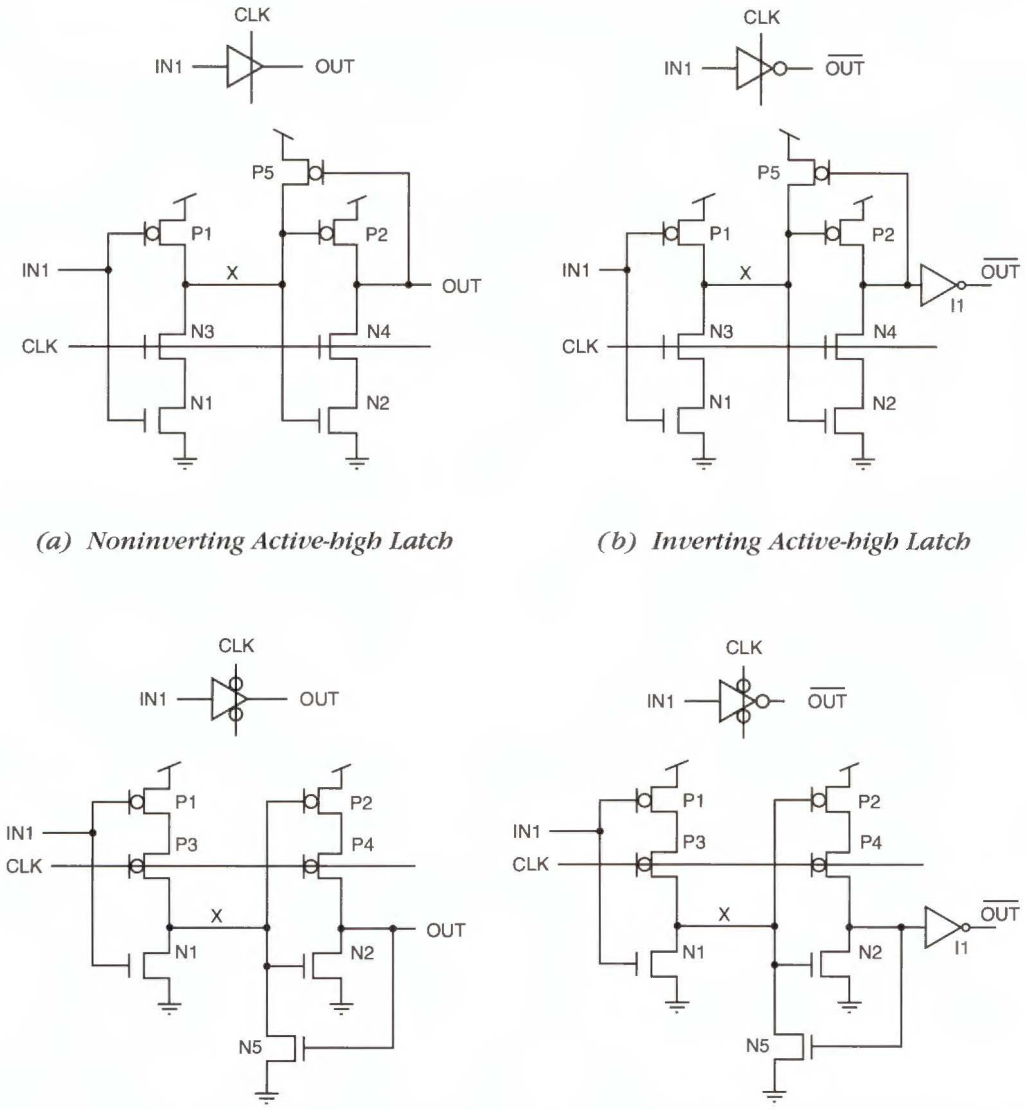
*Figure 8 Chip Latches*

The latches used in the chip can be classified into two categories: custom and standard. The custom latches were used to meet the unique needs of data-path structures and the special constraints of critical paths. The standard latches were used in the design of noncritical control and in some data-path applications. These latches were designed prior to the start of implementation and were included in the library of usable elements for logic synthesis. All synthesized logic used only this set of latches.

The standard latches are extensions of previously published work, and examples are shown in Figures 9 to 11.[2] To understand the operation of these latches, refer to Figure 9a. When CLK is high, *P*1, *N*1, and *N*3 function as an inverter complementing IN1 to produce *X*. *P*2, *N*2, and *N*4 function as a second inverter and complement *X* to produce OUT. Therefore, the structure passes IN1 to OUT. When CLK is low, *N*3 and *N*4 are cut off. If IN1, *X*, and OUT are initially high, low, and high respectively, a transition of IN1 *falling* pulls *X* high through *P*1 causing *P*2 to cut off, which tristates OUT high. If IN1, *X*, and OUT are initially low, high, and low respectively, a transition of IN1 *rising* causes *P*1 to cut off, which tristates *X* high leaving out tristated low. In both cases, additional transitions of IN1 leave *X* tristated or driven high with OUT tristated to its initial value. Therefore, the structure implements a latch that is transparent when CLK is high and opaque when CLK is low. Figure 9c shows the dual circuit of the latch just discussed; this structure implements a latch that is transparent when CLK is low and opaque when CLK is high. Figures 9b and 9d depict latches with an output buffer used to protect the sometimes dynamic node OUT and to drive large loads.

The design of the standard latches stressed three primary goals: flexibility, immunity to noise, and immunity to race-through. To achieve the desired flexibility, a variety of latches like those in Figures 9 to 11 in a variety of sizes were characterized for the implementors. Thus the designer could select a latch with an optional output buffer and an embedded logic function that was sized appropriately to drive various loads. Furthermore, it was decided to allow zero delay between latches, completely freeing the designer from race-through considerations when designing static logic with these latches.

In the circuit methodology adopted for the implementation, only one node, *X* (Figure 9a), poses




*(a) Noninverting Active-high Latch*

*(b) Inverting Active-high Latch*

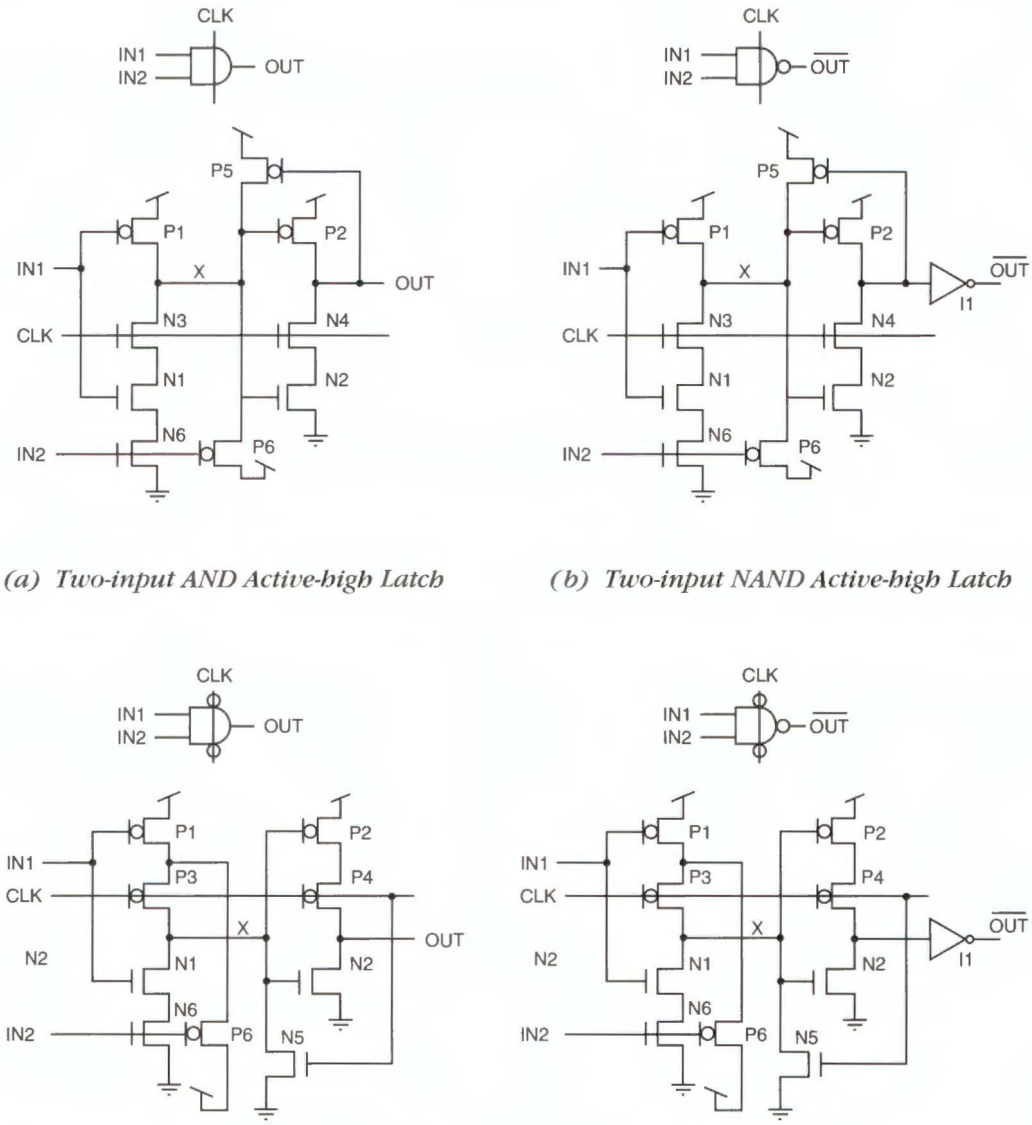
*(c) Noninverting Active-low Latch*

*(d) Inverting Active-low Latch*

*Figure 9 Basic Latches*

inordinate noise margin risk. As noted above, *X* may be tristated high with OUT tristated low when the latch is opaque. This maps into a dynamic node driving into a dynamic gate that is very sensitive to noise that reduces the voltage on *X*, causing leakage through *P*2, thereby destroying OUT. This problem was addressed by the addition of *P*5. This weak feedback device is sized to source enough current to counter reasonable noise and hold *P*2 in cutoff. *N*5 plays an analogous role in Figure 9c.

Race-through was the major functional concern with the latch design. It is aggravated by clock skew, the variety of available latches, and the zero delay goal between latches. The clock skew concern was actually the easiest to address. If data propagates in a direction that opposes the propagation of the clock wave front, clock skew is functionally harmless and tends only to reduce the effective cycle time locally. Minimizing this effect is of concern when designing the clock generator. If data propagates in a direction similar to the propagation of the clock wave front, clock skew is a functional concern. This was addressed by radially distributing the clock from the center of the chip. Since




*(a) Two-input AND Active-high Latch*

*(b) Two-input NAND Active-high Latch*

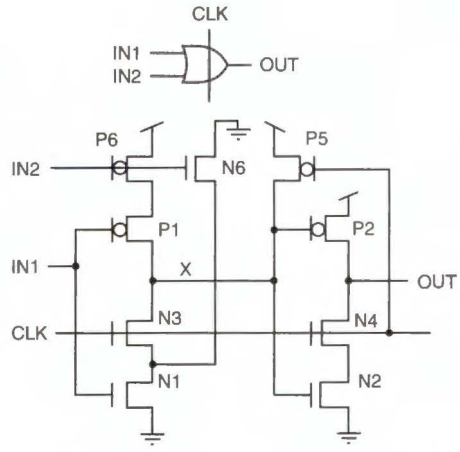
*(c) Two-input AND Active-low Latch*

*(d) Two-input NAND Active-low Latch*

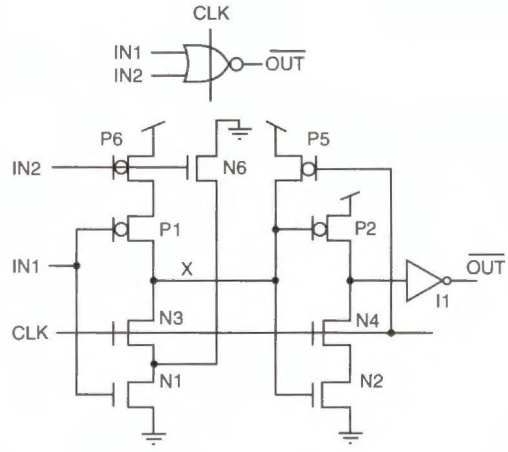
*Figure 10 AND/NAND Latches*

the clock wave front moves out radially from the clock driver toward the periphery of the die, it is not possible for the data to overtake the clock if the clock network is properly designed.

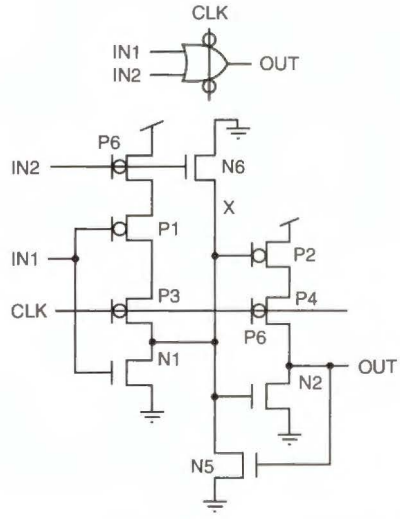
To verify the remaining race-through concerns, a mix-and-match approach was taken. All reasonable combinations of latches were cascaded together and simulated. The simulations were stressed by eliminating all interconnect and diffusion capacitance and by pushing each device into a corner of the process that emphasized race-through. Then many simulations with varying CLK rise and fall times, temperatures, and power supply voltages were performed. The results showed no appreciable evidence of race-through for CLK rise and fall times at or below 0.8 ns. With 1.0-ns rise and fall times, the latches showed signs of failure. To guarantee functionality, CLK was specified and designed to have an edge rate of less than 0.5 ns. This was not a serious constraint since other circuits in the chip required similar edge rates of the clock.



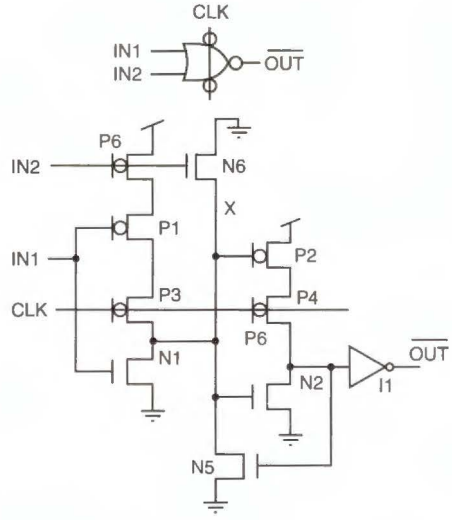

*(a) Two-input OR Active-high Latch*




*(b) Two-input NOR Active-high Latch*




*(c) Two-input OR Active-low Latch*




*(d) Two-input NOR Active-low Latch*

*Figure 11 OR/NOR Latches*

A last design issue worth noting is the feedback devices, *N*5 and *P*5, in Figures 10c, 10d, 11a, and 11b. Notice that these devices have their gates tied to CLK instead of OUT like the other latches. This difference is required to account for an effect not present in the other latches. In these latches, a stack of devices is connected to node *X*, without passing through the clocked transistors *P*3 or *N*3. Referring to Figure 11a, assume CLK is low, *X* is high, and OUT is low. If multiple random transitions are allowed by IN1 with IN2 high, then coupling through *P*1 can drive *X* down by more than a threshold even with weak feedback, thereby destroying OUT. To counter this phenomenon, *P*5 cannot be a weak feedback device and therefore cannot be tied to OUT if the latch is to function properly when CLK is high. Note that taller stacks aggravate this problem because the devices become larger and there are more devices to participate in coupling. For this reason, stacks in these latches were limited to three high. Also, note that clocking *P*5 introduces another race-through path

since $X$ will unconditionally go high with CLK falling, and OUT must be able to retain a stored ONE. So there is a two-sided constraint: $P5$ must be large enough to counter coupling and small enough not to cause race-through. These trade-offs were analyzed by simulation in a manner similar to the one outlined above.

## *64-bit Adder*

A difficult circuit problem was the 64-bit adder portion of the integer and floating-point ALUs. Unlike a previous high-speed design, we set a goal to achieve single-cycle latency in this unit.[3] Figure 12 has an organizational diagram of its structure. Every path through the adder includes two latches, allowing fully pipelined operation. The result latches are shown explicitly in the diagram; however, the input latches are somewhat implicit, taking advantage of the predischarge characteristics of the carry chains. The complete adder is a combination of three methods for producing a binary add: a byte long carry chain, a longword (32-bit) carry select, and local logarithmic carry select.[4] The carry select is built as a set of n-channel metal-oxide semiconductor (NMOS) switches that direct the data from byte carry chains. The 32-bit longword lookahead is implemented as a distributed differential circuit controlling the final stage of the upper longword switches. The carry chains are organized in groups of eight bits.

Carry chain width was chosen to implement a byte compare function specified by the architecture. The carry chain implemented with NMOS transistors is shown in Figure 13a. Operation begins with the chain predischarged to $V_{SS}$, with the controlling signal an OR of CLK and the kill function. Evaluation begins along the chain length without the delay associated with the $V_{gs}$-$V_t$ threshold found in a chain precharged to $V_{DD}$. An alternative to a predischarged state was to precharge to $V_{DD}$-$V_t$, but the resulting low noise margins were deemed unacceptable. From the least significant bit to the most significant bit, the width of the NMOS gates for each carry chain stage is tapered down, reducing the loading presented by the remainder of the chain. The local carry nodes are received by ratioed inverters. Each set of propagate, kill, and generate signals controls two carry chains, one that assumes a carry in and one that assumes no carry in. The results feed the bit-wise data switches as well as the carry selects.

The longword carry select is built as a distributed cascode structure used to combine the byte generate, kill, and propagate signals across the lower 32-bit longword. It controls the final data selection into the upper longword output latch and is out of the critical path.

The NMOS byte carry select switches are controlled by a cascade of closest neighbor byte carry outs. Data in the most significant byte of the upper longword is switched first by the carry-out data of the next lower byte, byte 6, then by byte 5, and finally byte 4. The switches direct the sum data from either the carry-in channel or the no-carry channel (Figure 13b). Sign extension is accomplished by disabling the upper longword switch controls on longword operations and forcing the sign of the result into both data channels.

## *I/O Circuitry*

To provide maximum flexibility in applications, the external interface allows for several different modes of operation all using common on-chip circuitry. This includes choice of logic family (CMOS/transistor-transistor logic [TTL] or emitter-coupled logic [ECL]) as well as bus width (64/128 bits), external cache size and access time, and BIU clock rate. These parameters are set into mode registers during chip power-up. The logic family choice provided an interesting circuit challenge. The input receivers are differential amplifiers that utilize an external reference level which is set to the switching midpoint of the external logic family. To maintain signal integrity of this reference voltage, it is resistively isolated and RC-filtered at each receiver.

The output driver presented a more difficult problem due to the 3.3-V $V_{DD}$ chip power supply. To provide a good interface to ECL, it is important that the output driver pull to the $V_{DD}$ rail (for ECL operation $V_{DD}$ = 0 V, $V_{SS}$ = -3.3 V). This precludes using NMOS pull-ups. P-channel metal-oxide semiconductor (PMOS) pull-ups have the problem of well-junction forward bias and PMOS turn-on when bidirectional outputs are connected to 5-V logic in CMOS/TTL mode. The solution, as shown in Figure 14, is a unique floating-well driver circuit that avoids the cost of series PMOS pull-ups in the final stage, while providing direct interface to 5-V CMOS/TTL as well as ECL.[5]

Transistors $Q1$, $Q2$, and $Q6$ are the actual output devices. $Q1$ and $Q2$ are NMOS devices arranged in cascode fashion to limit the voltages across a single

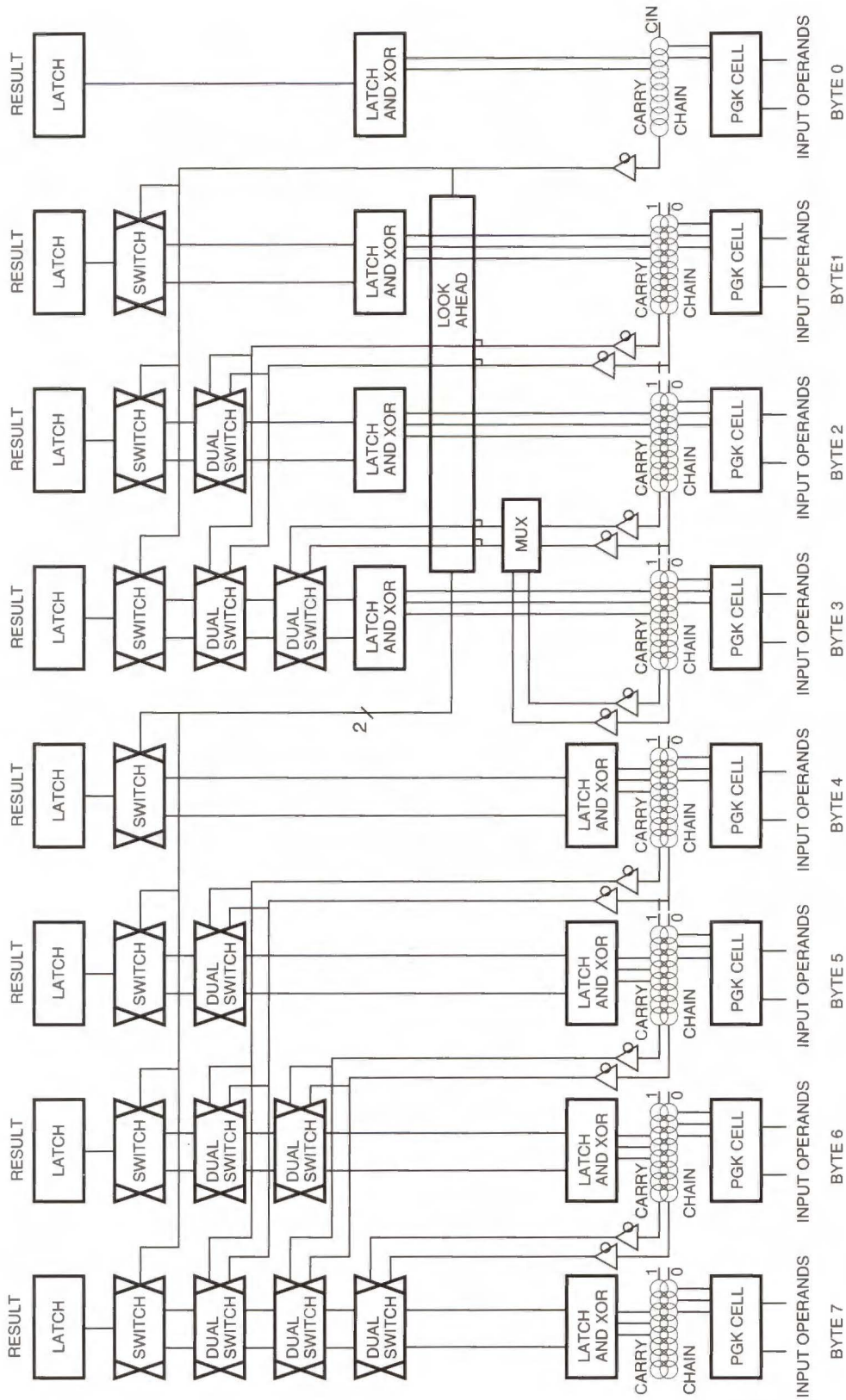
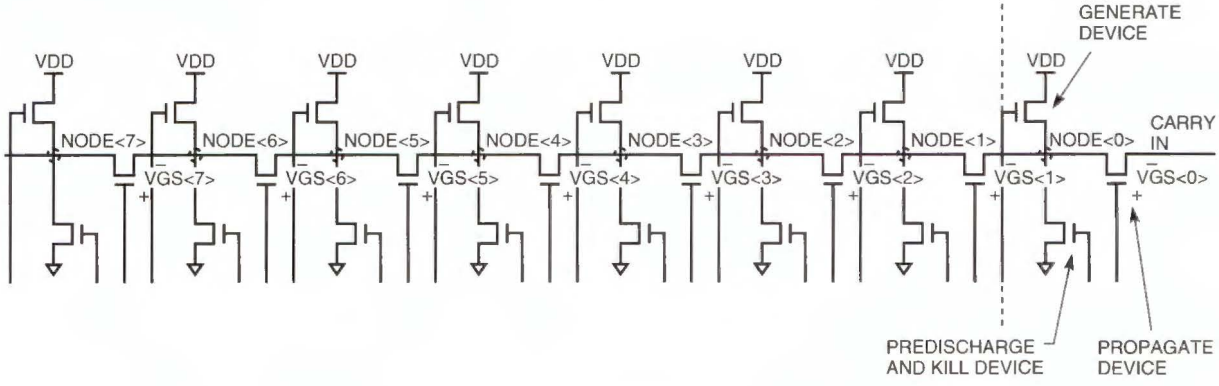
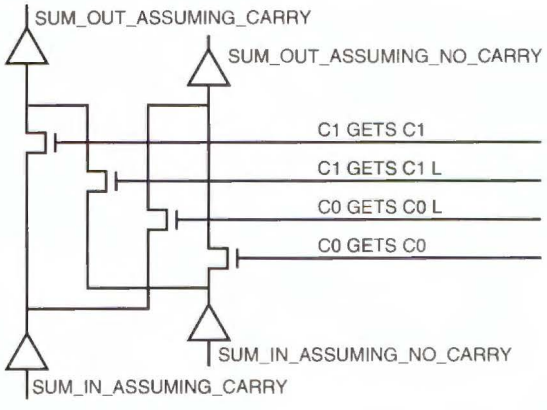



*Figure 12 64-bit Adder Block Diagram*




*(a) Adder Carry Chain*




*(b) Adder Carry-select Switches*

*Figure 13 Adder Carry*

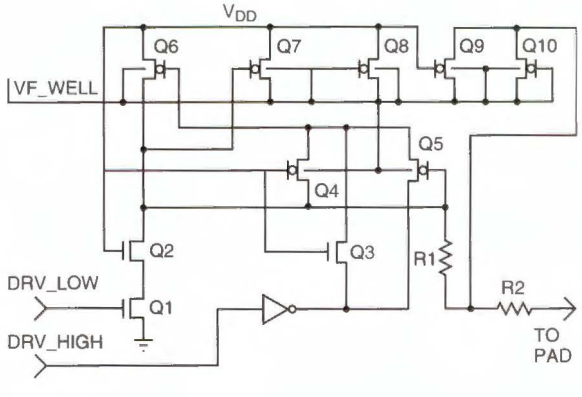



*Figure 14 Floating-well Driver*

transistor to no more than 4 V. $Q6$ is a PMOS pull-up device that shares a common n-well with $Q7$ through $Q10$, which have responsibility for supplying the well with a positive bias voltage of either $V_{DD}$ or the I/O pin potential, whichever is higher. $Q3$ through $Q5$ control the source of voltage for the gate of $Q6$—either the output of the inverter or the I/O pad if it moves above $V_{DD}$. $R1$ and $R2$ provide 50-ohm series termination in either operating mode.

## Caches

The two internal caches are almost identical in construction. Each stores up to 8KB of data (D-cache) or instruction (I-cache) with a cache block size of 32 bytes. The caches are direct mapped to realize a single cycle access, and can be accessed using

untranslated bits of the virtual address since the page size is also 8KB. For a read, the address stored in the tag and a 64-bit quadword of data are accessed from the caches and sent to either the memory management unit for the D-cache or the instruction unit for the I-cache. A write-through protocol is used for the D-cache.

The D-cache incorporates a pending fill latch that accumulates fill data for a cache block while the D-cache services other load/store requests. Once the pending fill latch is full, an entire cache block can be written into the cache on the next available cycle. The I-cache has a similar facility called the stream buffer. On an I-cache miss, the I-box fetches the required cache block from memory and loads it into the I-cache. In addition, the I-box will prefetch the next cache block and place it in the stream buffer. The data is held in the stream buffer and is written into the I-cache only if the data is requested by the I-box.

Each cache is organized into four banks to reduce power consumption and current transients during precharge. Each array is approximately 1,024 cells wide by 66 cells tall with the top two rows used as redundant elements. A six-transistor, 98-$\mu m^2$ static RAM cell is used. The cell utilizes a local interconnect layer that connects between polysilicon and active area, resulting in a 20-percent reduction in cell area compared to a conventional six-transistor cell. A segmented word line is used to accommodate the banked design, with a global word line implemented in third-level metal and a local word line implemented in first-metal layer. The global word line feeds into local decoders that decode the lower two bits of the address to generate the local word lines. As shown in Figure 15, the word lines are enabled while the clock is high, and the sense amplifiers are fired on the falling edge of the clock.

## Summary

A single chip microprocessor that implements a new 64-bit high-performance architecture has been described. By using a highly optimized design style in conjunction with a high-performance 0.75-$\mu m$ technology, operating speeds up to 200 MHz have been achieved.

The chip is superscalar degree 2 and has 7- and 10-stage pipelines for integer and floating-point instructions. The chip includes primary instruction and data caches, each 8KB in size. In each 5-ns cycle, the chip can issue two instructions to two of four units, yielding a peak execution rate of 400 mips and 200 MFLOPS.

The chip is designed with a flexible external interface providing integral support for a secondary cache constructed of ordinary SRAMs. The interface is fully compatible with virtually any multiprocessor write cache coherence scheme,

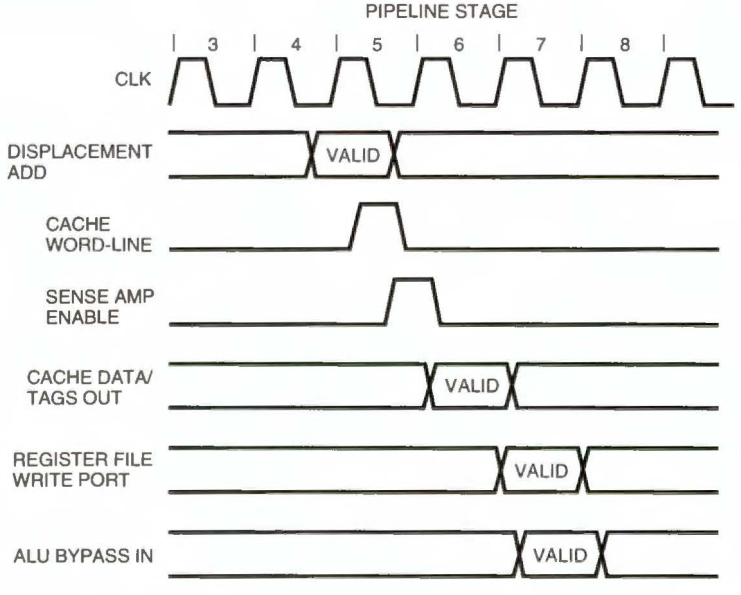



*Figure 15 D-cache Timing Diagram*

and can accommodate a wide range of timing parameters. It can interface directly to standard TTL and CMOS as well as 100K ECL technology.

## References


1. *Alpha Architecture Handbook* (Maynard: Digital Equipment Corporation, Order No. EC-H1689-10, 1992).
2. J. Yuan and C. Svensson, "High-Speed CMOS Circuit Techniques," *IEEE Journal of Solid-State Circuits,* vol. 24, no. 1 (February 1989).
3. R. Conrad et al., "A 50 MIPS (peak) 32/64b Microprocessor," *ISSCC Digest of Technical Papers* (February 1989): 76–77.
4. J. Sklansky, "Conditional-Sum Addition Logic," *IRE Transactions on Electronic Computing,* vol. EC-9 (1960): 226–231.
5. H. Lee et al., "An Experimental 1Mb CMOS SRAM with Configurable Organization and Operation," *ISSCC Digest of Technical Papers* (February 1988): 180–181.

*Charles P. Thacker*
*David G. Conroy*
*Lawrence C. Stewart*

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

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

> There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things.
>
> —Niccolo Machiavelli, *The Prince*

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

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

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

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

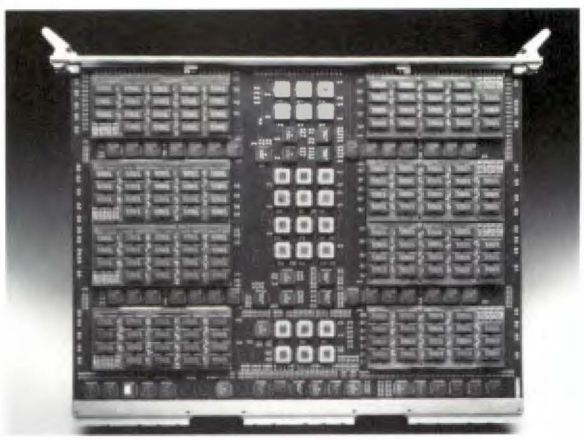
The project team was deliberately kept small. Two designers were located at SRC, one was with the Semiconductor Engineering Group's Advanced Development Group in Hudson, MA, and one was a member of Digital's Cambridge Research Laboratory in Cambridge, MA. Although the project team was separated both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

Since multiprocessing capability is central to the Alpha AXP architecture, we decided that the ADU had to be a multiprocessor. We chose to implement a bus-based memory coherence protocol. A high-speed bus connects three types of modules: The CPU module contains one microprocessor chip, its external cache, and an interface to the bus. A storage module contains two 32-megabyte (MB) interleaved banks of dynamic random-access memory (DRAM). The I/O module contains two 50MB per second (MB/s) I/O channels that are connected to one or two DECstation 5000 workstations, which provide disk and network I/O as well as a high-performance debugging environment. Most of the logic, with the exception of the CPU chip, is emitter-coupled logic (ECL), which we selected for its high speed and predictable electrical characteristics. Modules plug into a 14-slot card cage. The card cage and power supplies are housed in a 0.5-meter (m) by 1.1-m cabinet. A fully loaded cabinet dissipates approximately 4,000 watts and is cooled by forced air. Figures 1 and 2 are photographs of the system and the modules.

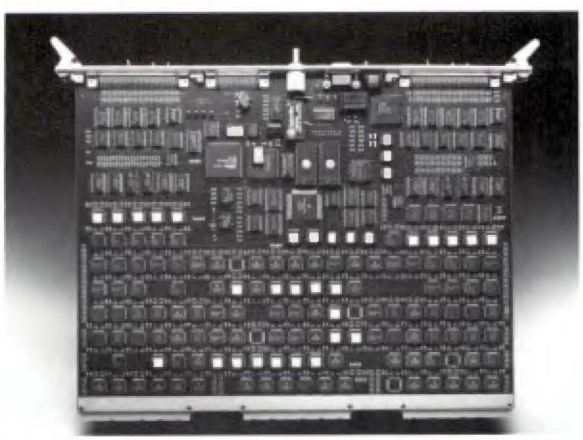
In the remaining sections of this paper, we discuss the backplane interconnect and cache coherence protocol used in the ADU. We then describe the system modules and discuss the design choices. We also present some of the uses we have found for the ADU in addition to its original purpose as a software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

*Figure 1 The Alpha Demonstration Unit*

*(a) CPU Module*



*(b) Storage Module*



*(c) I/O Module*

*Figure 2 ADU Modules*

### Backplane Interconnect

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

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

We designed the bus to provide high bandwidth, which is suitable for a multiprocessor system, and to offer minimal latency. As the CPU cycle time becomes very small, 5 nanoseconds (ns) for the DECchip 21064 chip, the main memory latency becomes an important component of system performance. The ADU bus can supply 320MB/s of user data, but still is able to satisfy a cache read miss in just 200 ns.

### Bus Signals

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

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

**Table 1 Bus Signals**

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

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

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

### Addressing

The bus supports a physical address space of 64 gigabytes ($2^{36}$ bytes). The resolution of a bus address is a 32-byte cache block, which is the only unit of transfer supported; consequently, 31 address bits suffice. One-quarter of the address space is reserved for control registers rather than storage.

Accesses to this region are treated specially: CPUs do not store data from this region in their caches, and the target need not supply correct ECC bits.

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

The geographic selection of the target means that a particular subnode only needs to compare the requested slot and subnode bits with its own slot and subnode numbers to decide whether it is the target. This reduces the time required for the decision compared to a scheme in which the target inspects the address field, but it means that each initiator must maintain a mapping between physical addresses and slot and subnode numbers. This mapping is performed by a RAM in each initiator. For CPU modules, the RAM lookup does not reduce performance, since the access is done in parallel with the access of the module's secondary cache. The slot-mapping RAMs in each initiator are loaded at system initialization time by the configuration process described previously.

## *Bus Operation*

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

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

An arbitration cycle is followed by a request cycle. The initiator places an address, node and subnode numbers, and a command on the bus. There are only three commands. A read command requests a 32-byte cache block from memory. The target memory or a cache that contains a more recent copy supplies the data after a five-cycle delay. A write command transmits a 32-byte block to memory, using the same cycles for the data transfer as the read command. Other caches may also take the block and update their contents. A victim write is issued by a CPU module when a block is evicted from the secondary cache. When such an eviction occurs, any other caches that contain the block are guaranteed to contain the same value, so

| CYCLE | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ARB REQUEST | R1 | | | | | R2 | | | | | R3 | | | | | R4 | | | | | | | | | | |
| DATA | | A1 | | | | | A2 | D1 | D1 | D1 | D1 | A3 | D2 | D2 | D2 | D2 | A4 | D3 | D3 | D3 | D3 | | D4 | D4 | D4 | D4 |
| B-SHARED, B-DIRTY | | | | | | S1 | | | | | S2 | | | | | S3 | | | | | S4 | | | | | |
| ERROR | | | | E1 | | | | | E2 | | | | | E3 | | | | | E4 | | | | | | | |

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

*Figure 3 Bus Timing*

they need not participate in the transfer at all. The block is stored in memory, as in a normal write.

## Cache Coherence

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

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

### Coherence Protocol

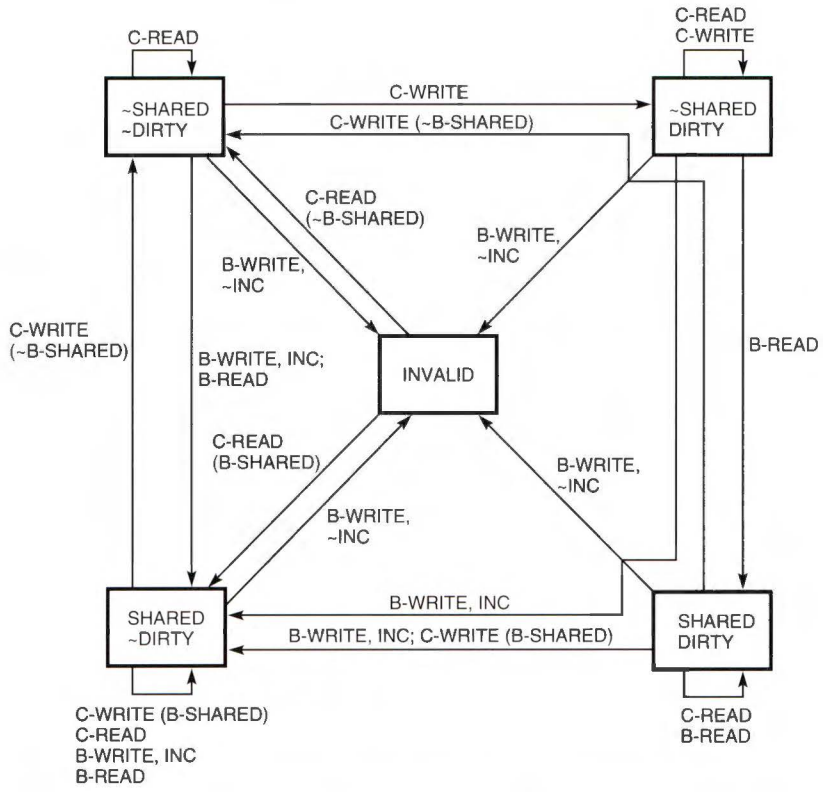
The coherence protocol used in the ADU is a hybrid of an update and an invalidation protocol, and like many hybrids, it combines the good features of both parents. The protocol depends on the fact that the CPU chips contain an on-chip cache backed by a much larger secondary cache that monitors all bus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

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

### Implementation of the Protocol

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

The CPU's on-chip cache uses a write-through strategy. A CPU write to a shared block in the secondary cache initiates a bus write to update the contents of other caches that share the block. Memory is written, so the block becomes clean. If another cache takes the update, it asserts B-shared, and the initiator's state becomes Shared not (~)




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

*Figure 4 Secondary Cache Line States*

Dirty. If no other cache takes the update, either because it does not contain the block or because it decides to invalidate it, then the B-shared signal is not asserted, and the initiator's state becomes ~Shared ~Dirty. The B-shared and B-dirty signals may be asserted by several modules during cycle five of bus operations. The responses are ORed by the open-emitter ECL backplane drivers. More than one cache can contain a block with Shared = true, but only one cache at a time can contain a block with Dirty = true.

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

## System Modules

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

### CPU Module

The ADU CPU module consists of a single CPU chip, a 256-kilobyte (KB) secondary cache, and an interface to the system bus. All CPU modules in the

system are identical. The CPU modules are not self-sufficient; they must be initialized by the console workstation before the CPU can be enabled.

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

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

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

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

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

The data state machine, which operates during bus cycles 6 through 10, moves data to and from the bus and handles the reading and writing of the secondary cache data store. The highly pipelined

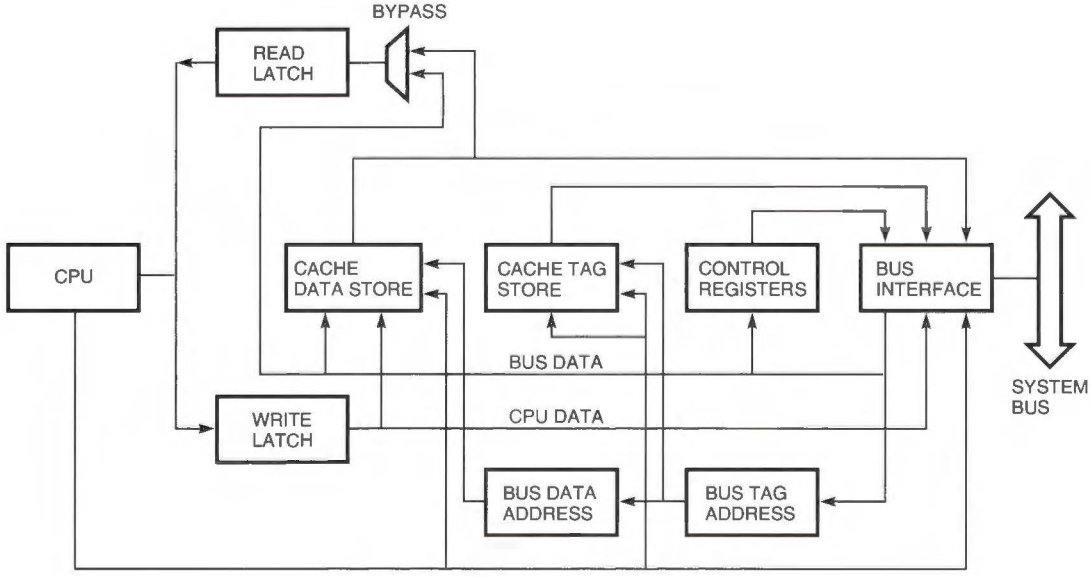



*Figure 5 CPU Module*

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

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

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

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

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

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

| CYCLE | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WRITE CYCLE | W0 | W1 | W2 | W3 | W4 | W5 | W6 | W7 | W8 | W9 | W10 | | | | | |
| READ CYCLE | | | | | | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 |
| CACHE | | | | | | | | | W7 | W8 | W9 | W10 R7 | R8 | R9 | R10 | |

Figure 6a shows a conflict for access to the secondary cache RAMs caused by back-to-back cycles. In the marked cycle, the cache writes the bus data that arrived in cycle W10, but it also needs to read data to supply it during cycle R7.

| CYCLE | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WRITE CYCLE | W0 | W1 | W2 | W3 | W4 | W5 | W6 | W7 | W8 | W9 | W10 | | | | | |
| READ CYCLE | | | | | | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | |
| CACHE EVEN | | | | | | | | | W7 | | W9 | R7 | | R9 | | R10 |
| CACHE ODD | | | | | | | | | | W8 | | W10 | R8 | | R10 | |

Figure 6b shows how this conflict can be resolved by treating the cache as two independent banks (even and odd).

*Figure 6 CPU Timing*

acknowledge signal passes from the bus clock domain to the CPU chip clock domain.

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

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

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

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

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

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

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

Eventually we connected an external pulse generator to the CPU chip's clock and an external power supply to the CPU chip. These modifications permitted us to vary both the operating frequency and the operating voltage of the CPU chip. By using a pulse generator and a power supply that could be remotely controlled by another computer, we were able to write simple programs that could run CPU chip diagnostics, without manual intervention, over a wide range of operating conditions. This greatly simplified the task of collecting the raw data needed by the chip designers to verify the critical paths in the chip.

### *Storage Modules*

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

Each storage module is organized as two independent bus subnodes, so that even in a system with one module, memory is two-way interleaved. Each of the subnodes consists of four banks, each of which stores two longwords of data and their associated error correction bits. With

1-megabit (Mb) RAM chips, the capacity of each module is 64MB. Figure 7 shows the organization of the storage module. The module consists of two independent subnodes, each with four banks of storage. Control signals are pipelined through the banks so that the module can deliver or accept a 64-bit data word (plus ECC) every 20 ns. With the exception of the DRAM interface signals, all signals are ECL levels. The G014 gallium arsenide (GaAs) driver chip improves performance by allowing parallel termination of the DRAM address lines.

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

Since a data rate of 1 word every 20 ns is beyond the capabilities of even the fastest nibble-mode RAMs, we needed an approach that did not require each RAM to provide more than 1 bit per access.

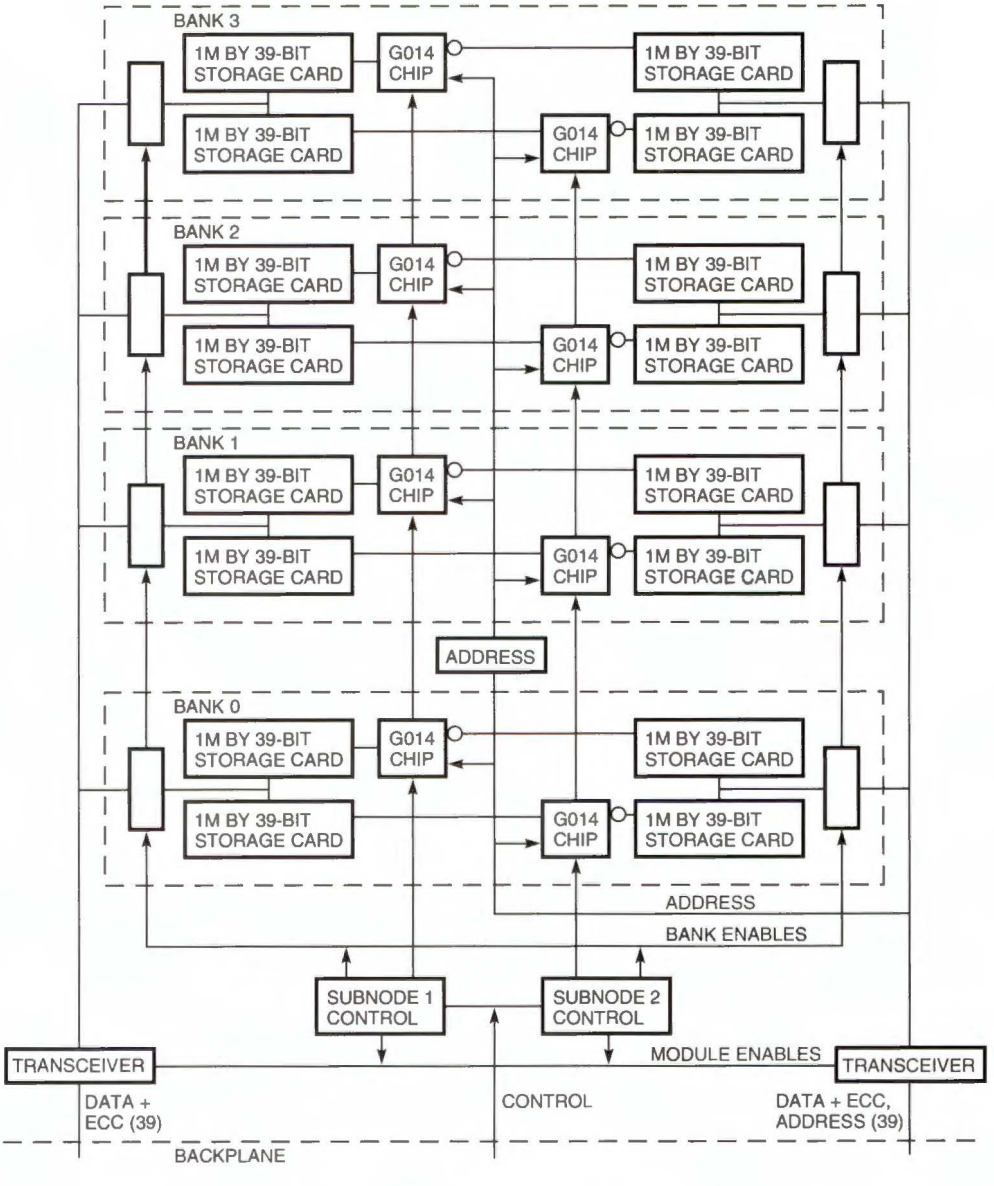



*Figure 7 ADU Storage Module*

We chose to pipeline the four banks of each subnode. Each of the four banks contributes only one 78-bit word to the block. The banks are started sequentially, with a one-cycle delay between each bank.

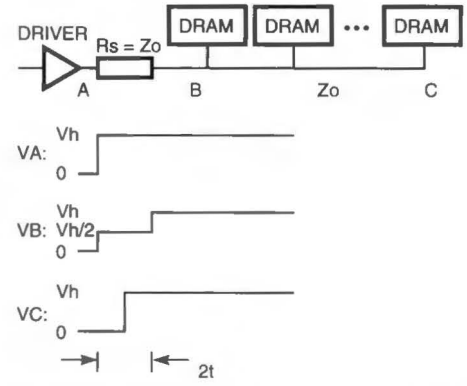
The high performance of the storage module is achieved by maintaining ECL levels and using ECL 100K components wherever possible. The RAM I/O pin levels are converted to ECL levels by latching transceivers associated with each bank. Fortunately, the timing of accesses to the two subnodes of a module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

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

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

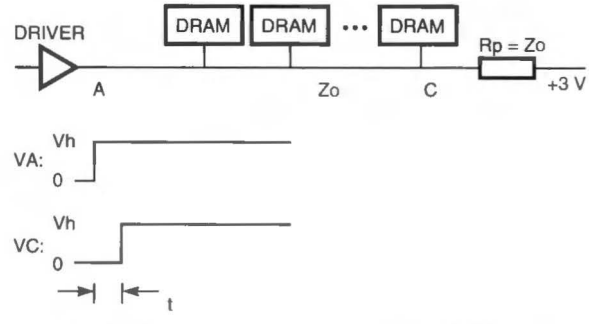
Since low latency was a primary design goal, we chose parallel termination of the RAM address and control lines, as shown in Figure 8b. Each address line is terminated to +3 volts with a series resistor ($R_s$) of 33 ohms, slightly higher than the line impedance. In this configuration, each line's driver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

As shown in Figure 9, each GaAs chip contains a register for eight address bits, row/column address multiplexing and high current drivers for the RAM




Series termination results in a half-amplitude signal at the first RAM on the line until the signal reflects from point C.

*(a) Series Termination*




Parallel termination saves one line transit time, but increases driver current.

*(b) Parallel Termination*

*Figure 8 Address Line Termination*

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

The storage module contains only a small amount of control logic. This logic generates the control signals for the RAMs and the various transceivers that route data from the backplane to each bank. This logic also generates the signals

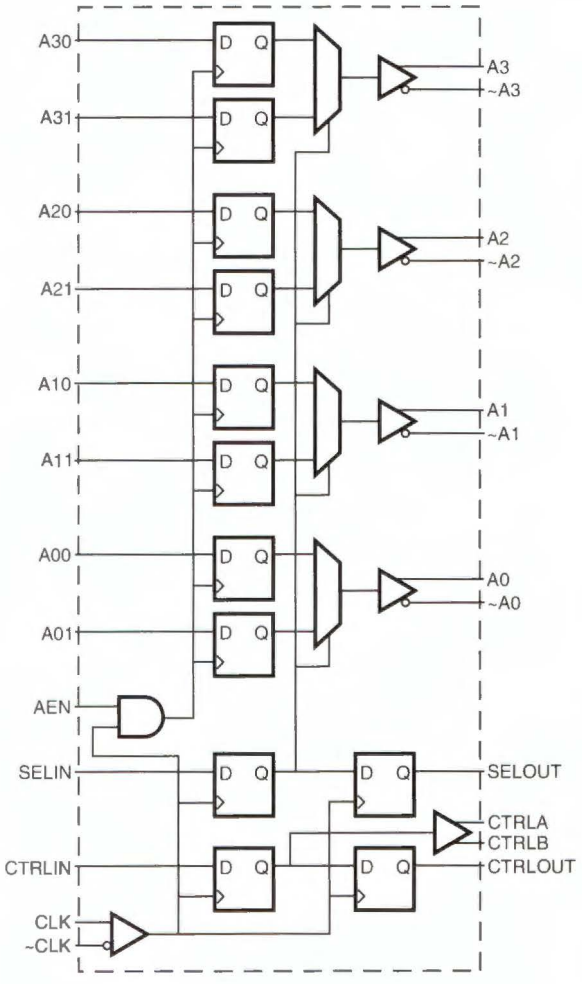



*Figure 9 Address and Control Driver*

needed to refresh the RAMs and to assert the retry signal if another node attempts to access the module while it is refreshing itself.

## *I/O Module*

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

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

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

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

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

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

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

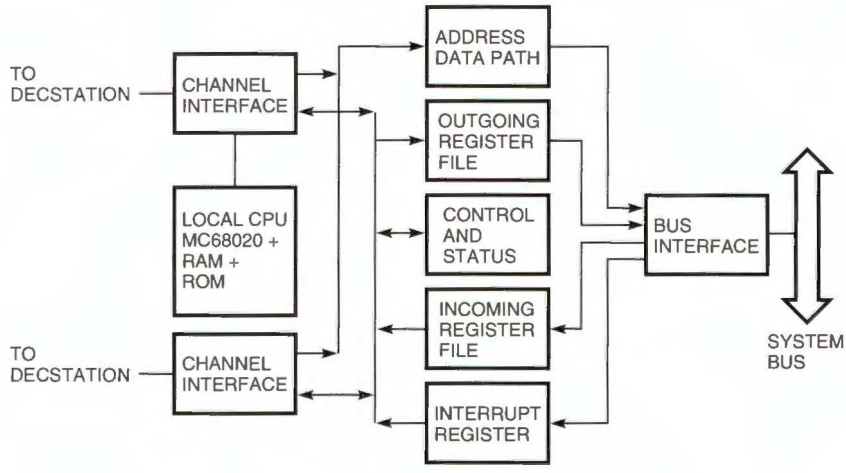
The central features of the I/O module, shown in Figure 10, are two 1K- by 80-bit register files built from 5-ns ECL RAMs. These memories are cycled every 10 ns to simulate dual-ported memories at the 20-ns bus cycle rate. One memory is used as a staging RAM for block transfers from the I/O processors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

On the bus side, the register files are connected directly to the backplane bus transceivers. On the I/O side, the register files are connected to a shared 40-ns bus that connects to the two I/O channels.

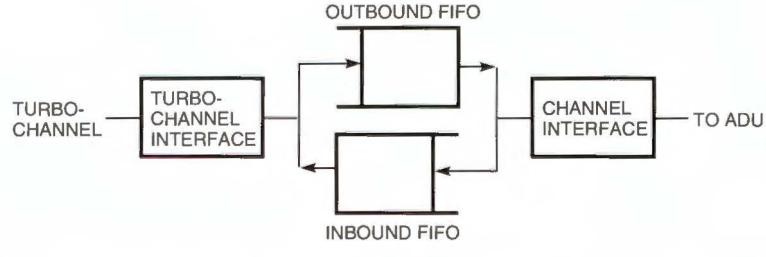
The buses are time-slotted to eliminate the need for arbitration logic. As a consequence, the I/O module control logic is contained in a small number of programmable array logic chips that implement the I/O channel controllers and a block-transfer state machine that handles bus transfers.

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

Each I/O channel interface also has an address counter and a slot-mapping RAM, which are loaded from the workstation. The slot-mapping function sets the correspondence between ADU bus addresses and the geographically addressed storage and CPU modules. The address and slot map for each channel are connected to a common address




*(a) ADU I/O Module*




*(b) TURBOchannel I/O Module*

*Figure 10 I/O Module*

bus. This bus bypasses the register files and directly drives the backplane transceivers during bus address cycles.

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

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

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

### *Packaging*

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

The ADU's modules and card cage are standard 9U (280 millimeter by 367 millimeter) Eurocard components, which are available from a number of vendors. The cabinet is a standard Digital unit, usually used to hold disks. Power supplies are off-the-shelf units. Three supplies are required to provide the 4,000 watts consumed by a system containing a full complement of modules. A standard power conditioner provides line filtering and distributes primary AC to the power supplies. This unit allows the system to operate on 110-volt AC in the United States, or 220-volt AC in Europe.

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

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

## *Conclusions*

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

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

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

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

## Acknowledgments

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

## References

1. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 137–152.

2. C. Thacker, L. Stewart, and E. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers,* vol. 37, no. 8 (August 1988): 909–920.

3. R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon, "Implementing a Cache Consistency Protocol," in *Proceedings of the 12th International Symposium on Computer Architecture* (IEEE, 1985).

4. J. Archibald and L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems,* vol. 4 (November 1986): 273–298.

5. *1991 GaAs IC Data Book and Designer's Guide* (GigaBit Logic, Newbury Park, CA, 1991): 2–39.

*Todd A. Dutton*
*Daniel Eiref*
*Hugh R. Kurth*
*James J. Reisert*
*Robin L. Stewart*

# *The Design of the DEC 3000 AXP Systems, Two High-performance Workstations*

*A family of high-performance 64-bit RISC workstations and servers based on the new Digital Alpha AXP architecture is described. The hardware implementation uses the powerful new DECchip 21064 CPU and employs a sophisticated new system interconnect structure to achieve the necessary high bandwidth and low-latency cache, memory, and I/O buses. The memory subsystem of the high-end DEC 3000 AXP Model 500 provides a 512KB secondary cache and up to 1GB of memory. The I/O subsystem of the Model 500 has integral two-dimensional graphics, SCSI, ISDN, and six TURBOchannel expansion slots.*

The DEC 3000 AXP system family consists of both workstations and servers that are based on Digital's Alpha AXP architecture.[1] The family includes the desktop (DEC 3000 AXP Model 400) and desk-side and rack-mounted (DEC 3000 AXP Model 500) systems. The available operating systems are the DEC OSF/1 AXP and the OpenVMS AXP systems. All systems use the DECchip 21064 microprocessor.[2]

Table 1 gives the specifications for the three DEC 3000 AXP systems.

The DEC 3000 AXP systems are designed to be significantly faster than all previous Digital workstations and to offer performance competitive with that of other reduced instruction set computer (RISC) workstations currently available. In general, RISC systems have larger code sizes and consequently require more instruction-stream bandwidth than complex instruction set computer (CISC) systems. Further, 64-bit machines require more data-stream bandwidth than 32-bit machines. To complement the power of the DECchip 21064 microprocessor, the systems need a balanced system architecture, including a high-bandwidth, low-latency memory system and an efficient, high-performance I/O subsystem.

Traditional workstation designs that use a common system bus exhibit increased memory latency and reduced memory bandwidth due to system bus contention. This is a special concern for designs using a large number of high-performance I/O devices. Increased latency can also result from the additional levels of buffering and system bus loading common to traditional architectures. Many system buses also exhibit multiplexing between address and data, leading to further performance degradation.

To meet the goals of low memory latency, high memory bandwidth, and minimal CPU-I/O memory contention in a cost-competitive manner, the designers implemented the DEC 3000 AXP system architecture in an unusual way. They chose to build the system interconnect from inexpensive application-specific integrated circuits (ASICs), as shown in Figure 1. The ASICs act as a crossbar between the CPU, memory, and I/O buses. Addresses and data are switched independently by the crossbar.

The system block diagram in Figure 2 shows the system architecture of the DEC 3000 AXP systems. The system crossbar in the center of the diagram is composed of six ASICs, consisting of the ADDR ASIC, the TURBOchannel (TC) ASIC, and four SLICE ASICs. The ADDR ASIC switches addresses between the CPU, the memory, and the TC ASIC. The four SLICE ASICs switch data between the CPU, the memory, and the TC ASIC. The TC ASIC switches I/O addresses and data between the ADDR and SLICE ASICs and the TURBOchannel bus. Connected to the TURBOchannel bus are the various I/O controllers,

**Table 1 DEC 3000 AXP Family Specifications**

| Specifications | Desk-side Model 500 | Rack-mount Model 500 | Desktop Model 400 |
|---|---|---|---|
| Height, inches | 24.7 | 15.75 | 5 |
| Width, inches | 12.75 | 17.5 | 20 |
| Depth, inches | 29.7 | 27 | 16.75 |
| Maximum DC power output, watts | 480 | 480 | 295 |
| Memory | | | |
| Standard, MB | 32 | 32 | 32 |
| Maximum, MB | 1024 | 1024 | 512 |
| Internal hard disk | | | |
| Standard, MB | 1050 | 1050 | 426 |
| Maximum, MB | 4200 | 4200 | 2100 |
| Serial ports | 2 | 2 | 2 |
| ISDN port | 1 | 1 | 1 |
| SCSI ports* | 2 | 2 | 2 |
| Ethernet ports† | 2 | 2 | 2 |
| TURBOchannel slots | 6 | 6 | 3 |
| Removable media‡ | 2 | 2 | 1 |
| Integral graphics accelerator | Yes | Yes | No |
| Audio | Yes | Yes | Yes |

Notes:
* One internal and one external.
† AUI (thick wire) and 10Base-T (twisted pair).
‡ 5.25-inch half-height slots.

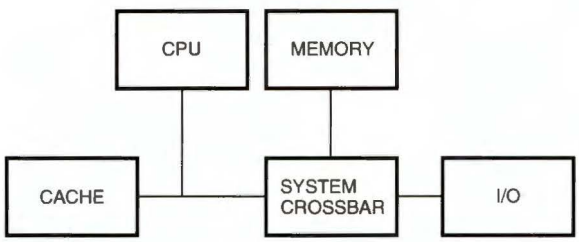



*Figure 1 Simple Crossbar*

including the dual small computer systems interface (SCSI) controller ASIC, the general I/O controller ASIC, and the two-dimensional graphics accelerator ASIC (not present in DEC 3000 AXP Model 400 systems). In addition, six TURBOchannel option slots are available for expansion (three slots in DEC 3000 AXP Model 400 systems).

## CPU Module

The DEC 3000 AXP systems are composed of two primary modules, the CPU module and the I/O module. The CPU module contains the processor, secondary cache, control logic, TURBOchannel interface and, in the Model 500, the two-dimensional graphics subsystem. It has connectors for the I/O module, four memory mother boards, a lights and switches module (LSM), three TURBOchannel options, and the power supply. Figure 3 shows the layout of the module.

### CPU

The DECchip 21064 microprocessor is the CPU of the DEC 3000 AXP systems. On the Model 500, the CPU runs at 150 megahertz (MHz), and on the Model 400, it runs at 133 MHz. The processor is a superscalar, fully pipelined implementation of the Alpha AXP architecture.[2] It contains two on-chip 8-kilobyte (KB) direct-mapped caches, one for use as an instruction cache, the other as a data cache. Both the integer and floating-point units are contained on-chip.

### B-cache Subsystem

The system employs a second-level cache (B-cache) to help minimize the performance penalty of misses and write throughs in the two relatively

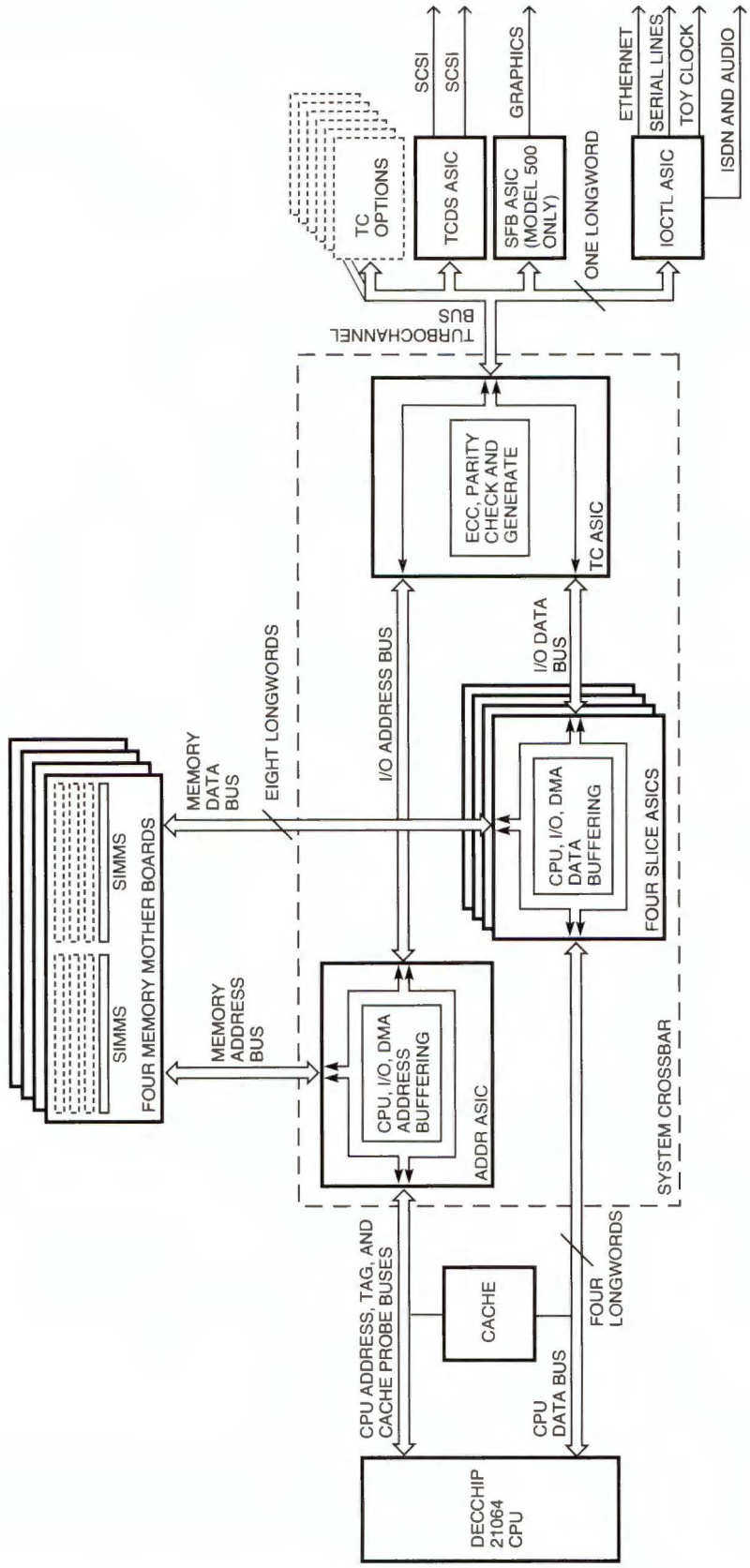



*Figure 2 System Block Diagram*

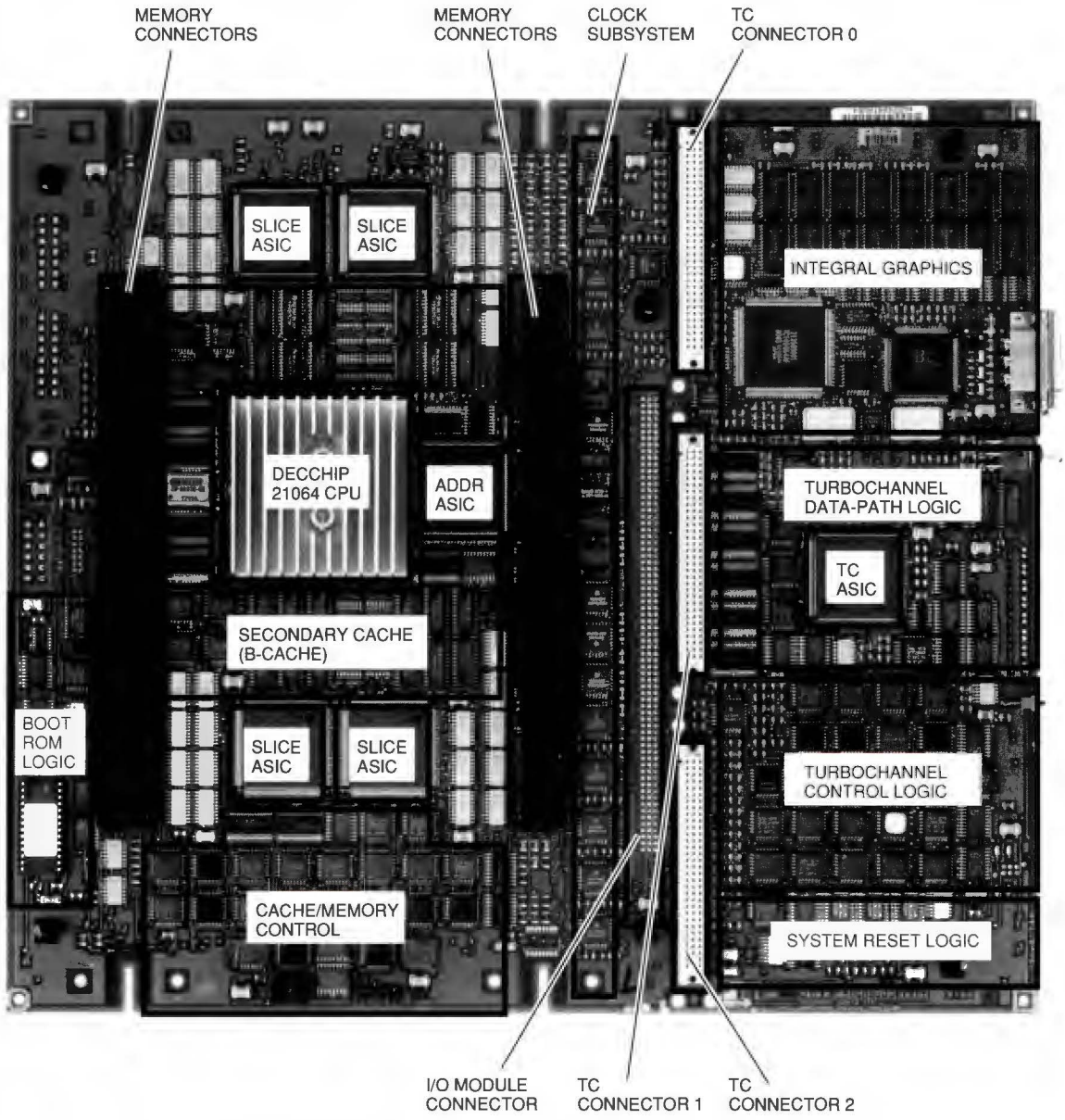



*Figure 3 CPU Module*

small 8KB primary caches of the DECchip 21064 processor. The B-cache is a 512KB, direct-mapped, write-back cache. A direct-mapped cache eliminates the logic needed to choose among the multiple sets of a set-associative cache, resulting in a faster cache cycle time. A write-back protocol was selected because it reduces the amount of write traffic from the B-cache to main memory, leaving more main memory bandwidth available for other memory transactions.

The block size of the B-cache is 32 bytes, matching the block size of the primary caches. The cache block allocation policy used is to allocate on both read miss and write miss. Hardware keeps the cache coherent on direct memory access (DMA) transactions; DMA reads probe the cache and DMA writes update the B-cache (and invalidate the primary data cache).

The DEC 3000 AXP systems are designed to be uniprocessor systems, which simplifies the cache

controller design in a number of ways. For example, since no other CPU's cache can contain a copy of a cache block, there is no need to implement cache coherency constructs such as a shared bit. Further, by loading the B-cache during the power-up sequence and keeping it coherent during DMA by using an always-update protocol, cache blocks in the B-cache are always guaranteed to be valid. This method eliminates stale data problems without needing to use a valid bit.

In addition to the cache memory, the subsystem consists of the cache controller, the main memory controller, and the protocol control logic for memory access arbitration. A block diagram of the CPU and B-cache subsystem is shown in Figure 4.

The B-cache is alternately controlled by the CPU and the external cache controller. When controlled by the CPU, the cache may be read by the CPU in five CPU cycles. The cache data bus width is 16 bytes; therefore two reads are necessary to fill a cache block. The Model 500 has a maximum cache read bandwidth of 480 megabytes per second (MB/s). The cache may be written by the CPU with an initial tag probe latency of five CPU cycles followed by up to two write cycles of five CPU cycles each. The Model 500 has a cache write bandwidth of 320 MB/s.

When a CPU probe misses in the B-cache, or when the CPU accesses the external lock register, control of the cache is turned over to the external cache controller. This logic controls filling the cache with the required data from main memory, handing the data to the CPU during reads, merging CPU write data into the cache on writes, and maintaining the contents of the external cache tag and tag control store. In addition, this logic maintains the architecturally defined lock flag and locked physical address register, which can be used to implement software semaphores and other constructs normally requiring atomic read-modify-write memory transactions.

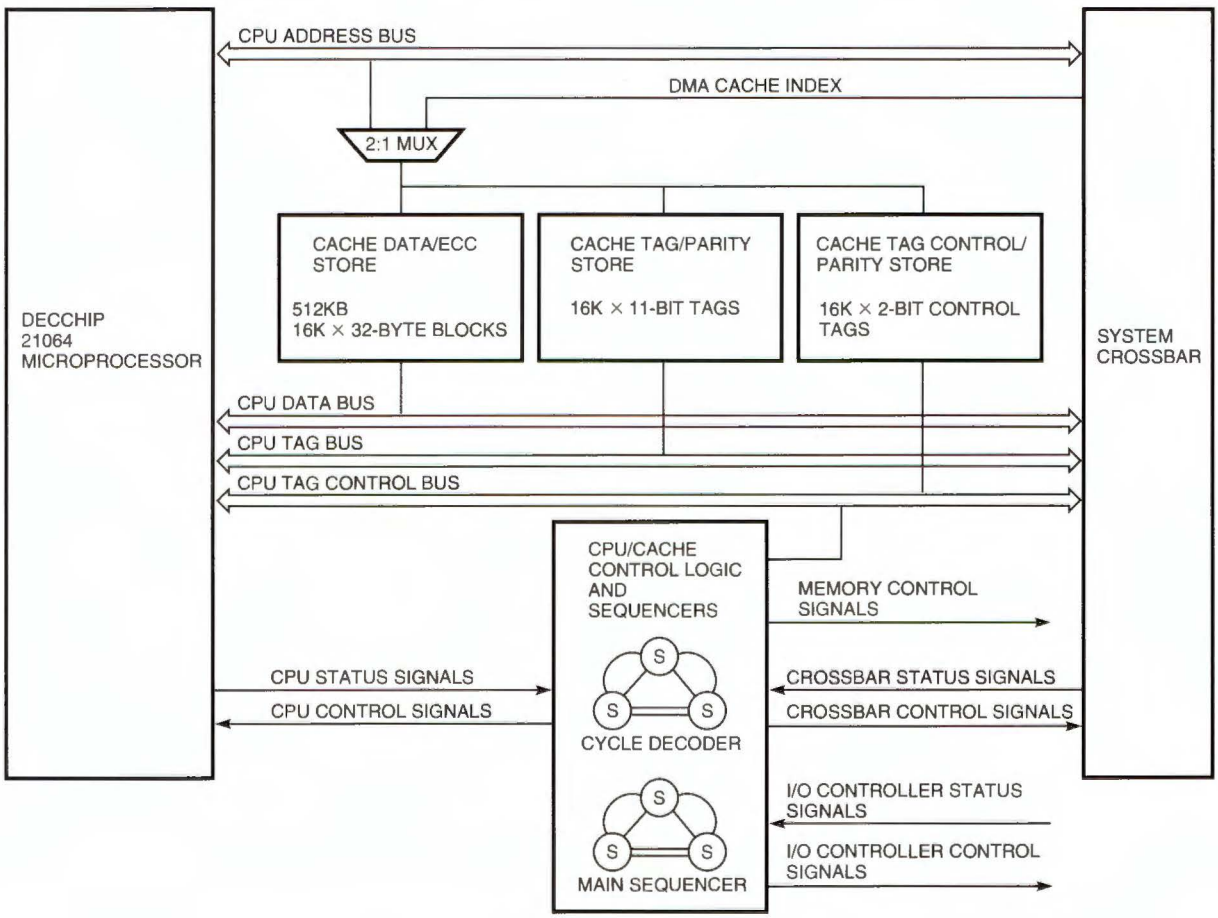



*Figure 4 CPU and B-cache Block Diagram*

The control logic for the B-cache consists of two interlocking state machines. These state machines control arbitration and decoding of processor and I/O subsystem requests. They also generate the control signals needed to execute these requests to the CPU, B-cache, and main memory.

The state machines prioritize and arbitrate requests from various sources, including the CPU, the I/O subsystem, and the memory refresh logic. Arbitration is done according to a fixed priority. First priority goes to DMA requests from the I/O subsystem. Second priority goes to memory refresh requests. Lowest in priority are requests made by the CPU. The one exception to this scheme occurs at the conclusion of a DMA transaction. In this case, the first arbitration cycle following the DMA changes the priority to memory refresh first, CPU request second, and DMA last. This guarantees that requests for CPU and memory refreshes are granted during heavy DMA traffic.

The larger state machine, or main sequencer, examines the command generated by the smaller state machine, or cycle decoder, and initiates the control flow necessary to perform that command. Fifteen unique flows are implemented by the main sequencer. They are

- Read cacheable memory with/without victim block
- Write cacheable memory with/without victim block
- Write noncacheable memory (diagnostic use only)
- Full block write cacheable memory with/without victim block
- Tag space write (diagnostic use only)
- Programmed I/O read/write
- Load lock hit
- Store conditional hit
- Memory refresh
- DMA read/write

When a cache miss occurs and the new cache block replaces a cache block that has been modified, as indicated by the "dirty" status bit, the displaced data is referred to as a "victim block" or "victim data."

The many variants of cacheable reads and writes provide optimized flows that maximize the parallelism of cache accesses and memory accesses. For example, during the "read cacheable memory with victim block" flow, the main sequencer reads the victim block from the B-cache and stores it in the SLICE ASICs in parallel with reading the new block from main memory. The same flow without a victim block makes use of the main memory access time to update the tag store. The control flows for writes to cacheable memory also take advantage of this parallelism. A further write optimization is used when the cycle decoder determines that the entire cache block will be written; in this case the data from memory is completely overwritten, and therefore it is never fetched from memory.

DMA flows are entered upon request of the DMA controller in the I/O control section. DMA control flows start by asserting a "hold request" to the CPU, causing the CPU to cease B-cache operations within a specified time, after which it asserts a "hold acknowledge" signal. It should be noted that the CPU will continue to execute instructions internally until such time as it experiences a miss in one of its internal caches, or it requires some other external cycle.

Each DMA write to memory results in a probe of the B-cache for the DMA target block, with a hit resulting in the B-cache block being updated in parallel with main memory and the corresponding primary data cache block being invalidated. DMA reads cause main memory to be read in parallel with probes and reads of the B-cache. If a cache probe hits, the B-cache data is used to fill the DMA read buffer in the SLICE ASICs; otherwise the main memory data is used. In this manner, cache coherence is maintained.

## Memory System and System Crossbar

The DEC 3000 AXP Model 400 and Model 500 architecture supplants the traditional system bus with a system crossbar constructed from ASICs. Tightly coupled to the crossbar is the system memory. Three types of ASICs—SLICE, ADDR, and TC—form the crossbar. SLICE and ADDR are discussed next and TC is discussed in the I/O Subsystem Interface section.

### SLICE ASICs

The four SLICE ASICs are used strictly for data path; together they form a 32-byte bus to main memory, a 16-byte bus to the CPU and cache, and a 4-byte bus to the TC ASIC. It is helpful to think of the SLICE ASICs as a train station for data with the data buses as train tracks. Data can come and go on any track, different tracks have different speeds and widths,

and data can find temporary storage in the ASICs. The SLICE ASICs provide the systems with a location to buffer DMA, I/O read, I/O write, and victim data while the data waits to travel the next leg of its journey. The use of the SLICE ASICs also eliminates one to two levels of buffering between the dynamic random-access memories (DRAMs) and the CPU, thus decreasing latency and improving bandwidth.

A key design decision was determining the width of the memory data bus. A conventional design would have matched the width of the memory bus to the width of the cache bus (16 bytes). However, to reduce the memory latency of the second half of the cache block (cache line size is 32 bytes), the system reads the entire cache block from memory at once using a 32-byte memory bus. This technique eliminates the additional latency from a second page-mode read.

The DEC 3000 AXP Model 500 returns the entire block to the cache and CPU with an average latency of only 180 nanoseconds (ns) from the CPU memory request. In contrast, a less aggressive preliminary design using a system bus and 16-byte-wide memory bus yielded an average memory latency of 320 ns. The 32-byte memory bus costs little more than a 16-byte bus—two low-cost ASICs, resistor packs, and some address fan-out parts.

### *ADDR ASIC*

The ADDR ASIC is a crossbar for addresses. ADDR sends addresses from the CPU to memory (CPU reads and writes), from the CPU to I/O (I/O reads and writes), and from the I/O to CPU and memory (DMA reads and writes). ADDR selects between CPU read, victim write, and DMA addresses to send to memory. A counter that increments DMA addresses on long TURBOchannel DMAs also resides in ADDR.

ADDR provides a home to the memory configuration registers. At power-on time, the boot firmware writes and reads memory space, determines the memory configuration, and writes the configuration registers. At run time, each memory address maps into a unique bank, regardless of the type and order of the single in-line memory modules (SIMMs) installed.

ADDR also provides a home for miscellaneous functions such as tag parity checking, refresh counter, and the locked physical address register. It generates the cache probe index to check the cache tags for a hit or a miss on DMA probes.

### *Memory Mother Board and SIMMs*

The memory system is composed of memory mother boards (MMBs) that rise from the system card, and SIMMs. This arrangement is a good solution to the problem of limited space on the system module. It allows for a wide data bus and for good signal integrity for short propagation times on the memory data bus.

As shown in Figure 5, an MMB module supports up to eight SIMMs at a time (four SIMMs in Model 400 systems). A minimum of two SIMMs is required for each board. A system always contains four MMBs. The MMBs act as a carrier for the SIMMs and also contain drivers for address and control signals.

A total of 8, 16, 24, or 32 SIMMs (maximum of 16 in Model 400 systems) can be plugged into the system. SIMMs may be single- or double-sided with 10 DRAMs

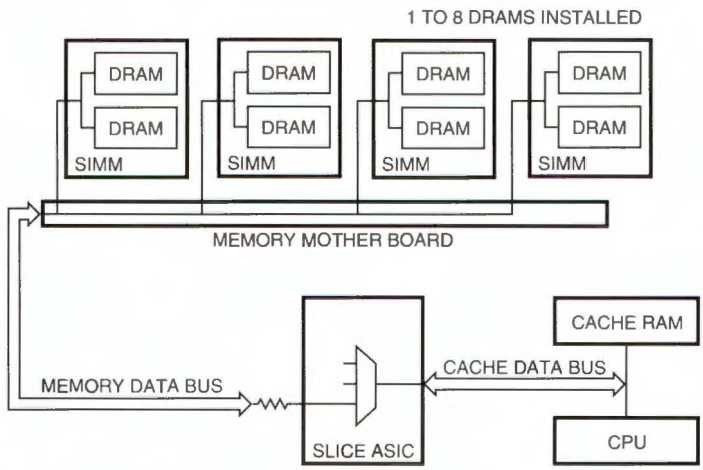



*Figure 5 Memory and Cache Data Bus*

per side. Each side of a SIMM constitutes one-eighth of a bank. Eight SIMMs must be plugged in to complete a bank; hence the 320-bit-wide data bus (4 bits per DRAM by 10 DRAMs per SIMM by 8 SIMMs). One megabit (Mb), 4Mb, and 16Mb DRAMs are supported, and users are allowed to populate banks in any order. In this way, the DEC 3000 AXP Model 500 can support from 8MB to 1 gigabyte (GB) of memory, and the DEC 3000 AXP Model 400 can support 8MB to 512MB of memory.

Main memory is protected by a single-bit-correct, double-bit-detect error-correcting code (ECC). In addition, the arrangement of data bits allows the detection of any number of errors restricted to a single DRAM chip. ECC corrections for CPU transactions are performed by the CPU, and corrections for I/O transactions are done in the TC ASIC.

### *Memory Transactions*

When data is stored in the B-cache by the CPU, it is not immediately sent to memory. Data is written to main memory only when a dirty block in the cache is replaced. Data destined for the cache is read from main memory only on cache misses. Reads to main memory, whether from the CPU or from DMA, always return 32 bytes. On CPU reads of main memory, data is returned to the cache and CPU in two halves by the SLICE ASICs. Likewise when the B-cache control writes victim data to main memory, two reads are made of the cache, but only one write is made to main memory.

On DMA writes, 4 bytes of data arrive from the TURBOchannel interface ASIC each cycle and are stored in the SLICE ASICs. The SLICE ASICs can buffer up to 128 bytes of data prior to writing the data to main memory using page-mode writes, 32 bytes at a time. To maintain cache/memory coherence, data is also provided to the cache RAMs so that it may be written in the case of a cache hit. On DMA reads, up to 128 bytes of data are read page mode out of main memory and buffered in the SLICE ASICs. Data flows out to the TC ASIC and the TURBOchannel bus at the rate of 4 bytes per cycle (100MB/s). In the event of a cache hit, data is taken preferentially from the cache.

The crossbar employs a technique that permits simultaneous transactions from CPU to main memory and DMA. The TURBOchannel bus supports DMA transactions of up to 512 bytes in length. Once the DMA starts, the system must be able to provide or receive data without any gaps. However, while the DMA buffer in the SLICE ASICs is sufficiently full (for DMA reads) or empty (for DMA writes), the CPU is allowed to use memory. When the I/O controller detects that the buffer is too full or too empty, it requests memory time to service the DMA buffer. At this time, further CPU requests are temporarily ignored. This technique prevents the CPU from being locked out of main memory, even during long DMA transactions and even though DMA has priority over CPU transactions.

The crossbar also permits simultaneous write transactions from the CPU to main memory and from the CPU to an I/O device. SLICE and ADDR ASICs can buffer one I/O write transaction of up to 32 bytes in size. Once the ASICs have accepted the data and address, the cache and crossbar are free to process other CPU transactions, which can include cache and main memory reads and writes. If the CPU issues an I/O write while a previous write is still pending in the ASICs, the cache controller simply stalls.

## *I/O Subsystem Interface/ TURBOchannel ASIC*

The I/O system is based on the TURBOchannel, a 32-bit high-performance, bidirectional, multiplexed address and data bus developed by Digital for workstations.[3] The DEC 3000 AXP supports up to six plug-in options, as well as the integral smart frame buffer (SFB) graphics ASIC, the I/O controller (IOCTL) ASIC, and the TURBOchannel dual SCSI (TCDS) ASIC. The TURBOchannel bus is synchronous and requires only five control signals in each direction between the system and the option cards.

The system interfaces to the TURBOchannel bus by a data-path TC ASIC and control logic contained in a number of programmable array logic devices (PALs). The TC ASIC completes the system crossbar by passing addresses between the TURBOchannel bus and the address ASIC, and passing data between the TURBOchannel bus and the SLICE ASICs. Furthermore, the TC ASIC checks and generates parity on the TURBOchannel, and checks, corrects, and generates ECC on the data bus to the SLICE ASICs. Parity checking of TURBOchannel data is optional and is enabled on a per-option basis through a configuration register in the TC ASIC. Finally, the TC ASIC contains a number of counters for tracking DMA progress, as well as configuration and error registers. All control logic was implemented in PALs to minimize the impact to the project schedule of any design changes. The TURBOchannel interface block diagram in shown in Figure 6.

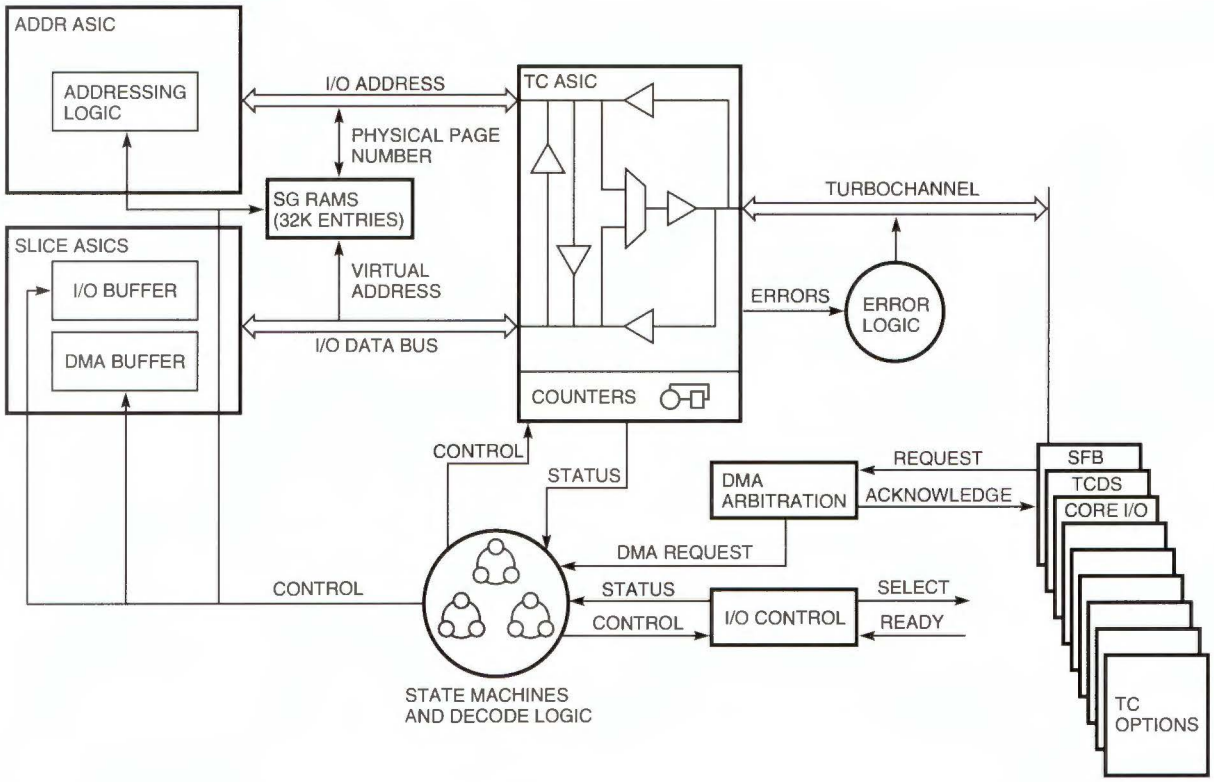



*Figure 6 TURBOchannel Interface Block Diagram*

There are two types of TURBOchannel operations: the system initiates I/O reads and writes, and the options initiate DMA reads and writes. On an I/O operation, the system sends the I/O address from the ADDR ASIC to the TC ASIC, and from there to the TURBOchannel. For I/O reads, the option returns data on the TURBOchannel. This data passes through the TC ASIC and over the bus to the SLICE ASICs. The system includes some special hardware for byte masking of I/O read data. This hardware is used to provide support for VMEbus adapters.

For I/O writes, the system sends data from the SLICE ASICs across the data bus to the TC ASIC. The TC ASIC then sends it to the option over the TURBOchannel. The DEC 3000 AXP workstation supports a block write extension to the original TURBOchannel protocol. In this mode, the system supplies a single address followed by multiple consecutive data transfers for improved I/O write performance. This extension is also configurable on a per-option basis through the TC configuration register.

The TURBOchannel protocol specifies that before any option can use the bus for DMA, it must issue a request to the system. The DEC 3000 AXP architecture employs an arbitration scheme using rotating priority that prevents any option from being locked out. After being granted the bus, the option supplies a DMA address on the TURBOchannel bus. This address routes through the TC ASIC and onto the address ASIC. In the case of a DMA write, data immediately follows the address on the TURBOchannel. This data passes through the TC ASIC and onto the data bus to the SLICE buffers.

DMA reads are more complicated than writes because the TURBOchannel bus does not transmit ahead of time the number of bytes of data to be read from memory. Instead, it continues to assert its read request signal for as long as it is requesting data. The SLICE buffers begin to fill up with DMA data, and only when they can guarantee that there will be no gaps in the DMA will the data transfer start. The TC ASIC receives the read data from the SLICE ASICs and sends it onto the TURBOchannel to the requesting option.

Virtual DMA allows the system to map noncontiguous regions of physical address space into contiguous regions of virtual address space. This

method allows TURBOchannel options to transfer large blocks of DMA data without knowledge of how that data is mapped in the physical address space in main memory. Virtual DMA enhances operating system performance because the memory mapping is performed before the transfer of DMA data.

The DEC 3000 AXP workstation supports virtual DMA through the use of a scatter/gather (SG) map, which acts as a translation buffer. SG mapping is enabled on a per-option basis through the configuration register in the TC ASIC. The SG map is organized as 32K 24-bit entries. Each entry contains a 17-bit physical page number (PPN), parity, and valid bit. Software sets up the map through I/O space reads and writes. DMA byte address bits [27:13] index the SG map, which produces a 17-bit PPN (bits [29:13]) to append to the virtual DMA byte address bits [12:0]. The resulting 30-bit physical DMA byte address can then address all 1GB of the possible system address space. An SG map is shown in Figure 7.

## I/O Subsystem

Most of the I/O subsystem is implemented on its own module. This I/O module, shown in Figure 8, contains the connectors for attachment unit interface (AUI) Ethernet, 10Base-T Ethernet, Integrated Services Digital Network (ISDN), alternate console/serial printer, mouse/keyboard, communications, internal and external SCSI, three TURBOchannel options, and audio module port. The various I/O controllers interface to the TURBOchannel through one of three ASICs. These ASICs are the smart frame buffer (SFB) on the CPU module and the TURBOchannel dual SCSI (TCDS) ASIC and the I/O controller (IOCTL) ASIC on the I/O module.

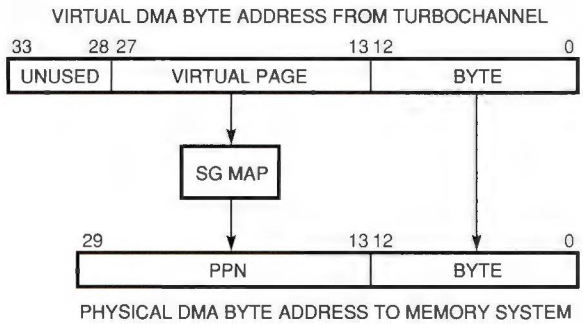



*Figure 7 Scatter/Gather Mapping*

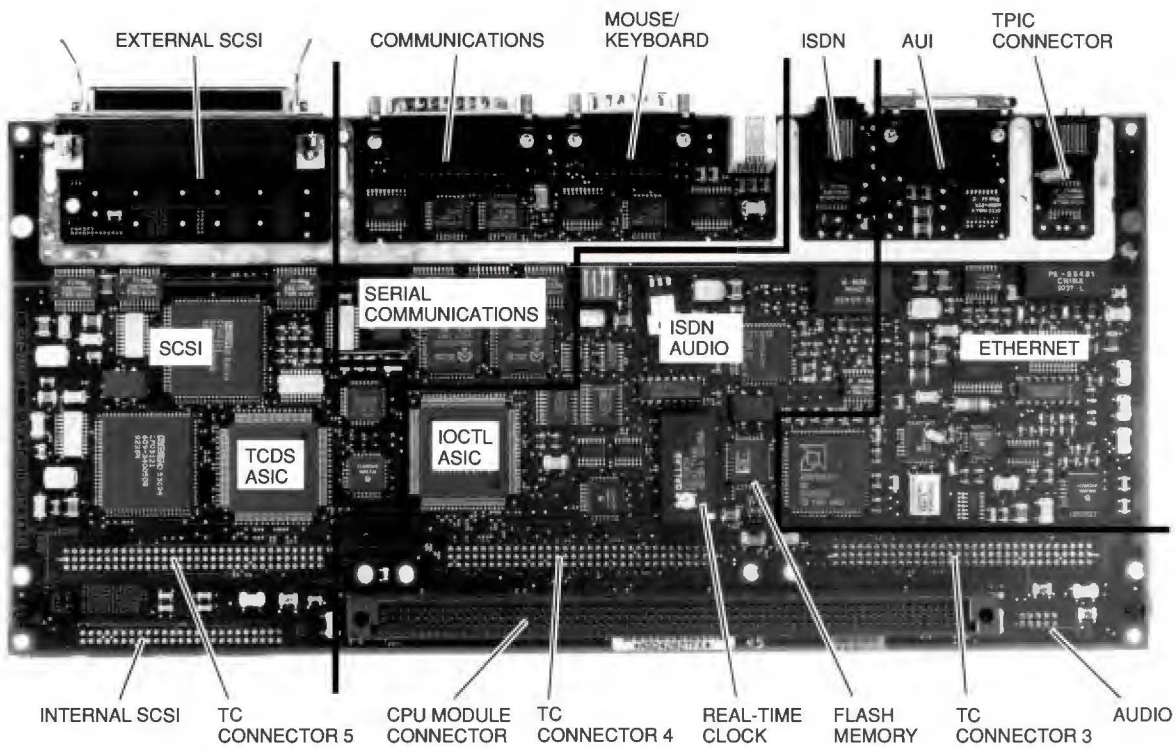



*Figure 8 I/O Module*

### *I/O Module–IOCTL ASIC*

A key I/O subsystem design decision was to reduce time-to-market by eliminating unnecessary new hardware and software development. To support most of the I/O functionality, the designers chose the IOCTL ASIC developed for the DECstation 5000 Model 240.

The IOCTL ASIC provides an interface to a 16-bit, general-purpose I/O bus, which supports the following devices: two Zilog Z85C30 serial communications controllers (SCCs), an AMD 7990 local area network controller for Ethernet (LANCE), a Dallas semiconductor DS1287 real-time clock, an AMD 79C30A ISDN data controller (IDC), a SCSI controller, and an AMD 27C020 256KB erasable programmable read-only memory (EPROM).

The SCCs implement the keyboard, mouse, alternate console/printer, and communications ports. The mouse and keyboard do not use DMA. The alternate console/printer and the communications port do use DMA.

The LANCE implements the Ethernet interface, which connects to the local area network (LAN) through either the AUI (thickwire) or 10Base-T (twisted-pair interconnect [TPIC]) connectors. Software controls which one of these interfaces is enabled.

The real-time clock provides time-of-year (TOY) reference and 50 bytes of nonvolatile RAM. A lithium battery supplies power in the event of system power-off or failure.

The IDC implements both an ISDN interface and telephone-quality audio. The audio connects to the audio interface module (AIM), which provides the audio I/O in the Model 500. Audio I/O in the Model 400 is on its I/O module.

The AIM on the Model 500 supports audio input through either a ⅛-inch minijack for microphone input, a 4-pin modular jack (MJ) connector for use of a telephone handset, or an RCA-style phonograph jack used as a line-in input. Output is provided by the MJ connector as well as by a ⅛-inch stereophonic jack. The stereophonic jack accepts only a stereophonic plug. If monophonic headphones are used, a mono-to-stereophonic adapter is required. On the Model 400, audio input and output is implemented using a 4-pin MJ connector.

Analysis of the complete audio system in a Model 500 shows a frequency response of 145 Hz to 3,500 Hz, with typical distortion in the 0.8 percent to 1.9 percent range for the microphone and 0.4 percent to 1.5 percent for the telephone handset. The signal-to-noise ratio ranged from 24 decibels with a minimal signal input to 58 decibels with a high-level signal input.

### *I/O Module–TCDS ASIC*

Although the IOCTL ASIC contains an interface to a SCSI controller, the DEC 3000 AXP systems implement their SCSI interface using the TCDS ASIC. This design has several advantages. First, the TCDS ASIC supports two SCSI ports rather than the one supported by the IOCTL ASIC, permitting separate internal and external SCSI chains. Second, this design eliminates contention between the Ethernet controller and the SCSI controller for the IOCTL bus. Third, the TCDS ASIC supports much longer TURBOchannel DMA bursts (64-byte bursts rather than 16-byte bursts). Finally, the resulting ASIC design is used to implement a dual SCSI TURBOchannel option module.

The TCDS ASIC implements two separate SCSI ports using two NCR 53C94 advanced SCSI controllers (ASCs). The TCDS allows both controllers to have DMA transfers in progress simultaneously.

TCDS TURBOchannel DMA transactions are aligned 64-byte blocks. Starting DMA addresses that are not aligned to these boundaries begin with a smaller DMA transaction. This technique aligns the address so that succeeding transactions are aligned 64-byte blocks. Large, aligned transactions increase both TURBOchannel and memory access efficiency.

The TCDS ASIC and the ASCs provide odd parity protection on major data paths. This protection includes 8-bit parity on the 16-bit bus between the TCDS and the ASCs, 32-bit parity on TCDS DMA buffer entries, and 32-bit parity on TURBOchannel transactions, both I/O and DMA.

## *Graphics*

The graphics subsystem on the Model 500 system card provides integral 8-plane graphics with hardware enhancements for improved frame buffer performance. These enhancements increase the performance of stipple, line drawing, and copy operations. The graphics system consists of an SFB ASIC, 2MB video RAM, and the Brooktree Bt459 RAMDAC chip for sourcing the 8-plane RGB data. The user can select either a 66-Hz or a 72-Hz monitor refresh rate through a switch on the back of the workstation. The graphics subsystem can draw 615K two-dimensional vectors per second and can perform copy operations at 31.8MB/s.

The graphics subsystem is available separately as the TURBOchannel HX graphics option card. In addition, high-performance two-dimensional and three-dimensional graphics accelerators are available through the TURBOchannel bus for all systems.

## Clock System

The input clock circuitry to the DECchip 21064 CPU contains a differential 300-MHz oscillator (266 MHz for the Model 400), which drives an alternating current (AC) decoupling circuit and the CPU chip. The CPU chip divides down the input clock frequency by a factor of two and operates internally at 150 MHz. The DEC 3000 AXP Model 500 is capable of supporting a 200-MHz CPU with a 400-MHz oscillator.

The entire system, with the exception of some I/O devices, runs synchronously. The master system clock is generated by the CPU chip at a frequency of 25 MHz (22 MHz for the Model 400), resulting in system clock cycles of 40-ns duration. This master system clock is duplicated and distributed with differential pseudo-emitter coupled logic (PECL) to maintain minimum skew and to improve noise margin. The PECL clocks are converted to transistor-transistor logic (TTL) in the last stage of the clock fan-out tree.

Two stages of system clock fan-out are used as shown in Figure 9. Two MC100E111 ECL clock buffer chips (PECL input and output) provide 18 low-skew differential copies of the clock. Seventeen MC100H641 ECL-to-TTL converters (PECL input, TTL output) are distributed throughout the system and I/O boards to provide more than 100 clock lines. All clock lines are length matched to reduce skew, and PECL wires are separated from TTL. Worst-case SPICE simulation indicates a skew between typical components such as PALs to be 1.5 ns. Actual skews measured in the lab are approximately 0.5 ns.

To give designers maximum flexibility, four phases of the system clock are generated, one every 10 ns. Delay lines are used to generate an offset of 10 ns. By swapping the high and low differential inputs to selected MC100H641 converters, the 20- and 30-ns delayed clocks are generated. The master system clock is delayed using delay lines so that the eventual system clock is synchronous with the CPU chip.

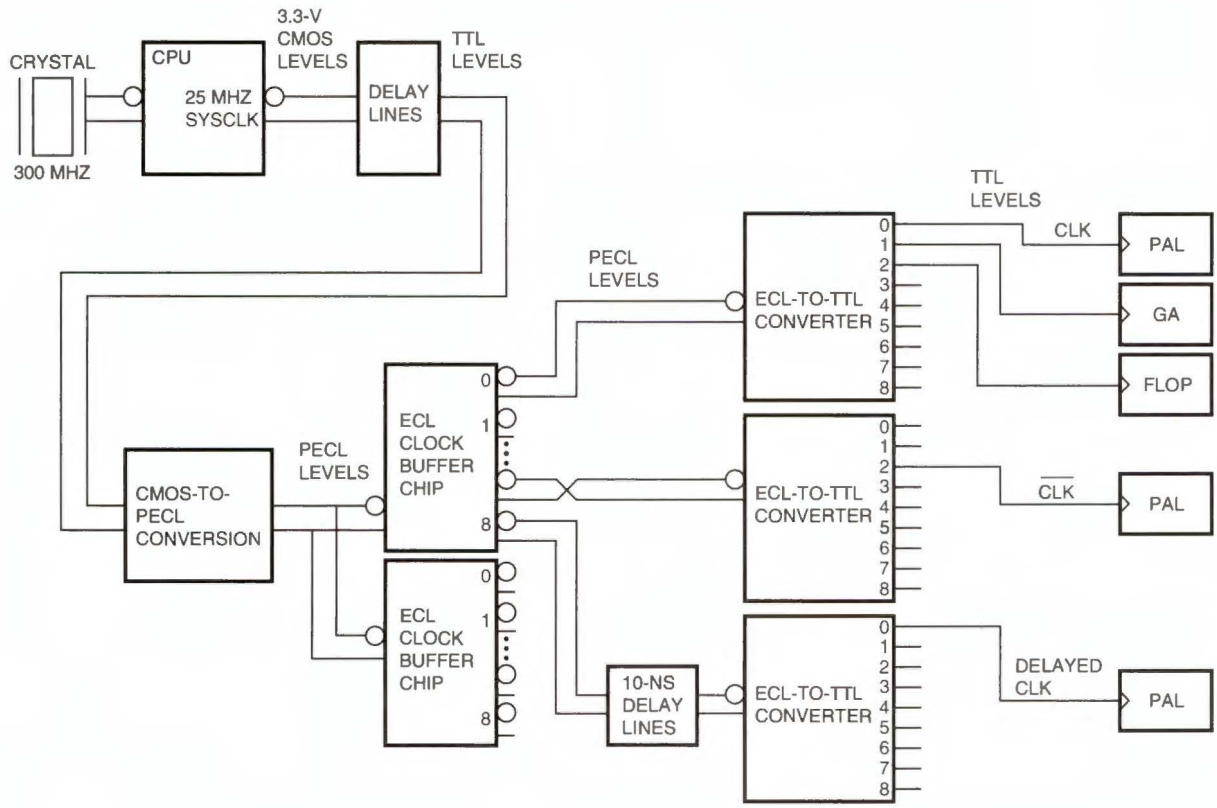



*Figure 9 Clock Distribution*

## Technology

The goal in choosing semiconductor devices was to select mature silicon technologies and then push those technologies to the limit. Module- and chip-level signal integrity was verified by correlating silicon bench characterization data to device simulation modules. CAD tools were used to perform worst-case module timing and signal integrity simulation. This methodology minimized device costs, reduced risks, and shortened time-to-market.

The nine ASICs in a DEC 3000 AXP workstation use six unique 1.0-micrometer complementary metal-oxide semiconductor (CMOS) designs. (See Table 2.) Plastic quad flat packs (PQFP) are used as the packaging technology to limit device cost. Because the ASICs are I/O limited and the PQFPs do not have ground planes, the effects of simultaneous switching outputs (SSOs) were a concern. The potential effects of SSOs in CMOS output buffers include corrupted data and undesirable oscillations. Simulation and bench characterization were used to quantify the SSO effects, and in some cases SSOs were reduced by staggering output driver timing.

Although ASICs were chosen for the data path, PALs were used for control logic due to their greater flexibility and faster turnaround time. A total of 63 20XX (5 ns) and 22V10 (10 ns) PALs with 57 different codes was used. Exhaustive system-level simulation and bench characterizations were performed to understand device behavior in the many different loading scenarios.

The CPU board technology proved moderately difficult for system-level assembly due to the large distance between the fine-pitch (25 mil) components. There are 19 fine-pitch components on the 14- by 16-inch CPU board, with a maximum distance of 14 inches between any two devices. With this large distance, an aggressive, true positional diameter (TPD) tolerance requirement of 6 mils was implemented. TPD is defined as the total diameter of permissible movement from a theoretical exact location around the true position of the pads. This TPD requirement ensures proper positional accuracy between the solder paste stencil apertures and the surface-mount features. In addition, solder mask between pads on the fine-pitch components is used to reduce manufacturing defects.

To reduce power and cost, the slower DEC 3000 AXP Model 400 design substitutes CMOS technology for the BiCMOS cache SRAMs and for many of the bipolar PALs.

## Power and Packaging

The following fixed disk drive options are currently available.

- RZ25 3.5-inch half-height 426MB disk drive
- RZ26 3.5-inch half-height 1050MB disk drive

The following removable media options are also available.

- RRD42 5.25-inch half-height 600MB CD-ROM drive
- RX26 3.5-inch half-height 2.8MB floppy disk drive
- TZK10 5.25-inch half-height 525MB QIC tape drive
- TLZ06 5.25-inch half-height 4000MB DAT drive

The Model 500 has a 480-watt output, off-line, switching regulated power supply, which includes a capacitor-input, automatic voltage-selecting circuit to permit worldwide operation without a voltage-select jumper for 120 or 240 volt (V) input. The power supply provides five outputs to the load: +3.3 V, +5.1 V-CPU, +5.1 V-turbo, +12.1 V, and −12.1 V.

The power supply also provides power for three external fans. Temperature-sensing fan speed control is provided to reduce system noise. The power

**Table 2 ASICs Used on the DEC 3000 AXP Workstations**

| Chip | Total Number of Pins | Number of Pins Used | Number of Signal Pins | Used Gates | Available Gates |
|---|---|---|---|---|---|
| SFB | 184 | 184 | 150 | 21.6K | 54K |
| TC | 184 | 182 | 144 | 12.1K | 44K |
| SLICE | 184 | 184 | 153 | 11.2K | 44K |
| ADDR | 184 | 183 | 148 | 5.7K | 44K |
| TCDS | 120 | 120 | 94 | 26.5K | 68K |
| IOCTL | 160 | 160 | 126 | 11.2K | 44K |

supply senses tachometer outputs from the fans, and when a fan fails, it shuts down and illuminates an indicator.

## Manufacturability/Testability

The designers provided several debugging features, including test points on the module, tristate outputs on ASICs and PALs, an on-board diagnostic ROM, and programmable console ROM. Since the module is composed almost exclusively of surface-mount devices, the designers specified as many vias as possible for use as test points. Consequently, all wires on the board have test points, which allows for 100 percent short-circuit coverage and 94 percent open-circuit coverage.

The DEC 3000 AXP workstation takes full advantage of the serial ROM port on the DECchip 21064 CPU. This port allows code to be directly loaded into the instruction cache. During prototype development, designers loaded special debug programs into the CPU through this port. However, the real innovation is in also wiring this port to the output of a 64K by 8 EPROM on the module to provide 8 programs that are individually selectable by moving a jumper on the module. On system reset, serial program data from the selected EPROM output is loaded into the instruction cache. These programs include power-up code for loading the real console, a miniconsole, and five diagnostic programs for testing memory and the graphics subsystem. Other tests are available by replacing the EPROM. These programs are of great value in the manufacturing debug environment.

Two flash EPROMs contain the console code for the system. On power-up, code in the serial ROM loads the console code into memory and begins executing it. Users can easily update the console ROMs (for example, to provide PAL code enhancements) through a special utility booted off a CD-ROM connected to the system. Field service can update the console code in the system remotely through the Ethernet.

## Conclusions

The primary goal of this project was to design a balanced system that exhibited low memory latency, high memory bandwidth, and minimal CPU-I/O memory contention in a cost-effective manner. Table 3 gives the measured peformance numbers for these characteristics. Except where noted, all numbers are for sustained performance. Of particular note are the numbers showing that the CPU

**Table 3 System Performance**

| | DEC 3000 AXP Model 500 | DEC 3000 AXP Model 400 |
|---|---|---|
| CPU speed | 150 MHz | 133 MHz |
| B-cache size | 512KB | 512KB |
| B-cache read bandwidth | 480MB/s | 426MB/s |
| B-cache write bandwidth | 320MB/s | 284MB/s |
| Maximum main memory | 1GB | 512MB |
| CPU memory latency (average) | 32 bytes/180 ns | 32 bytes/203 ns |
| CPU memory read bandwith | 114MB/s | 101MB/s |
| CPU read with victim write memory bandwidth | 160MB/s | 141MB/s |
| TURBOchannel peak bandwidth | 100MB/s | 89MB/s |
| I/O read bandwidth 8 bytes | 13MB/s | 12MB/s |
| I/O write bandwidth 8 bytes | 33MB/s | 29MB/s |
| Block I/O write bandwidth 32 bytes | 67MB/s | 59MB/s |
| Block I/O write bandwidth 32 bytes with CPU read and victim write memory bandwidth | I/O=53MB/s MEM=107MB/s | I/O=47MB/s MEM=95MB/s |
| DMA read bandwidth 512 bytes | 91MB/s | 81MB/s |
| 64 bytes | 57MB/s | 51MB/s |
| DMA write bandwidth 512 bytes | 93MB/s | 82MB/s |
| 64 bytes | 59MB/s | 52MB/s |
| 64-byte DMA write bandwith with CPU reads from memory | DMA=59MB/s CPU=30MB/s | DMA=52MB/s CPU=27MB/s |

receives significant memory bandwidth even in the presence of heavy block I/O and DMA traffic.

Another goal of the project was to offer performance that is competitive with RISC workstations available from other vendors. The benchmark performance of any system derives from the interdependent performance of the hardware, the operating system, and the compilers that generate the application code. The benchmark performance should improve as each element matures. Table 4 shows the performance of the DEC 3000 AXP systems on a selected set of benchmarks as of the announcement dates of these products. Table 5 compares the performance of the DEC 3000 AXP Model 500 to the published performance of several currently available competitive systems.[4]

## *Acknowledgments*

The DEC 3000 AXP Model 500 design was a team effort—more people were involved than can be acknowledged in this space. Recognition is due to those who contributed to the design of original hardware: Dave Archer, Mark Baxter, John DeRosa, Chris Gianos, Leon Hesch, Dave Laurello, Bob McNamara, Dick Miller, Rick Rudman, Dave Senerchia, Petr Spacek, Bob Stewart, Ned Utzig, Debbie Vogt, and John Zurawski. The tight schedule could not have been met without the special efforts of the Power and Packaging, Console, Qualification, Proto Management, and Technology and Operating Systems Groups. The design team for the DEC 3000 AXP Model 400 project is also recognized: John Day, Jamie Pierce, Dennis Rainville, and Ken Ward. The thorough device evaluations by Rob Zahora contributed significantly to the success of the projects. We would also like to acknowledge the contributions by FXO personnel. The Electronic Storage Development Group was responsible for the design of the DEC 3000 AXP Model 500 memory module. Significant efforts by the Maynard TME, Albuquerque, and Ayr Manufacturing Plants should be recognized for delivering quality hardware

**Table 4 Benchmark Performance**

| | DEC 3000 AXP Model 400 | DEC 3000 AXP Model 500 |
|---|---|---|
| Clock (MHz) | 133 | 150 |
| SPECmark89 | 108.1 | 121.5 |
| Dhrystones | | |
| V1.1 (Dhrystones per second) | 228.3K | 257.7K |
| V2.1 (Dhrystones per second) | 249.6K | 281.2K |
| LINPACK 64-bit double precision | | |
| 100 × 100 (MFLOPS)* | 26.4 | 30.2 |
| 1000 × 1000 (MFLOPS) | 70.8 | 79.9 |
| X11PERF | | |
| Two-dimensional vectors per second | 564.0K | 636.0K |
| Two-dimensional pixels per second | 27.4M | 31.0M |

Note: *Million floating-point operations per second

**Table 5 Competitive Comparison**

| | DEC 3000 Model 500 | IBM RS6000 Model 580 | HP9000 Model 750 |
|---|---|---|---|
| SPECmark89 | 121.5 | 126.2 | 86.6 |
| Dhrystones | | | |
| V1.1 (Dhrystones per second) | 257.7K | n/a | 133.7K |
| V2.1 (Dhrystones per second) | 281.2K | n/a | 122.3K |
| LINPACK 64-bit double precision | | | |
| 100 × 100 (MFLOPS) | 26.4 | 38.1 | 23.7 |
| 1000 × 1000 (MFLOPS) | 79.9 | 84.0 | n/a |

during the development and production phases; a special thanks to Jim Ersfeld for his significant efforts in this regard.

## References

1. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, MA: Digital Press, Order No. EY-L520E-DP, 1992).

2. D. Dobberpuhl et al., "A 200-MHz 64-bit Dual-issue CMOS Microprocessor," *IEEE Journal of Solid-State Circuits,* vol. 27, no. 11 (November 1992): 1555-1567 and *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 35–50.

3. TURBOchannel Specifications, Version 2C (Palo Alto, CA: Digital Equipment Corporation, TRI/ADD Program, Order No. EK-TCDEV-DK-004, September 1991).

4. Alpha AXP Workstation Family Performance Brief—OpenVMS, Second Edition (Maynard: Digital Equipment Corporation, Order No. EB-N0102-51, November 20, 1992).

*Barry A. Maskas*
*Stephen F. Shirron*
*Nicholas A. Warchol*

# *Design and Performance of the DEC 4000 AXP Departmental Server Computing Systems*

*DEC 4000 AXP systems demonstrate the highest performance and functionality for Digital's 4000 series of departmental server systems. DEC 4000 AXP systems are based on Digital's Alpha AXP architecture and the IEEE's Futurebus+ profile B standard. They provide symmetric multiprocessing performance for OpenVMS AXP and DEC OSF/1 AXP operating systems in an office environment. The DEC 4000 AXP systems were designed to optimize the cost-performance ratio and to include upgradability and expandability. The systems combine the DECchip 21064 microprocessor, submicron CMOS sea-of-gates technology, CMOS memory and I/O peripherals technology, a high-performance multiprocessing backplane interconnect, and modular system design to supply the most advanced functionality for performance-driven applications.*

The goal of the departmental server project was to establish Digital's 4000 family as the industry's most cost-effective and highest-performance departmental server computing systems. To achieve this goal, two design objectives were proposed for the DEC 4000 AXP server. First, migration was necessary from the VAX architecture, which is based on a complex instruction set computer (CISC), to the Alpha AXP architecture, which is based on a reduced instruction set computer (RISC). Second, for expansion I/O in an upgradable office environment enclosure, migration was necessary from the Q-bus to the Futurebus+ I/O bus.[1] In addition, the new system had to provide balance between processor performance and I/O performance. Maintaining customer investments in VAX and MIPS applications through support of OpenVMS AXP and DEC OSF/1 AXP operating systems was implicit in the architecture migration objective. Migration, porting, and upgrade paths of various applications were defined.

This paper focuses on the design of the DEC 4000 AXP hardware and firmware. It begins with a discussion of the system architecture and the selection of the system technology. The paper then details the CPU, I/O, memory and power subsystems. It concludes with a performance summary.

## *System Overview*

The DEC 4000 AXP system provides supercomputer class performance at office system cost.[2] This combination was achieved through architecture and technology selections that provide optimized uniprocessor performance, low additional cost symmetric multiprocessing (SMP), and balanced I/O throughput. High I/O throughput was accomplished through a combination of integrated controllers and a bridge to Futurebus+ expansion I/O. The system uses a modular, expandable, and portable enclosure, as shown in Figure 1. With current technologies, the system supports up to 2 gigabytes (GB) of dynamic random-access memory (DRAM), 24GB of fixed mass storage, and 16GB of removable mass storage. The DEC 4000 AXP system is partitioned into the following modular subsystems:

- Enclosure (BA640 box)
- CPU module (DECchip 21064 processor)
- I/O module
- Memory modules
- Mass storage compartments and storage device assembly (brick)

*Figure 1 DEC 4000 AXP System Enclosure*

- Futurebus+ Expansion I/O, Futurebus+ controller module (FBE)
- Power supply modules - universal line front-end unit (FEU)
  - Power system controller (PSC)
  - DC-DC converter unit 5.0 volt (V) (DC5)
  - DC-DC converter unit 2.1 V, 3.3 V, 12.0 V (DC3)
- Cooling subsystem
- Centerplane module
- Operator control panel (OCP)
- Digital storage systems interface (DSSI) and small computer systems interface (SCSI) termination voltage converter (VTERM)

Figure 2 shows these subsystems in a functional diagram. The subsystems are interconnected by a serial control bus, which is based on Signetic's $I^2C$ bus.[3]

## System Architecture

From the beginning of the project, it was apparent that the I/O subsystem had to be equal to the increased processing power provided by the DECchip 21064 CPU. Although processing power was taking a revolutionary jump in performance with no cost increase, disk and main memory technology were still on an evolutionary cost and performance curve. The metrics that had been used for VAX systems were difficult, if not impossible, to meet through linear scaling within a fixed cost bracket. These metrics were based on VAX-11/780 units of performance (VUPs); they give main memory capacity in megabytes (MB)/VUP, disk-queued I/O (QIO) completions in QIO/s/VUP, and disk data rate in MB/s/VUP. As an example, Table 1 gives the metrics for a VAX 4000 Model 300 scaled linearly to 125 VUPs and then nonlinearly scaled for the DEC 4000 AXP system implementation. Performance modeling of the DECchip 21064 CPU suggested that 125 VUPs was a reasonable goal for the DEC 4000 AXP.

Without an Alpha AXP architecture customer base, we did not know if these metrics would scale linearly with the processor performance. The DECchip 21064 processor technology has the potential for attracting new classes of compute-intensive applications that may make these metrics obsolete. We therefore chose a nonlinear extrapolation of the metrics for our initial implementation. By trading off disk and memory capacity for I/O throughput performance, we kept within established cost and performance goals. The implementation metrics were not limited by the architecture; further scaling up of metrics was planned. Of the four metrics, the disk capacity metric has the most growth potential.

To ensure compliance with both the Alpha AXP architecture and the Futurebus+ specifications, the system was partitioned as shown in Figure 2. The bridge between the CPU subsystem and the Futurebus+ subsystem afforded maximum design flexibility to accommodate specification changes, modularity, and upgradability. The I/O module was organized to balance the requirements between CPU performance and I/O throughput rates. The DEC 4000 AXP system implementation is based on open standards, with a six-slot Futurebus+ serving as the expansion I/O bus and the system bus serving to interconnect memory, CPUs, and the I/O module. The modularity of the system enables module swap upgrades and configurability of the I/O subsystem such that performance and functionality may be tailored to user requirements. The modularity aspects of the system design extend into the storage

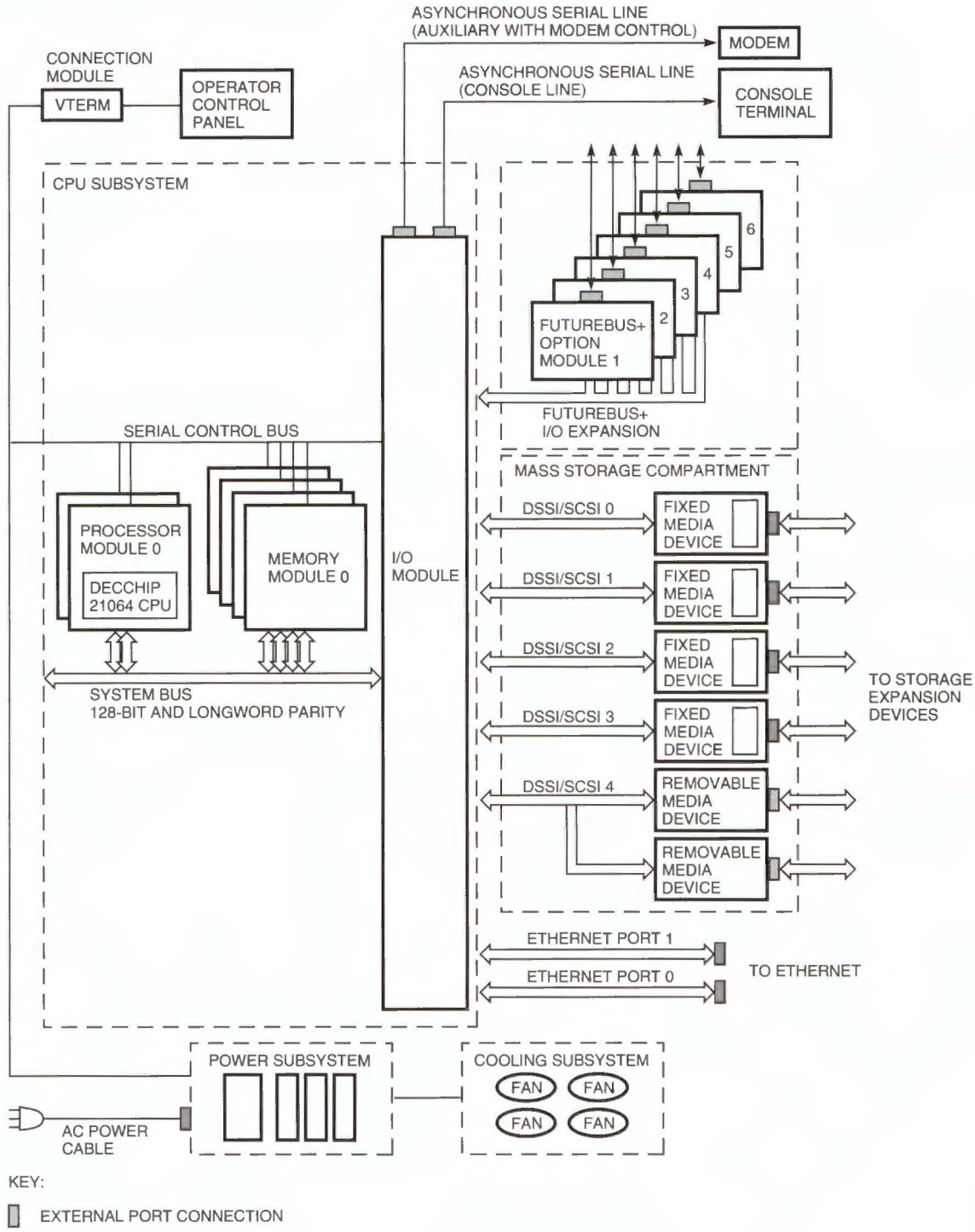



*Figure 2 DEC 4000 AXP System Functional Partition*

**Table 1 Extrapolated VAX Metrics**

| | VAX 4000 Model 300 Metrics | Scaled Linearly to 125 VUPs | Scaled Nonlinearly for DEC 4000 AXP |
|---|---|---|---|
| Memory capacity | 60 MB/VUP | 7.5 GB | 2 GB |
| Disk capacity | 1.65 GB/VUP | 206 GB | 100 GB |
| Disk QIO rate | 49 QIO/s/VUP | 6,125 QIO/s | >4,000 QIO/s |
| I/O data transfer rate | 1.4 MB/s/VUP | 175 MB/s | 210 MB/s |

compartment where each brick has a dedicated controller and power converter. Support for DSSI, SCSI, and high-speed 10MB/s SCSI provides maximum flexibility in the storage compartment. The modular mass storage compartments enable user optimization for bulk storage, fast access, or both.

The cost of SMP was a key issue initially, since Digital's SMP systems were considered high-end systems. Pulling high-end functionality into lower-cost systems through architecture and technology selection was managed by evaluation of performance and cost through trial designs and software breadboarding. Several designs of a CPU module were proposed, including various organizations of one or two DECchip 21064 CPUs per module interfaced to I/O and memory subsystems. Optimization of complexity, parts cost, performance, and power density resulted in a CPU module with one processor that could operate in either of two CPU slots on the centerplane. Consequently, a system bus had to be developed that could be interfaced by processors, memory, and I/O subsystems in support of the shared-memory architecture.

As development of the DECchip 21064 processor progressed, hardware engineers and chip designers established a prioritized list of design goals for the system bus as follows:

1. Provide a low-latency response to the CPU's second-level cache-miss transactions and I/O module read transactions without pending transactions.
2. Provide a low-cost shared-memory bus, based on the cache coherence protocol, that would facilitate upgrades to faster CPU modules. This provision implied a simple protocol, synchronous timing, and the use of transistor-transistor logic (TTL) levels rather than special electrical interfaces.
3. Provide I/O bandwidth enabling local I/O to operate at 25 megabytes per second (MB/s) and the Futurebus+ to operate at 100MB/s.
4. Provide scalable memory bandwidth, based on protocol timing of 25 nanoseconds (ns) per cycle, which scales with improvements in DRAM and static memory (SRAM) access times.
5. Use module and connector technology consistent with Futurebus+ specifications.

The cache coherence protocol of the system bus is designed to support the Alpha AXP architecture and provide each CPU and the I/O bus with a consistent view of shared memory. To satisfy the bandwidth and latency requirements of the processor's instruction issue rate, the processor's second-level cache size, 128-bit access width, and 32-byte block size were optimized to avoid bandwidth limits to performance. The block size and access width were made consistent with the system bus, which satisfied the I/O throughput metrics. Consideration was given to support of a 64-byte block on the 128-bit-wide bus. Such support would have resulted in a 17 percent larger miss penalty and higher average memory access time for the CPU and I/O, more storage and control complexity, and hence higher cost.

Simplicity of the bus protocol was achieved by limiting the number and variations of transactions to four types—read, write, exchange, and null. The exchange transaction enables the second-level cache of the CPU to exchange data, that is, to perform a victim write to memory at the same time as the replacement read transaction. This avoided the coherence complexity associated with a lingering victim block after the replacement read transaction completed.

To address the issue of bandwidth requirements over time as faster processors become available, an estimate of 40 percent bus utilization for each processor with a 1MB second-level cache was obtained from trace-based performance models. The utilization was shown to be reduced by using a 4MB second-level cache or by using larger caches on the DECchip 21064 chip. This approach was reserved as a means to support future CPU upgrades.

Figure 3 is a block diagram of the length-limited seven-slot synchronous system bus. To achieve tight module-to-module clock skew control for this single-phase clock scheme, clocks are radially distributed from the CPU 1 module to the seven slots. This avoided the added cost of a separate module dedicated for radial clock distribution, and enabled the bus arbitration circuitry to be integrated onto the CPU 1 module.

Arbitration of the two CPU modules and the I/O module for the system bus is centralized on the CPU 1 module. To satisfy the I/O module's latency requirements, the arbitration priority allows the I/O module to interleave with each CPU module. In the absence of other requests, a module may utilize the system bus continuously. Shared-memory state evaluations from the bus addresses during continuous bus utilization causes CPU "starvation" from the second-level cache. To avoid CPU starvation from the second-level cache, the arbitration controller creates one free cycle after three consecutive bus transactions.

## Technology Selection

The primary force behind technology selection was to realize the full performance potential of the DECchip 21064 microprocessor with a balanced I/O subsystem, weighted by cost minimization, schedule goals, and operation in an office environment. SPICE analysis was used to evaluate various module and semiconductor technologies. A technology demonstration module was designed and fabricated to correlate the SPICE models and to validate possible technology. Based on demonstrations, the project proceeded with analytical data supported by empirical data.

The 25-watt DECchip 21064 CPU was designed in a 3.3-V, 0.75-micrometer complementary metal-oxide semiconductor (CMOS) technology and was packaged in a 431-pin pin grid array (PGA). The CPU was the only given technology in the system. The power supply, air cooling, and logical and electrical CPU chip interfacing aspects of the CPU module and system bus designs evolved from the DECchip 21064 specifications. System design attention focused on powering and cooling the CPU chip. Compliance with power and cooling specifications was determined to be achievable through conventional voltage regulation and decoupling technology and conventional fan technology.

To address system integrity and reliability requirements, all data transfer interconnects and storage devices had to be protected. The DECchip 21064 CPU's data bus and second-level cache are longword error detection and correction (EDC) protected. The system bus is longword parity protected. The memory subsystem has 280-bit-wide EDC-protected memory arrays. The Futurebus+ is longword parity protected.

### System Bus Clocking

To establish the 25-ns bus cycle time, analog models of the interconnect were developed and analyzed for 5.0-V CMOS transceivers. Assuming an edge-to-edge data transfer scheme, the modelers evaluated the timing from a driver transition to its settled signal, including clock input to driver delay, receiver setup time, and module-to-module clock skew. The cycle time and the data transfer width were combined to determine compliance with low latency and bandwidth. Further analysis revealed that the second-level cache access timing was critical for performing shared-memory state lookups from the bus. One solution to this problem was to store duplicate tag values of the second-level cache. This was evaluated and found to be too expensive to implement. However, the study did show that a duplicate tag store of the CPU's primary data cache had a performance advantage and was affordable if implemented in the CPU module's bus interface unit (BIU) chips.

To evaluate second-level cache access timing, a survey of SRAM access times, density, availability, and cost was taken. Results showed that a 1MB cache using 12-ns access time SRAMs was optimal. With a 12-ns access time SRAM, the critical timing could be managed through the design of the BIU chips. The SRAM survey also showed that a 4MB second-level cache could be planned as a follow-on boost to performance, as SRAM prices declined. Trace-based performance simulations proved that these cache sizes satisfied performance goals of 125 VUPs. This clock rate required a bus stall mechanism to accommodate current DRAM access times in the memory subsystem, which will enable future enhancements as access times are reduced.

The system bus clocks are distributed as positive emitter-coupled level (PECL) differential signals; four single-phase clocks are available to each slot. Each module receives, terminates, and capacitively couples the clock signals into noninverting and inverting PECL-to-CMOS level converters to provide four edges per 25-ns clock cycle. System bus handshake and data transfers occur from clock edge to

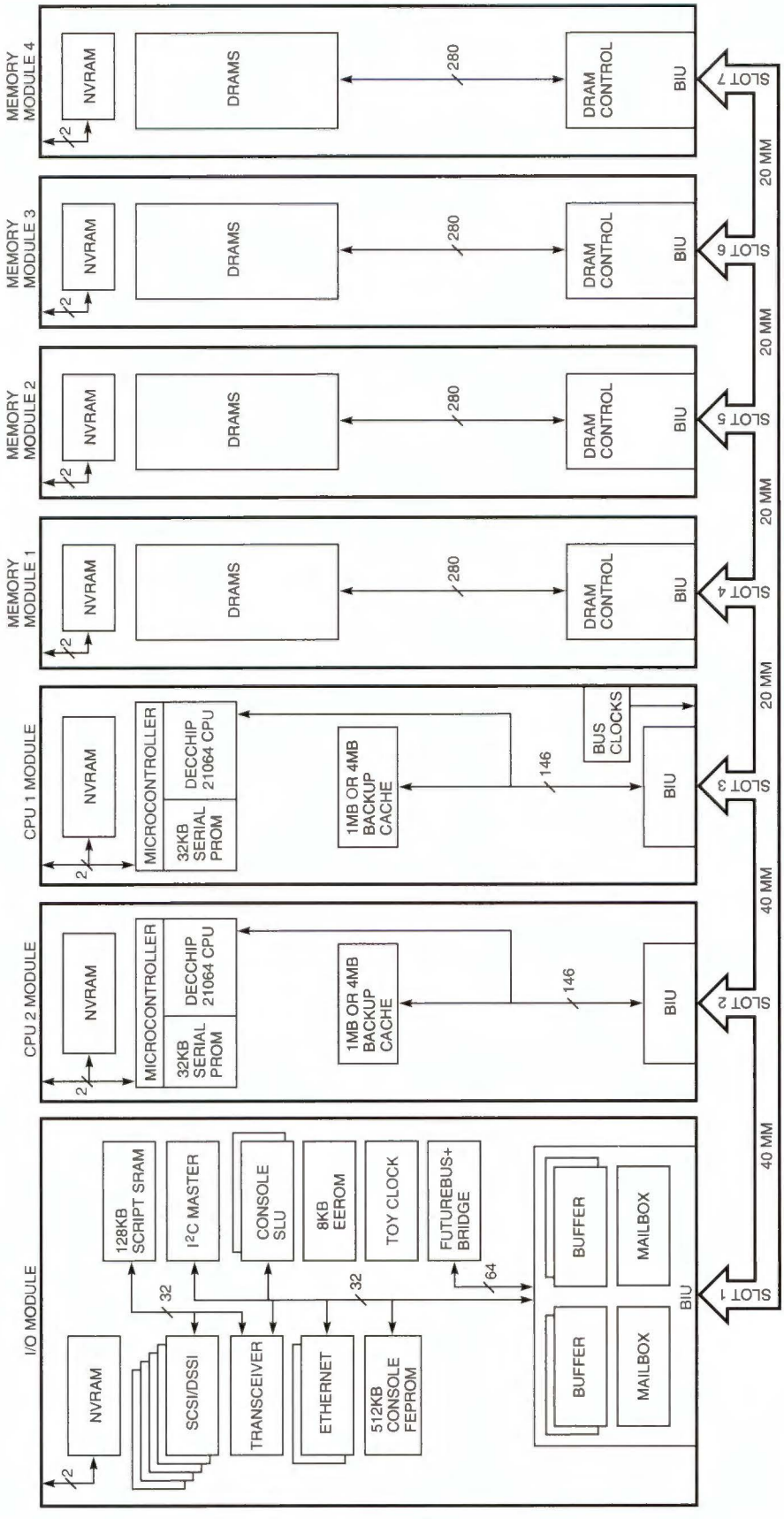



*Figure 3 DEC 4000 AXP System Bus*

clock edge and utilize one of two system bus clocks. A custom clock chip was implemented to provide process, voltage, temperature, and load (PVTL) regulation to the pair of application-specific integrated circuit (ASIC) chips that compose each BIU. The clock chip achieves module-to-module skews of less than 1 ns.

Our search for a clock repeater chip that could minimize module-to-module skew and chip-to-chip skew on a module, and yet directly drive high fan-out ASIC chips with CMOS-level clocks, led us to Digital's Semiconductor Operations Group. Such a chip was in design; however, it was tailored for use at the DEC 6000 system bus frequency. The Semiconductor Operations Group agreed to change the chip to accommodate the DEC 4000 AXP system bus frequency.

## *I/O Bus Technology*

Because of technology obsolescence, I/O buses have a 21-year life cycle divided into 3 phases. During the first 7 years of acceptance, peripherals and applications are developed and supported. Sustained acceptance takes hold in the next 7 years as peripherals and applications are enhanced. In the last 7 years, a phase out or migration of peripherals and applications occurs. For the DEC 4000 AXP systems, our first priority was selection of an open expansion I/O bus in the first third of its life cycle. In addition, we wanted to select an open IEEE standard bus that would attract third-party developers to provide I/O solutions to customers. The following prioritized criteria were established for the selection of a new I/O bus:

1. Open bus that is an accepted industry standard in the beginning third of its life cycle
2. Compatibility with Alpha AXP architecture
3. Minimum data rate of 100MB/s
4. Scalable features that are performance-extensible through architecture (e.g., bus width), and/or through technology improvements (e.g., semiconductor device performance and integration)
5. Minimum 64-bit data path
6. Support of bridges to other I/O buses
7. Minimal interoperability problems between devices from different vendors

After examination of several I/O buses that satisfied these criteria, the Futurebus+ was selected. At the time of our investigation, however, the Futurebus+ specification was in development by the IEEE and a wide range of interest was evident throughout the industry. By providing the right support to the Futurebus+ committee, Digital was in a position to help stabilize and bring the specification to completion.

A Digital team represented the project's interests on the IEEE P896.2 Specification Committee and proposed standards as the DEC 4000 AXP system design evolved. This team achieved its goal by helping the IEEE Committee define a profile that enabled the Futurebus+ to operate as a high-performance I/O expansion bus. To mitigate schedule impact due to instability of the Futurebus+ specifications, the I/O module's Futurebus+ interface was architected to accommodate changes through a more discrete, rather than a highly integrated implementation. Compliance with the Futurebus+ specifications influenced most mechanical aspects of the module compartment design, as is evident from the centerplane, card cage, module construction and size, and power supply voltage specifications and implementations.

## *Module Technology*

Module technology was selected to maximize signal density within the fewest layers with minimal crosstalk and to provide a uniform signal distribution impedance for any module layer. Physical-to-electrical modeling tools were used to create SPICE models of connectors, chip packages, power planes, signal lines of various lengths and impedances (based on the module construction technology), and multiple signal lines. Because the placement of components affects signal performance and quality and system performance (e.g., in the second-level processor cache), module floor plans and trial layouts were completed. A module layout tool was used to ensure producibility compliance with manufacturing standards as well as signal routing constraints. The module layout process was iterative. As sections of the module routing were completed, SPICE models of the etch were extracted. These extracted models were connected to SPICE models of chip drivers and run. Analysis was completed and required changes were implemented and analyzed again. The process continued until the optimal specification conformance was achieved for all signals.

Module size was estimated based on system functionality requirements and a study of the size and power requirements of that functionality. To simplify the enclosure design, module size specifications are consistent with the Futurebus+ module specifications. To achieve lower system costs, the processor, memory, and I/O modules are based on the same ten-layer controlled impedance construction.

Chip engineers avoided the specification of fine-pitch surface-mount chips when possible. Component choices and module layouts were completed with a view toward manufacturability. Cost analysis showed that mixed, double-sided surface-mount components and through-hole components had insignificant added cost when fused tin-lead module technology and wet-film solder-mask technology were used. The required layer construction and impedances of 45, 70, and 100 ohms could easily be achieved within cost goals through this technology. Solder-mask over bare copper technology was also evaluated to determine if fine-pitch surface-mount components achieved higher yield through the solder reflow process. This evaluation showed fused tin-lead technology was better suited, based on defect densities, for the manufacturing process. Consequently, all DEC 4000 AXP modules are implemented with fused tin-lead module technology and wet-film solder-mask technology.

## *Semiconductor Technology*

As a result of a performance, cost, power, and module real estate study, CMOS technology was used extensively. The custom-designed PVTL clock chips were developed in 1.0-micrometer CMOS technology to supply CMOS-level signals for driving directly into the BIU chips. Each module's BIU used the same 0.8-micrometer ASIC technology and die size to closely manage clock skews. Each system bus module's BIU is implemented by two identical chips operated in an even and an odd slice mode. Chip designers invented a method for accepting 5.0-V signals to be driven into their 3.3-V biased DECchip 21064 CPU. Consequently, the selection and implementation of 5.0-V ASIC technology were easier. ASIC vendor selection was based on (1) performance of trial designs and timing analysis of parity and EDC trees, (2) SPICE analysis of I/O drivers with direct-drive input clock cells, and (3) a layout ability to support wide clock trunks and distributed clock buffering to effect low skews.

All memory chips on the CPU module, memory module, and I/O module were implemented in submicron CMOS or BiCMOS technology. All the I/O and power subsystem controller chips such as the SCSI and DSSI controllers, Ethernet controllers, serial line interfaces, and analog-to-digital converters were implemented in CMOS technology.

Speed or high drive is critical in radial clock distribution, Futurebus+ interfacing, or memory module address and control signal fan-out. In these special cases, 100K ECL operated in positive mode (PECL) or BIPOLAR technology was employed.

## *System Bus Protocol and Technology*

The cache coherence protocol for the shared-memory system bus is based on a scheme in which each cache that has a copy of the data from memory also has a copy of the information about it. All cache controllers monitor or snoop on the bus to determine whether or not they have a copy of the shared block. Hence the system bus protocol is referred to as a snooping protocol, and the system bus is referred to as a snooping bus.[4]

The 128-bit-wide synchronous system bus provides a write update 5-state snooping protocol for write-back cache-coherent 32-byte block read and write transactions to system memory address space. Each module uses a 192-pin signal connector—the same connector used by Futurebus+ modules. Each module interfaces between the system bus and its back port with two 299-pin PGA packages containing CMOS ASIC chips, which implement the bus protocol. A total of 157 signals and 35 reference connections implement the system bus in the 192-pin connector (6 interrupt and error, 8 clock and initialization, 128 command and address or data, 4 parity, 11 protocol). All control/status registers (CSRs) are visible from the bus to simplify the data paths as well as to support SMP.

To simplify the snooping protocol, only full block transactions are supported; masking or sub-block transactions occur in each module's BIU. Transactions are described from the perspectives of a commander, a responder, and a bystander. The address space is partitioned into CSR space that cannot be cached, memory space that can be cached, and secondary I/O space for the Futurebus+ and I/O module devices. Secondary I/O space is accessible through an I/O module mailbox transaction, which pends or retries the system bus when access to very slow I/O controller registers conflicts with direct memory access (DMA) traffic. This software-assisted procedure also provides masked byte read and write access to I/O devices as well as a standard

software interface. The use of 32-bit peripheral DMA devices avoided the need to implement hardware address translators. The software drivers provide physical addresses; hence mapping registers are not necessary.

The I/O module drives two device-related interrupt signals that are received by both CPU modules due to SMP requirements. One interrupt is associated with the Futurebus+, and the other is associated with all the device controllers local to the I/O module. The I/O module provides a silo register of Futurebus+ interrupt pointers and a device request register of local device interrupt requests. CPU 1 or CPU 2 is the designated interrupt dispatcher module. Privileged architecture library software subroutines, known as PALcode, run on the primary CPU module and read the device interrupt register or Futurebus+ interrupt register to determine which local devices or which Futurebus+ device handlers are to be dispatched.

The enclosure, power, and cooling subsystems are capable of interrupting both processors when immediate attention is required. A CPU can obtain information from subsystems shown in Figure 2 through the serial control bus. The serial control bus enables highly reliable communications between field replaceable subsystems. During power-up, it is used to obtain configuration information. It is also used as an error-logging channel and as a means to communicate between the CPU subsystem, power subsystem, and the OCP. The nonvolatile RAM (NVRAM) chip implemented on each module allowed the firmware to use software switches to configure the system. The software switches avoided the need for hardware switches and jumpers, field replaceable unit identification tags, and handwritten error logs. As a result, the hardware system is fully configured through firmware, and fault information travels with the field replaceable unit.

The five-state cache coherence protocol assumes that the processor's primary write-through cache is maintained as a subset of the second-level write-back cache. The BIU on the CPU module enforces this subset policy to simplify the simulation verification process. Without it, the number of verification cases would have been excessive, difficult to express, and difficult to simulate and check for correctness. The I/O module implements an invalidate-on-write policy, such that a block it has read from memory will be invalidated and then re-read if a CPU writes to the block. The I/O module participates in the coherency policy by signaling shared status to a CPU read of a block it has buffered. The five states of the cache coherence protocol are given in Table 2.

The cache coherence protocol ensures that only one CPU module can return a dirty response. The dirty response obligates the responding CPU module to supply the read data to the bus, since the memory copy is stale and the memory controller aborts the return of the read data. Bus writes always clear the dirty bit of the referenced cache block in both the commander module and the module that takes the update.

A CPU has two options when a bus transaction is a write and the block is found to be valid in its cache. A CPU either invalidates the block or accepts the block and updates its copy, keeping the block valid. This decision is based on the state of the primary cache's duplicate tag store and the state of the second-level cache tag store. Acceptance of the transaction into the second-level cache on a tag

**Table 2 Five States of the Cache Coherence Protocol**

| State | | Remarks |
|---|---|---|
| 1 | NOT VALID | Block is invalid. |
| 2 | VALID<br>NOT SHARED<br>NOT DIRTY | Valid for read or write, this cached block contains the only copy of the block; the copy is identical to the memory copy. |
| 3 | VALID<br>NOT SHARED<br>DIRTY | Valid for read or write, this cached block contains the only cached copy of the block. The cached copy has been modified more recently than the memory copy. |
| 4 | VALID<br>SHARED<br>NOT DIRTY | Block is valid for read or write, but a write must broadcast to the bus. This block may be in another cache, but the memory copy is identical. |
| 5 | VALID<br>SHARED<br>DIRTY | Block is valid for read or write, but a write must broadcast to the bus. This block may be in another cache, but the contents have been modified more recently than the memory copy. This is a transitional state that occurs when arbitrating for the bus to broadcast a write or when an unshared dirty block is returned to a bus read transaction. |

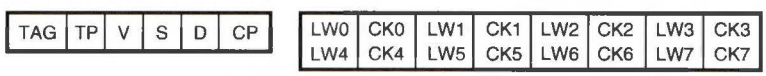
match is called conditional update. When the commander is the I/O module, the write is accepted by a CPU only if the block is valid. Depending on the state of the primary data cache duplicate tag store, two types of hit responses can be sent to an I/O commander–I/O update always and I/O conditional update. In the case of either I/O or CPU commander writes, if the valid block is in the primary data cache, the block is invalidated. The two acceptance modes of I/O writes by a CPU are programmable because accepting writes uses approximately 50 percent more second-level cache bandwidth than invalidating writes.

To implement the cache coherence protocol, the CPU module's second-level cache stores information as shown in Figure 4 for each 32-byte cache block.

Figure 5 shows the cycle timing and transaction sequences of the system bus. Write transactions occur in six clock cycles. Read, null, and exchange transactions occur in seven clock cycles. A null transaction enables a commander to nullify the active transaction request or to acquire the bus and avoid resource contention, without modifying memory. The arbitration controller monitors the bus transaction type and follows the transactions, cycle by cycle, to know when to rearbitrate and signal a new address and command cycle. Additional cycles can be added by stalling in cycle 2 or cycle 4. Transactions begin when the arbitration controller grants the use of the CPU module's second-level caches to a commander module. The controller then signals the start of the address and command cycle 0 (CA). The commander drives a valid address, command, and parity (CAD) in cycle 1. A commander may stall in cycle 2 before supplying write data (WD) in cycles 2 and 3.

Read data (RD) is received in cycles 5 and 6. The addressed responder confirms the data cycles by asserting the acknowledge signal two cycles later. The commander checks for the acknowledgment and, regardless of the presence or absence, completes the number of cycles specified for the transaction. Snooping protocol results are made available half way through cycle 3. As shown in Figure 5, the protocol timing from valid address to response of two cycles is critical. A responder or bystander may stall any transaction in cycle 4 by asserting a stall signal in cycle 3. The bus stalls as long as the stall signal is asserted. Arbitration is overlapped with the last cycle of a transaction, such that tristate conflict is avoided.

A 29-bit lock address register provides a lock mechanism per cache block to assist with software synchronization operations. The lock address register is managed by each CPU as it executes load from memory to register locked longword or quadword (LDx_L) and store register to memory conditional longword or quadword (STx_C) instructions.[5] The lock address register holds an address and a valid bit, which are compared with all bus transaction addresses. The valid bit is cleared by bus writes to a




- TAG consists of 9 physical address bits with a 4MB second-level cache, or 11 physical address bits with a 1MB second-level cache.
- TAG PARITY (TP) bit indicates even parity.
- VALID (V) bit indicates whether or not this block can be considered for a response to the snoop transaction.
- SHARED (S) bit indicates whether or not this block may also be resident in another module's cache.
- DIRTY (D) bit indicates whether or not this block has been modified by this processor.
- CONTROL PARITY (CP) bit indicates even parity.
- DATA (LW) bits organized as two 128-bit-wide half blocks; each 128-bit block is composed of four longwords.
- CHECK (CK0 through CK7) bits detect errors for each longword.

*Figure 4 Second-level Cache Structure*

| WRITE CYCLE | 6 | 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| ARBITRATE | GRANT | | | | | | GRANT | | |
| COMMAND | | CA | | | | | | CA | |
| ADDRESS | | CAD | CAD | | | | | | |
| ACKNOWLEDGE | | | | | CA | WD1 | WD2 | | |
| SHARED | | | | | CAD | | | | |
| DATA | | | | WD1 | WD2 | | | | |

| READ, NULL, EXCHANGE CYCLE | 5 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| ARBITRATE | GRANT | | | | | | | GRANT | |
| COMMAND | | CA | | | | | | | CA |
| ADDRESS | | CAD | CAD | | | | | | |
| ACKNOWLEDGE | | | | | CA | WD1 | WD2 | | |
| SHARED/DIRTY | | | | | CAD | | | | |
| DATA | | | | WD1 | WD2 | | RD1 | RD2 | |

KEY:

| | |
|---|---|
| CA | COMMAND |
| CAD | ADDRESS |
| WD | WRITE DATA |
| RD | READ DATA |

*Figure 5 System Bus Transaction Sequences*

matching address or by CPU execution of STx_C instructions. The register is loaded and validated by a CPU's LDx_L instruction. This hardware and software construct, as a means of memory synchronization, statistically avoids the known problems with exclusionary locking schemes. Exclusionary locking schemes create resource deadlocks, transaction ordering issues, and performance degradation as side effects of the exclusion. This construct allows a processor to continue program execution while hardware provides the branch conditions. The lock fails only when it loses the race on a write collision to the locked block.

A bus transaction address that hits on a valid lock address register must return a snooping protocol shared response, even if the block is not valid in the primary and second-level caches. The shared response forces writes to the block to be broadcast, and STx_C instructions to function correctly. The NULL transaction is issued when a STx_C write is aborted due to the failure of the lock to avoid system memory modification.

## CPU Module Subsystems

Each CPU module consists of a number of subsystems as shown in Figure 3. The CPU module's subsystems are

1. DECchip 21064 processor
2. 1MB or 4MB physically addressed write-back second-level cache
3. BIU chips containing write merge buffers, a duplicate tag store of the processor's 8-kilobyte (KB) data cache for invalidate filtering and write update policy decisions, an arbitration controller, a system bus interface, an address lock register, and CSRs
4. System bus and processor clock generators, clock and voltage detectors, and clock distributors
5. System bus reset control
6. 8KB serial ROM for power-up software loading of the processor
7. Microcontroller (MC) with serial system bus interface and serial line unit for communication with the processor's serial line interface
8. NVRAM chip on the serial control bus

Since a CPU module has to operate in either CPU 1 or CPU 2 mode, the CPU 2 connector was designed to provide an identification code that enables or disables the clock drivers and configures the CSRs'

address space and CPU identification code. As a result, arbitration and other slot-dependent functions are enabled or disabled when power is applied.

A reliability study of a parity-protected second-level cache showed that the SRAMs contributed 44.7 percent of the failure rate. By implementing EDC on the data SRAM portion of the second-level cache, a tenfold improvement in per processor mean time to failure was achieved. Consequently, six SRAM chips per processor were implemented to ensure high reliability.

The multiplexed interface to the second-level cache of the CPU module allows the processor chip and the system bus equal and shared access to the second-level cache. To achieve low-latency memory access, both the microprocessor and the system bus operate the second-level cache as fast as possible based on their clocks. Hence the second-level cache is multiplexed, and ownership defaults to the microprocessor. When the system bus requires access, ownership is transferred quickly with data SRAM parallelism while the tag SRAMs are monitored.

Many of the CPU module subsystems are found in the interface gate array called the $C^3$ chip. Two of these chips working in tandem implement the BIU and the second-level cache controller. Write merge buffers combine masked write data from the microprocessor with the cache block as part of an allocate-on-write policy. Since the microprocessor has write buffers that perform packing, full block write around the second-level cache was implemented as an exception to the allocate-on-write policy. To meet schedule and cost goals with few personnel, one complex gate array was designed rather than several lower-complexity gate arrays. Hence the data path and the control functions were partitioned such that the microprocessor could operate as an even or odd member of a pair on the CPU 1 or the CPU 2 module.

The system bus clock design is somewhat independent of the processor clock, but the range is restricted due to the implementation of the snooping protocol timing, the multiplexed usage of the second-level cache, and the CPU interface handshake and data timing. Therefore, the system bus cycle time is optimized to provide the maximum performance regardless of the processor speed. Likewise, the processor's cycle time is optimized to provide maximum performance regardless of the bus speed. Considerable effort resulted in a second-level cache access time that enabled the CPU's read or write accesses to complete in four internal clock cycles, called the four-tick loop timing of the second-level cache. To realize both optimizations, the CPU's synchronous interface is supported by an asynchronous interface in the BIU. Various timing relationships between the processor and the system bus are controlled by programmable timing controls in the BIU chips.

To achieve the tight, four-tick timing of the second-level cache, double-sided surface-mount technology was used to place the SRAM chips physically close together. This minimized address wire length and the number of module vias; hence the driver was loaded effectively. This careful placement was combined with the design of a custom CMOS address fan-out buffer and multiplexer chip (CAB) to achieve fast propagation delays. The CAB chip was implemented in the same CMOS process as the DECchip 21064 CPU to obtain the desired throughput delay. Combined with 12-ns SRAM chips, the CAB chip enabled optimization of the CPU's second-level cache timing as well as the system bus snooping protocol response timing.

## I/O Module, Mass Storage, and Expansion I/O Subsystems

The I/O module consists of a local I/O subsystem that interfaces to the common I/O core and a bridge to the Futurebus+ for I/O options. By incorporating modularity into the design, a broad range of applications could be supported. To satisfy the disk performance and bulk storage metrics given in Table 1, mass storage was configured based on applications requirements. Fast access times of 3.5-inch disks and multiple spindles were selected for applications with results in QIO/s. The density of 5.25-inch disks was selected for applications requiring more storage space. As indicated in Table 1, the metrics of greater than 4,000 QIO/s determined the performance requirements of the storage compartment. Each of the four disk storage compartments in the system enclosure can hold a full-size 5.25-inch disk if cost-effective bulk storage is needed. If the need is for the maximum number of I/Os per second, each compartment can hold up to four 3.5-inch disks in a mini array.

Configurations of 3.5-inch disks in a brick enable optimization of throughput through parallelism techniques such as stripe sets and redundant array of inexpensive disks (RAID) sets. The brick configuration also enables fault tolerance, at the expense of throughput, by using shadow sets. With

this technique, each storage compartment is interfaced to the system through a separate built-in controller. The controller is capable of running in either DSSI mode for high availability storage in cluster connections with other OpenVMS AXP or VMS systems, or in SCSI mode for local disk storage available from many different vendors. For applications in which a disk volume is striped across multiple drives that are in different storage cavities, the benefit from the parallel seek operations of the drives combines with the parallel data transfers provided by the multiple bus interfaces. The main memory capacity of the system allows for disk caching or RAM disks to be created, and the processing power of the system can be applied to managing the multiple disk drives as a RAID array. With current technology, maximum fixed storage is 8GB with 5.25-inch disks and 24GB with 3.5-inch disks. If the built-in storage system is inadequate, connection to an external solution can occur through the Futurebus+.

The BIU is implemented by two 299-pin ASIC chips. The bridge to the Futurebus+ and the interface to the local I/O devices are provided with separate interfaces to the system bus. Each interface contains two buffers that can each contain a hexword of data. This allows for double buffering of I/O writes to memory for both interfaces and for the prefetching of read data by which the bridge improves throughput. These buffers also serve to merge byte and longword write transaction data into a full block for transfer over the system bus. In this case, the write to main memory is preceded by a read operation to merge modified and unmodified bytes within the block.

The Ethernet controllers and SCSI and DSSI controllers can handle block transfers for most operations, thus avoiding unnecessary merge transactions. As shown in Figure 3, the I/O module integrates the following:

1. Four storage controllers that support SCSI, high-speed SCSI, or DSSI for fixed disk drives and one SCSI controller for removable media drives
2. 128KB of SRAM for disk-controller-loadable microcode scripts
3. Two Ethernet controllers and their station address ROMs, with switch-selectable ThinWire or thick-wire interfaces
4. 512KB flash erase programmable ROM (FEPROM) for console firmware
5. Console serial line unit (SLU) interface
6. Auxiliary SLU interface with modem control support
7. Time-of-year (TOY) clock, with battery backup
8. 8KB of electrically erasable memory (EEROM) for console firmware support
9. Serial control bus controller and 2 kilobits of NVRAM
10. 64-bit-wide Futurebus+ bridge
11. BIU, containing four hexwords of cache block buffering, two mailbox registers, and the system bus interface

The instability of the Futurebus+ specifications and the use of new, poorly specified controller chips presented a high design risk for a highly integrated implementation. Therefore the Futurebus+ bridge and local I/O control logic were implemented in programmable logic to isolate the high risk design areas from the ASIC development process.

## *Memory Subsystem*

As shown in Figure 3, up to four memory modules can reside on the system bus. This modularity of the memory subsystem enabled maximum configuration flexibility. Based on the metrics listed in Table 1, 2GB of memory were expected to satisfy most applications requirements. Given this 2GB design goal, the available DRAM technology, and the module size, the total memory size was configured for various applications.

The memory connectors provide a unique slot identification code to each BIU, which is used to configure the CSRs' address space based on the slot position. Memory modules are synchronous to the system bus and provide high-bandwidth, low-latency dynamic storage. Each memory module uses 4-bit-wide, 1- and 4-megabit-deep DRAM technology in various configurations to provide 64MB, 128MB, 256MB, or 512MB of storage on each module.

To satisfy memory performance goals, each memory module is capable of operating alone or in one of numerous cache block interleaving configurations with other memory modules with a read-stream capability. A performance study of stream buffers revealed an increase in performance from memory-resident read-stream buffers. The stream buffers allow each memory module to reduce the

average read latency of a CPU or I/O module. Thus more bandwidth is usable on a congested bus because the anticipated read data in a detected access sequence is prefetched. The stream buffer prefetch activity is statistically determined by bus activity.

Overall memory bandwidth is also improved through exchange transactions, which use victim writes with replacement read parallelism. Intelligent memory refresh is scheduled based on bus activity and anticipated opportunities. Write transactions are buffered from the bus before being written into the DRAMs to avoid stalling the bus.

Data integrity, memory reliability, and system availability are enhanced by the EDC circuitry. Each memory module consists of 2 or 4 banks with 70 DRAM chips each. This enables 256 data bits and 24 EDC bits to be accessed at once to provide low latency for the system bus. A cost-benefit analysis showed a savings of DRAM chips when EDC is implemented on each memory module. The processor's 32-bit EDC requires 7 check bits as opposed to the 128-bit EDC, which requires 12 check bits and uses fewer chips per bank. The selected EDC code also provides better error detection capability of 4-bit-wide chips than the processor's 32-bit EDC.

To improve performance, separate EDC logic is implemented on the write path and read path of each memory module's BIU. The EDC logic performs detection and correction of all single-bit errors and most 2-bit, 3-bit, and 4-bit errors in the DRAM array. The EDC's generate function can detect certain types of addressing failures associated with the DRAM row and column address bits, along with the bank's select address bits. Failures associated with these addressing fields can be detected, thus improving data integrity. Software errors can be scrubbed from memory by the CPU when requested through use of PALcode subroutines, which use the LDx_L and STx_C synchronization construct without having to suspend system operations.

## Enclosure and Power Subsystems

The DEC 4000 AXP enclosure seen in Figure 1 is called the BA640 box and is 88.0 centimeters (cm) high, 50.6 cm wide, and 76.2 cm deep. It weighs 118 to 125 kilograms fully configured. The enclosure is designed to operate in an office environment from 10 to 35 degrees Celsius. The power cord can connect to a conventional wall outlet which supplies up to 20 amperes at either 120 V AC or 240 V AC.

The DEC 4000 AXP system is a portable unit that provides rear access and simplified installation and maintenance. The system is mounted on casters and fits easily into an open office environment. Modular design allowed compliance with standards, ease of manufacturing, and easy field servicing. Constructed of molded plastics, the chassis is sectioned into a card cage, a storage compartment, a base containing four 6-inch variable-speed DC fans and casters, an air plenum and baffle assembly, front and rear doors, and side panels. The mass storage compartment supports up to 16 fixed-storage devices and 4 removable storage devices. Expansion to storage enclosures is supported for applications that require specialized storage subsystems.

Feedback from field service engineers prompted us to omit useless light-emitting devices (LEDs) in each subsystem, since access to most electronics is from the rear. As a result, the OCP was made common to all subsystems through the serial control bus and made visible inside the front door of the enclosure. It provides DC on/off, halt, and restart switches, and eight LEDs, which indicate faults of CPU, I/O, memory, and Futurebus+ modules. The fault lights are controlled either by a microcontroller on either CPU module or by an interface on the I/O module.

Futurebus+ slot spacing was provided by the IEEE specification. The system bus slot spacing for each module was determined by functional requirements. For example, the CPU module requires 300 linear feet of air flow across the DECchip 21064 microprocessor's 3-inch square heat sink, as seen in Figure 1, to ensure the 25-watt chip could be cooled reliably. Since VAX 4000 systems provide this same air flow across modules, cooling was not a major design obstacle. The module compartment's Futurebus+, system bus, and power subsystems can be seen in the enclosure back view of Figure 6.

All electronics in the enclosure, as shown in Figure 7, are air cooled by four 6-inch fans in the base. Air is drawn into the enclosure grill at the top front, guided along a plenum and baffle assembly and down through the module compartment and power supply compartment to the base. Air is also drawn through front door louvers and across the storage compartments and down to the base. Electronics connected to the power subsystem monitor ambient and module compartment exhaust temperatures. Thus the fan speed can be regulated based on temperature measurements,

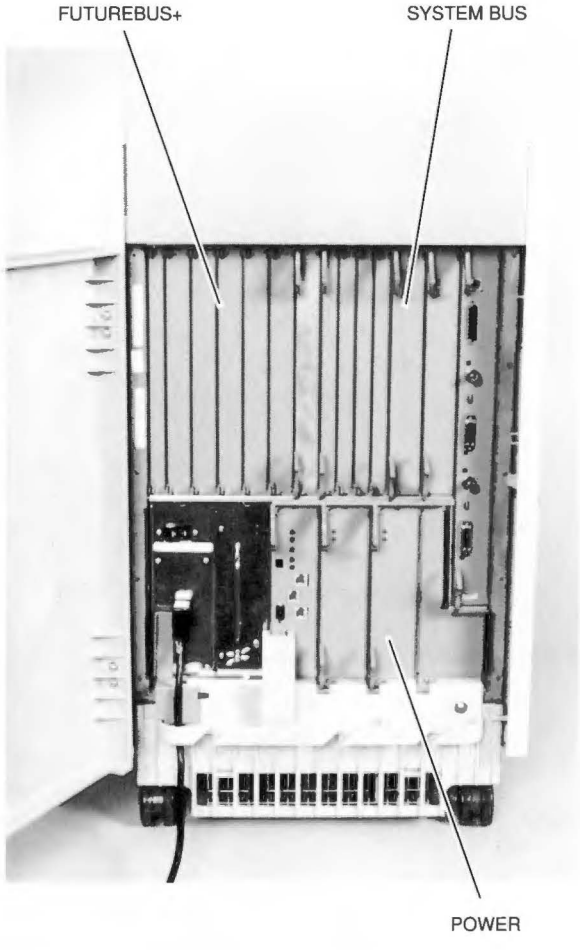



*Figure 6 DEC 4000 AXP Enclosure Rear View*

reducing acoustic noise in an air-conditioned office environment.

The centerplane assembly consists of a storage backplane, a module backplane, and an electromagnetic shield. This implementation avoids dependence on cable assemblies, which are unreliable and difficult to install and repair. Defined connectors and module sizes allowed the enclosure development to proceed unencumbered by module specification changes. The shielded module compartment provides effective attenuation of signals up to 5 gigahertz. There are six Futurebus+ slots, four memory slots, two CPU slots, one I/O slot, and four central power module slots, which include the FEU, PSC, DC5, and DC3 units.

The storage compartment contains six cavities, as seen in the enclosure front view of Figure 8. Two cavities are for removable media, and four are for fixed storage bricks. A storage brick consists of a base plate and mounting hardware, disk drives, local disk converter (LDC), front bezel assembly, and wiring harnesses. The LDC converts a distributed 48.0 V to 12.0-V and 5.0-V supplies and a 5.0-V termination reference for the brick to ensure compliance with voltage regulation specifications and termination voltage levels of current and future disks.

The 20-ampere power subsystem can deliver 1,400 watts of DC power divided across 2.1 V, 3.3 V, 5.0 V, 12.0 V, and 48.0 V. The enclosure can cool 1,500 watts of power and can be configured as a master or a slave of AC power application. Use of a universal FEU eliminates the need for selecting operating voltages of 120 V or 240 V AC. The FEU converts the input AC into 385 V DC, which is distributed to provide 48 V DC to two step-down DC-to-DC converters, which work in parallel to share the load current. The combined 48 V DC output from these converters is delivered to the rest of the system.

Control of distributed power electronics is difficult and expensive with dedicated electronics. A cost-effective alternative was use of a one-chip CMOS microcontroller, surrounded with an array of sensor inputs through CMOS analog-to-digital converters, to provide PSC intelligence. Decision-making ability in the power subsystem enabled compliance with voltage-sequencing specifications and fail-safe operation of the system. The microcontroller can control each LDC and communicate with the CPU and OCP over the serial control bus. It monitors over and under voltage, temperature, and energy storage conditions in the module and storage compartments. It also reports status and failure information either to the CPU or to a display on the PSC module, which is visible inside the enclosure back door.

## Firmware

The primary goal of the console interface is to bootstrap the operating system through a process called boot-block booting. The console interface runs a minimal I/O device handler routine (boot primitive) to read a boot block from a device that has descriptors. The descriptors point to the logical block numbers where the primary bootstrap program can be found, and the console interface loads it into system memory. To accomplish this task, the firmware must configure and

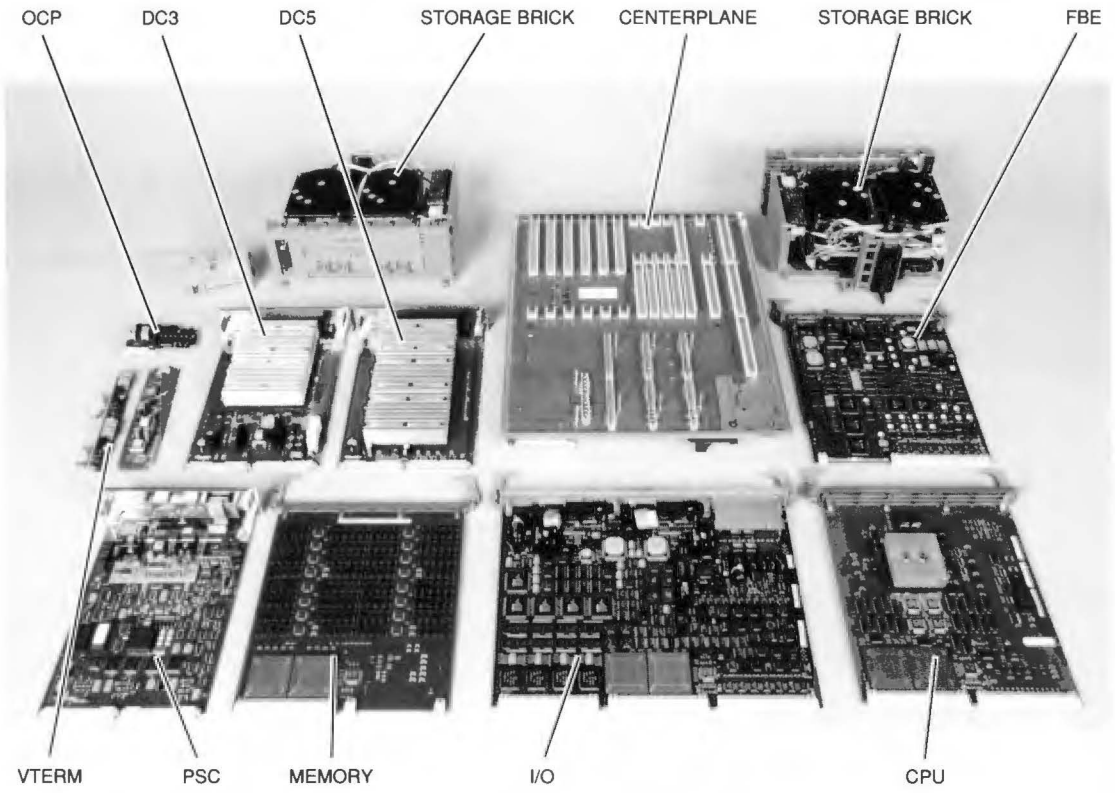



*Figure 7 DEC 4000 AXP Modular Electronics*

test the whole system to ensure the boot process can complete without failures. To minimize the bootstrap time, a fast memory test executes in the time necessary to test the largest memory module, regardless of the number of memory modules. If failures are detected after configuration is completed, the firmware must provide a means for diagnosis, error isolation, and error logging to facilitate the repair process. The DEC 4000 AXP system provides a new console command interface as well as integrated diagnostic exercisers in the loadable firmware.

The firmware resides on two separate entities, a block of serial ROM on the CPU module and a block of FEPROM on the I/O module. The serial ROM contains software that is automatically loaded into the processor on power-up or reset. This software is responsible for initial configuration of the CPU module, testing minimal module functionality, initializing enough memory for the console, copying the contents of the FEPROM into this initialized console memory, and then transferring control to the console code.

The FEPROM firmware consists of halt dispatch, entry/exit, diagnostics, system restart, system bootstrap, and console services functional blocks.

PALcode subroutines provide a layer of software with common interfaces to upper levels of sofware. PALcode serves as a bridge between the hardware behavior and service requirements and the requirements of the operating system. The system takes advantage of PALcode for hardware-level interrupt handling and return, security, implementation of special operating system kernel procedures such as queue management, dispatching to the operating system's special calls, exception handling, DECchip 21064 virtual instruction cache management, virtual memory management, and secondary I/O operations. Through a combination of hardware- and software-dependent PALcode subroutines, OpenVMS AXP, DEC OSF/1 AXP, and future operating systems can execute on this hardware architecture.

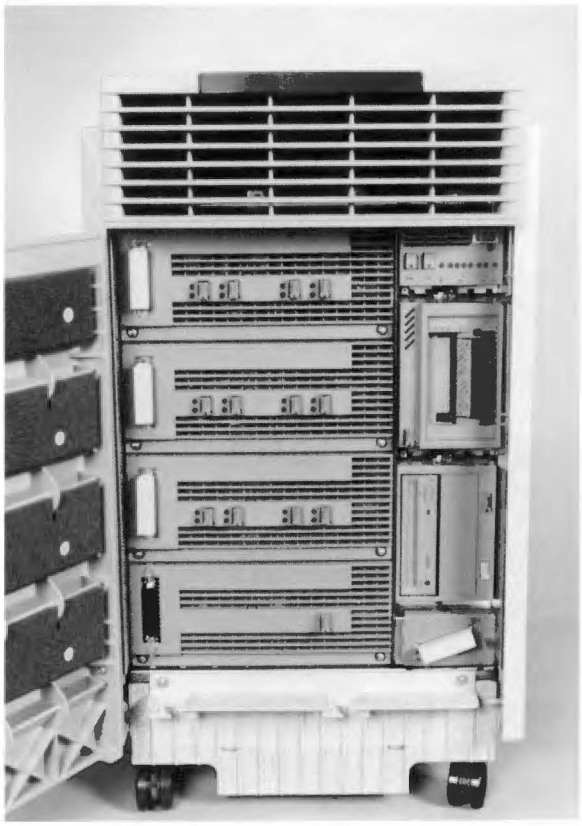


*Figure 8 DEC 4000 AXP System Enclosure Front View*

## Performance Summary

The DEC 4000 AXP Model 610 system's performance numbers as of November 10, 1992 are given in Table 3. Its performance will continue to improve.

## Summary

DEC 4000 AXP systems demonstrate the highest performance and functionality for Digital's 4000 series of departmental server systems. Based on Digital's Alpha AXP architecture and the IEEE's Futurebus+ profile B standard, the systems provide symmetric multiprocessing performance for OpenVMS AXP and DEC OSF/1 AXP operating systems in an office environment. The DEC 4000 AXP systems were designed to optimize the cost-performance ratio and to include upgradability and expandability. The systems combine Digital's CMOS technology, I/O peripherals technology, ahigh-performance multiprocessing backplane interconnect, and modular system design to supply the most advanced functionality for performance-driven applications.

## Acknowledgments

Development of a new system requires contributions from individuals throughout the corporation. The authors wish to acknowledge those who contributed to the key aspects of the DEC 4000 AXP system. Centerplanes: Henry Enman, Jim Padgett; CPU: Nitin Godiwala, George Harris, Jeff Metzger, Eugene Smith, Kurt Thaller; Firmware: Dave Baird, Harold Buckingham, Marco Ciaffi, John DeNisco, Charlie Devane, Paul LaRochelle, Keven Peterson; Futurebus Exerciser: Philippe Klein, Kevin Ludlam, Dave Maruska; Futurebus+: Barbara Archinger, Ernie Crocker, Jim Duval, Sam Duncan, Bill Samaras; I/O: Randy Hinrichs, Tom Hunt, Sub Pal, Prasad Paranjape, Chet Pawlowski, Paul Rotker, Russ Weaver; Management: Jesse Lipcon, Gary P. Lidington; Manufacturing: Mary Doddy, Al Lewis, Allan Lyall, Cher Nicholas; Marketing: Kami Ajgaonkar, Charles Monk, Pam Reid; Mechanical: Jeff Lewis, Dave Moore, Bryan Porter, Dave Simms; Memory: Paul Goodwin, Don Smelser, Dave Tatosian; Operations: Jeff Kerrigan; Operating Systems: AJ Beaverson, Peter Smith; Power: John Ardunio, Robert White; Simulation: Paul Kinzelman; Systems: Vince Asbridge, Mike Collins, Dave Conroy, Al Deluca, Roger Gagne, Tom Orr, Eric Piip; Thermal: Steve Cieluch, Sharad Shah.

## References and Note

1. *IEEE Standard for Futurebus+—Physical Layer and Profile Specification* IEEE Standard P896.2-1991 (New York: The Institute of Electrical and Electronics Engineers, April 24, 1992).

2. Supercomputer performance as defined by the composite theoretical performance (CTP) rating of 397, with the DECchip 21064 operated at 6.25 ns, as established by the U.S. export regulations.

3. *Inter-Integrated Circuit Serial Bus Specification* ($I^2C$ Bus Specification), (Sunnyvale, CA: Signetics Company, 1988).

4. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990): 467–474.

5. R. Sites, ed., *Alpha AXP System Reference Manual,* Version 5.0 (Maynard: Digital Equipment Corporation, 1992).

**Table 3 CPU Performance Summary for the DEC 4000 AXP System**

| Futurebus+ Performance | | |
|---|---|---|
| | **Latency** | **Bandwidth** |
| Peak | 16 bytes/100 ns | 160MB/s |
| Read | 16 bytes/182 ns | 88MB/s |
| Write | 16 bytes/133 ns | 120MB/s |
| **Local Bus Performance** | | |
| | **Latency** | **Bandwidth** |
| Peak | 4 bytes/80 ns | 50MB/s |
| Read | 4 bytes/160 ns | 25MB/s |
| Write | 4 bytes/160 ns | 25MB/s |
| **System Bus Performance** | | |
| | **Latency** | **Bandwidth** |
| Peak | 16 bytes/25 ns | 640MB/s |
| Read | 32 bytes/175 ns | 182MB/s |
| Write | 32 bytes/150 ns | 213MB/s |
| Exchange | 64 bytes/175 ns | 365MB/s |
| **Internal Cache Miss, Second-level Cache Hit (Four-tick) Performance** | | |
| | **Latency** | **Bandwidth** |
| Read | 16 bytes/25 ns | 640MB/s |
| Write | 16 bytes/25 ns | 640MB/s |
| **CPU Second-level Cache Miss Performance** | | |
| | **Latency** | **Bandwidth** |
| Read | 32 bytes/275 ns | 116MB/s |
| Write | 32 bytes/200 ns | 160MB/s |
| Exchange | 64 bytes/275 ns | 232MB/s |

**DEC 4000 Model 610 SPECmark89 and SPECthruput89* Estimated CPU Performance Summary**

| **Integer (INT) Benchmarks** | **Ratio** | **Ratio** | **Ratio** |
|---|---|---|---|
| GCC | 61.58 | 1@ 54.80 | 2@ 50.78 |
| ESPRESSO | 82.91 | 1@ 81.76 | 2@ 78.33 |
| LI | 93.05 | 1@ 92.19 | 2@ 92.18 |
| EQNTOTT | 103.46 | 1@ 100.76 | 2@ 97.94 |
| **Floating-point (FP) Benchmarks** | | | |
| SPICE2G6 | 72.58 | 1@ 68.19 | 2@ 64.95 |
| DODUC | 113.81 | 1@ 113.53 | 2@ 108.95 |
| NASA7 | 229.27 | 1@ 221.56 | 2@ 197.80 |
| MATRIX300 | 1019.17 | 1@ 963.81 | 2@ 948.66 |
| FPPPP | 180.32 | 1@ 177.89 | 2@ 175.83 |
| TOMCATV | 128.70 | 1@ 123.25 | 2@ 105.90 |
| SPECmark > | 136.23 | SPECthruput > 1@ 131.18 | 2@ 124.40 |
| SPECint > | 83.73 | SPECintthruput > 1@ 80.32 | 2@ 77.41 |
| SPECfp > | 188.45 | SPECfpthruput > 1@ 181.92 | 2@ 170.68 |

| | | |
|---|---|---|
| LINPACK – double precision | 100 × 100 | 36.8 MFLOPS |
| LINPACK – double precision | 1000 × 1000 | 78.4 MFLOPS |
| Dhrystone | | 165.0 MIPS |

Note:
*Version 1.0 OpenVMS AXP operating system, 160-MHz clocked DECchip 21064 microprocessor, 1MB second-level cache. Notice the 1.9 scaling of the second CPU.

*Brian R. Allison*
*Catharine van Ingen*

# *Technical Description of the DEC 7000 and DEC 10000 AXP Family*

*The DEC 7000 and DEC 10000 products are mid-range and mainframe Alpha AXP system offerings from Digital Equipment Corporation. These machines were designed to meet the needs of large commercial and scientific applications and therefore are high-performance, expandable systems that can be easily upgraded. The DEC 7000 and 10000 systems utilize the DECchip 21064 microprocessor operating at speeds up to 200 MHz. The high-speed chips, large caches, multiprocessor system architecture, high-performance backplane interconnect, and large memory capacity combine to create mainframe-class performance with a cost and size previously attributed to mid-range systems.*

The design of the DEC 7000 and 10000 systems provides a high-end platform and system environment for multiple generations of Alpha AXP chips. This platform, combined with a multiprocessor architecture, yields a multidimensional upgrade capability that will allow the system to meet users' needs for several years. System upgrade can take place by adding processors, replacing existing processors with next-generation processors, or both. This upgrade capability ensures stability to the system in terms of the physical and fiscal aspects of the end user's computing environment.

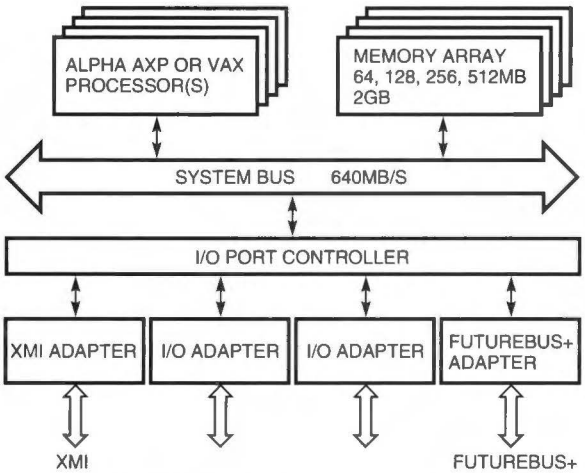
The DEC 7000 and DEC 10000 systems are the logical follow-on products of the highly successful VAX 6000 family.[1] The new systems are capable of supporting either VAX processors or Alpha AXP processors. The capability to upgrade from a VAX processor to an Alpha AXP processor without changes to the system is essential for minimal disruption of large commercial applications. Most features of the VAX 6000 systems have been carried forward to the DEC 7000 and DEC 10000 products, and any deficiencies have been corrected.

The DEC 7000 and DEC 10000 products are derived from the same system design. The DEC 10000 is a more fully configured system and includes an $n+1$ uninterruptible power system, additional I/O subsystems, and I/O expansion cabinets. The DEC 7000 uses a 182-megahertz (MHz) DECchip 21064 whereas the DEC 10000 uses a 200-MHz DECchip 21064.

A very important goal for the project that encompassed the development of the DEC 7000 and 10000 systems was to provide a similar pair of systems based on a VAX microprocessor. A VAX microprocessor, called NVAX+, was designed to be pin compatible with the DECchip 21064 (the Alpha AXP microprocessor).[2,3] The system was designed to be somewhat microprocessor independent, and both VAX and Alpha AXP versions of the systems were implemented. The VAX products (VAX 7000 and VAX 10000) were introduced in July 1992 and can be upgraded to DEC 7000 and DEC 10000 systems by a simple swap of CPU modules.

## *System Architecture*

The DEC 7000 system consists of CPU(s), memory, an I/O port controller, and I/O adapters, as shown in Figure 1. The system is configured in a variety of ways, depending on the size and function of the system. A system backplane consists of nine slots and houses CPUs, memory, and an I/O port controller. The I/O port controller resides in a fixed slot, and CPUs and memories occupy the remaining eight slots. The initial system offerings allow up to 6 CPUs. (Architecturally, the system may support up to 16 CPUs.) Up to 14 gigabytes (GB) of memory can be supported if only 1 CPU module is present and all remaining slots contain memory.




Note: All four I/O ports are identical. Any combination of XMI, Futurebus+, or "custom" interfaces may be configured.

*Figure 1 DEC 7000 and DEC 10000 System Architecture*

The I/O subsystem consists of an I/O port controller and four I/O ports which have been adapted to the XMI or the Futurebus+. The I/O ports are generic and may be adapted to other forms of interconnect in the future. The system backplane, power system, and up to two I/O backplanes are housed in the system cabinet. Additional I/O backplanes (up to a system total of four) may be configured in expansion cabinets.

## Technology

The DEC 7000 system is built primarily of CMOS (complementary metal-oxide semiconductor) components. The DECchip 21064 microprocessor is built using Digital's 0.75-micrometer CMOS-4 process. All modules utilize LSI Logic LCA100K series gate arrays for the system bus interface and for on-board logic functions. The LSI Logic LCA100K features up to 235K two-input NAND gates. All modules use the same custom I/O driver circuit within their respective gate arrays to drive and receive the system bus. A custom 419-pin pin grid array (PGA) package was developed to house all bus interface gate arrays. Unlike the VAX 6000 series, a common bus driver part is not used in order to minimize the number of levels of buffering in the system.

Module technology is standard 10-layer construction with 4 signal layers, 4 power layers, and top and bottom cap layers. Double-side, surface-mount construction is used extensively throughout the system. Etch width is 5 mils with 7.5-mil minimum spacing. Via sizes down to 15 mils are used. A mixture of physical component technologies is used with all large VLSI (very large-scale integration) parts in 100-mil PGA packages. Most standard logic utilizes 50-mil surface-mount technology. Module interconnect to the backplane is made through a 340/420-connection, four-row, 100-mil-spaced pin and socket type connector. Forty-eight-volt power is distributed throughout the system; local regulation is provided on the module for specific voltages required.

## System Interconnect

The heart of the DEC 7000 system is a high-performance system interconnect, called the LSB, which allows communications between multiple processors, memory arrays, and I/O subsystems. It provides a low-latency, high-bandwidth data path among all components. A common shared view of memory is maintained by means of the system interconnect and cache logic on processor modules.

Three types of modules are defined for the LSB.

- Processor modules, which contain the CPU chip, cache subsystem, and console functions. The initial DEC 7000 design has the capacity for a maximum of six processor modules.
- Memory modules, which contain dynamic random-access memory (DRAM) chips and a memory controller. A system can contain up to seven memory modules, each with a capacity of 64 megabytes (MB) to 2GB.
- I/O interface modules, which provide access to I/O buses and I/O adapters. Only a single I/O port controller module may reside in the system. The I/O port controller module can arbitrate at a higher priority than CPU nodes to improve I/O direct memory access (DMA) latency and provide atomic DMA writes of data less than a cache block in size.

The LSB is a limited-length, non-pended, pipelined, synchronous, 128-bit-wide bus with distributed arbitration. All transactions occur in a set of fixed cycles relative to an arbitration cycle. Up to three transactions can be in the pipeline at a given time, enabling the full capability of the bus to be realized. Arbitration occurs on a dedicated set of control signals and may be overlapped with data transfer. Data and address are multiplexed on the same set of signals. The bus protocol supports

write-back caches, and all memory transfers are 64 bytes in length. The cycle time of the bus is 20 nanoseconds (ns), providing an overall data rate of 800MB per second and a utilized system bandwidth of 640MB per second.

The LSB transmits 40-bit physical addresses, providing a physical address space of 1 terabyte. Given the current rate of DRAM technology evolution, the LSB will have a useful life of 8 to 10 years before physical address space is exhausted. A 40-bit physical address was chosen to minimize the data path width in the processor bus control gate array.

A non-pended pipelined bus was chosen instead of a traditional pended bus to allow for simple node interface designs. Transactions start and finish at precisely defined times. A "stall" function may be used if a given transaction cannot be completed within the system timing constraints. The "stall" function freezes the bus pipeline, maintaining the order of all transactions. Consequently, nodes can be designed with no queuing between the bus interface and local storage (DRAMs for main memory or static RAMs [SRAMs] for cache memory). The maintenance of strict bus transaction ordering also alleviates many potential lockout conditions experienced on pended buses.

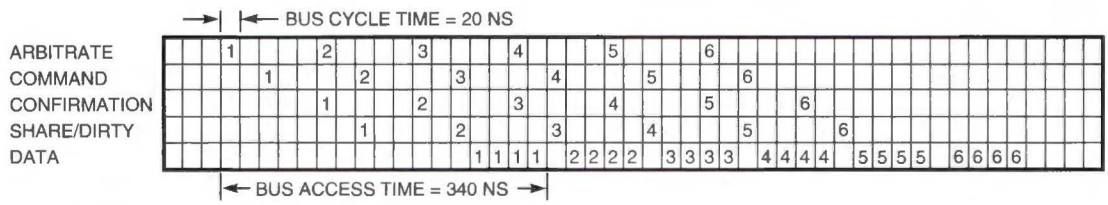
Digital's previous mainframe systems have used a switch-based system interconnect instead of a bus. This interconnect was typically required because these systems were based on emitter coupled logic (ECL) with only a small, single-level cache subsystem; therefore, high bandwidth was required between main memory and the processor. The CMOS design of the DEC 7000 allows a large (4MB) second-level cache to complement the 16-kilobyte (KB) on-chip cache. The large amount of cache minimizes the need for memory bandwidth. A bus-based design was chosen over a switch-based design to minimize memory latency, minimize design complexity, and reduce system cost.

All LSB transactions consist of a single command cycle and four data cycles. These five cycles appear in fixed cycles relative to the arbitration cycles. Up to three transactions may be pipelined, as shown in Figure 2.

The LSB uses a distributed arbitration scheme. Ten request wires are driven by the CPUs or the I/O module that wishes to use the bus. Eight request lines are allocated to the eight potential CPU modules. The remaining two request lines are used by the I/O controller module. All modules independently monitor the request wires to determine whether a transaction has been requested, and if so, which module wins the right to send a command cycle to start the transaction.

The arbitration scheme employs a least-recently-used rotating priority algorithm for CPU modules and a fixed high/low scheme for the I/O port controller. The I/O port controller arbitrates using the highest and lowest priority levels, arbitrating high six times then low two times. This arrangement ensures that the I/O port controller can utilize greater than 50 percent of the available system bus bandwidth while still ensuring the CPUs some access to the system bus. The I/O port controller also uses its unique arbitration scheme to ensure atomic read/modify/write sequences on the bus necessary for performing writes of less than a full naturally aligned 64-byte quantity. The I/O port controller does the read at its next scheduled priority and then immediately follows up with the write at highest priority. This scheme ensures that no other node can access the data between the read and the write.

All command/address and control/status register (CSR) cycles are protected with parity. Data cycles to and from memory are protected with error correction code (ECC). Transmit check is used by all modules to verify that what a given module is asserting on the bus is actually being seen on the




Bus Data Rate = 16 bytes per 20 ns = 800MB/s
Utilized Bus Bandwidth = 16 bytes per 20 ns × 4 data cycles per 5 bus cycles = 640MB/s

*Figure 2 LSB Transaction Pipeline*

bus. Transmit check allows the detection of bus collisions and faulty bus drivers or receivers.

The system interconnect is physically implemented as a centerplane which is 350 millimeters (mm) wide and 500 mm high. There are four module connections on one side, and five on the other. The centerplane-module connection is implemented using a four-row pin and socket connector with connections on a 100-mil grid. Modules are 410 mm high and 340 mm deep. This module size was chosen to allow the maximum module size within the constraints of an 865-mm-deep cabinet and of the centerplane technology. Modules are spaced on 65-mm centers and are contained within a box that provides customized air flow for each different module design.

The DEC 7000 was designed with a centerplane interconnect to solve the problem of bus length and to meet the need for wide module spacing that allows for the anticipated heat-dissipation requirements of future processor chips. With a centerplane, the number of module slots available for a given length of bus increases by $(n \cdot 2)-1$ where $n$ is the number of slots available in a conventional backplane. A centerplane configuration leaves little space on the backplane for termination networks. Designers solved this problem by adopting a distributed termination scheme with bus terminator networks present on all modules in the backplane.

## Processor Module

The primary purpose of the processor module is to provide a large second-level cache to the processor chip and to act as an interface to the system bus and memory for missed cache references. The processor module in the DEC 7000 system was designed to use either VAX or Alpha AXP chips. As noted above, a common design is used in the implementation of the VAX and DEC 7000 and 10000 systems, with the only significant differences being the processor chip and the console/diagnostic code. Figure 3 is a block diagram of the processor module.

The processor module provides a 4MB external cache, which is shared by the processor chip and the bus interface chips. The cache is organized as a single set (direct mapped), with a block and fill size of 64 bytes. The external cache conforms to a write-back, conditional update, cache coherency protocol. The processor on-chip data cache is a proper subset of the external cache and uses a write-through protocol.[4]

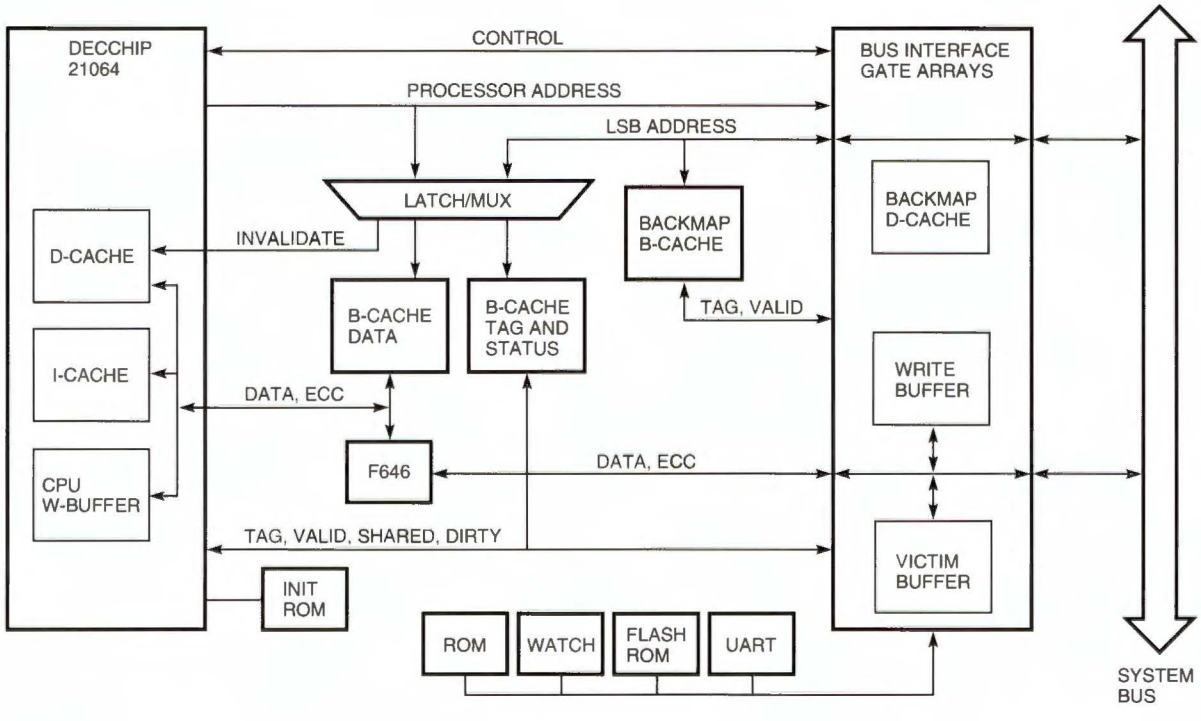



*Figure 3 Block Diagram of the DEC 7000 Processor Module*

The structure of the cache is shown in Figure 4. Each cache line consists of 512 bits of data (with 112 bits of ECC), 12 bits of tag (with 1 parity bit), and 3 status bits (with 1 parity bit). The 12 bits of tag data applied to a 4MB cache size sets a processor physical address capability of 16GB. (This is a processor limitation, and future processors will address larger memory sizes.) The control bits contain information that allows the cache and memory systems to maintain coherency. The control bits are defined as follows:

- A valid bit, indicating whether or not this line contains valid data
- A shared bit, indicating whether or not this line may also be resident in another processor's cache in the system
- A dirty bit, indicating whether or not this line has been written to by this processor

Upon detection of a cache read miss in the processor on-chip cache, the processor accesses the external cache tag to see if the given block is resident. The processor chip contains the tag comparitor and status logic to determine a "hit." If the block is resident in the external cache, the processor then cycles the external cache data store twice, each time reading in 128 bits of data and 28 bits of ECC for a total of 32 bytes (internal processor cache block size is 32 bytes). The external cache cycles at a rate five times the processor chip clock period (and at two times the period for the VAX variant). Upon the detection of a "miss," the processor chip informs the bus interface chips by means of handshake signals and waits until the miss is serviced on the LSB.

Upon a data write by the processor, the data is written through to the external cache. If the data is already resident in the cache, it is updated and conditionally broadcast onto the system bus if marked as shared. If the selected cache line contains a different valid tag, the current (old) cache line is written to memory and replaced by the new tag and data. To improve performance during this operation, the current cache line is stored in a local victim buffer while the new data is read. After the new data has been placed in the cache, the old data is written back to memory as a background operation.

A duplicate set of cache tags (backmaps) are kept by the bus interface logic for both the external cache and the internal processor chip D-cache. These backmaps are accessed by the bus interface logic on all bus references to determine the action necessary to maintain cache/memory coherency.

On bus read requests, the processor bus interface references its external cache backmap and supplies data from the on-board cache if a "dirty" copy of the data is present. On bus writes, a check is performed to see if the data is present in the processor on-chip D-cache. If the data line is present, the updated data is accepted. If the data line is not present but is instead in the external cache, the line is invalidated. This cache update policy is an attempt to minimize false sharing of data by only updating on references to a cache line in the processor on-chip cache, which is small and should contain only freshly referenced data.

False sharing of data is a problem common to multiprocessor systems running fully symmetric operating systems. When a process is migrated from one processor to another, dirty data often remains in the cache of the previous processor. When the new processor requests that data, it becomes "shared," resulting in the need to update all copies by means of bus transactions on all subsequent modifications of the data. Since the process has migrated, there is no need to maintain the state of the data in the cache of the previous processor; doing so slows down execution of the process due to the bus transactions required to update. The write-update policy described in the previous paragraph provides a means to estimate if "shared" data is still in use by the previous processor and provides a means to flush it from the previous cache if it has not been recently referenced.

The external cache is 128 bits wide with longword ECC protection. The ECC scheme used to

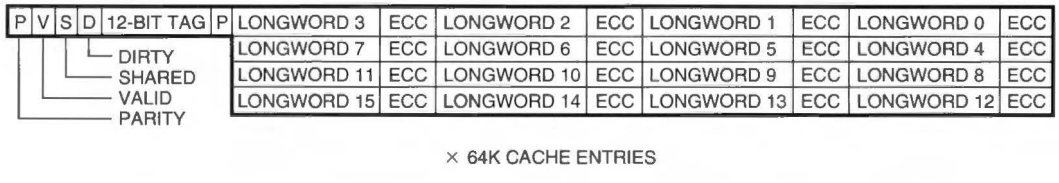



*Figure 4 External Cache Structure*

protect the external cache is identical to that used on the LSB, which allows flow-through ECC. The processor chip checks and corrects data for all processor refills. The bus interface chips perform lookaside ECC checking for fault isolation purposes but do not perform ECC correction.

The processor module also provides system console functions. The module includes universal asynchronous receiver/transmitters (UARTs) for communication with the console terminal and power subsystems, a time-of-year clock, and 896KB of flash read-only memories (ROMs) for console and diagnostic code. Each processor contains a complete console subsystem, but only one module uses this function in a multiprocessor system. This approach allows static reconfiguration of the system in the event of a module failure.

A 4MB module-level cache was chosen because it was the largest natural implementation using 256K × 4 SRAMs driven by the 128-bit-wide cache data path defined by the DECchip 21064 microprocessor. Denser SRAMs were not available at the necessary speed (10 to 12 ns), and a multiway cache architecture is not easily implemented with the DECchip 21064. The fill size of 64 bytes was selected to efficiently use the 16-byte-wide system bus and provide 80 percent bus data efficiency.

Figure 5 shows a photograph of side 1 of a processor module. Additional cache RAMs and drivers reside on side 2.

## Memory Module

The memory subsystem of the DEC 7000 comprises one to seven memory array modules with a single module capacity of 64 to 2048MB. The primary functions of the memory array modules are to respond to bus read/write functions, refresh the memory RAMs, and maintain ECC data for the memory. The design supports either 4MB or 16MB DRAMs, on-board interleaving on modules with greater than 64MB, and multimodule interleaving under many conditions.

The DEC 7000 memory modules run synchronous with the LSB. Memory transactions occur in fixed cycles relative to the system bus. All memory space transfers consist of 64-byte blocks that are transferred 16 bytes at a time over four contiguous data cycles. Read and write data wrapping is done on 32-byte naturally aligned boundaries. The DRAMs are 4-bit-wide parts, and an entire 64-byte block is read or written in parallel and buffered for bus transmission.

Data wrapping is a method used to provide a lower latency return of the data required by a read command. The bus contains an extra address bit that indicates in which half of a 64-byte block the requested data lies. The memory controller returns the half block containing the target data first, allowing faster resumption of processing. Data wrapping has no benefit on write transactions but is done to simplify the design of the system.

DEC 7000 memory modules are protected with a quadword ECC algorithm. The chosen ECC implementation allows detection and correction of single-bit failures, detection of all 2-bit failures, and detection and correction of any error wholly contained within a 4-bit-wide DRAM. Memory modules convert LSB longword (32-bit) ECC into quadword (64-bit) ECC that is stored with LSB data on writes. During LSB reads, quadword ECC is converted to longword ECC. Quadword ECC allows for higher packing densities on the memory module with fewer DRAM components. Longword ECC is used on the system bus because the DECchip 21064 microprocessor dictates the use of longword ECC in its external caches, and the timing of the external cache will not allow a conversion to a different ECC for bus transactions.

The memory module contains a hardware-based self-test that checks each bit on the module to be sure it can be set to either a 0 or a 1 state and initializes the memory to a known good ECC state. All memory modules execute self-test in parallel upon system initialization at a rate of approximately 35MB per second. This approach results in substantial savings in boot time as compared to a system that tests memory with initialization code executed by the processor. Moreover, the self-test provides excellent error isolation in the event of a failure.

DEC 7000 memory is designed in 64MB, 128MB, 256MB, 512MB, and 2GB modules. The 64MB, 128MB, and 256MB modules use 4MB DRAMs, double-side surface mounted. The 512MB modules use 4MB DRAMs mounted on soldered-in single in-line memory modules (SIMMs). (PC-style socketed SIMMs proved unreliable for large configurations.) The 2GB modules use 16MB DRAMs mounted on soldered-in SIMMs.

## I/O Subsystem

The DEC 7000 I/O subsystem consists of an I/O port controller and four high-speed parallel ports. The I/O controller provides an interface between the system bus and the parallel ports. Additional

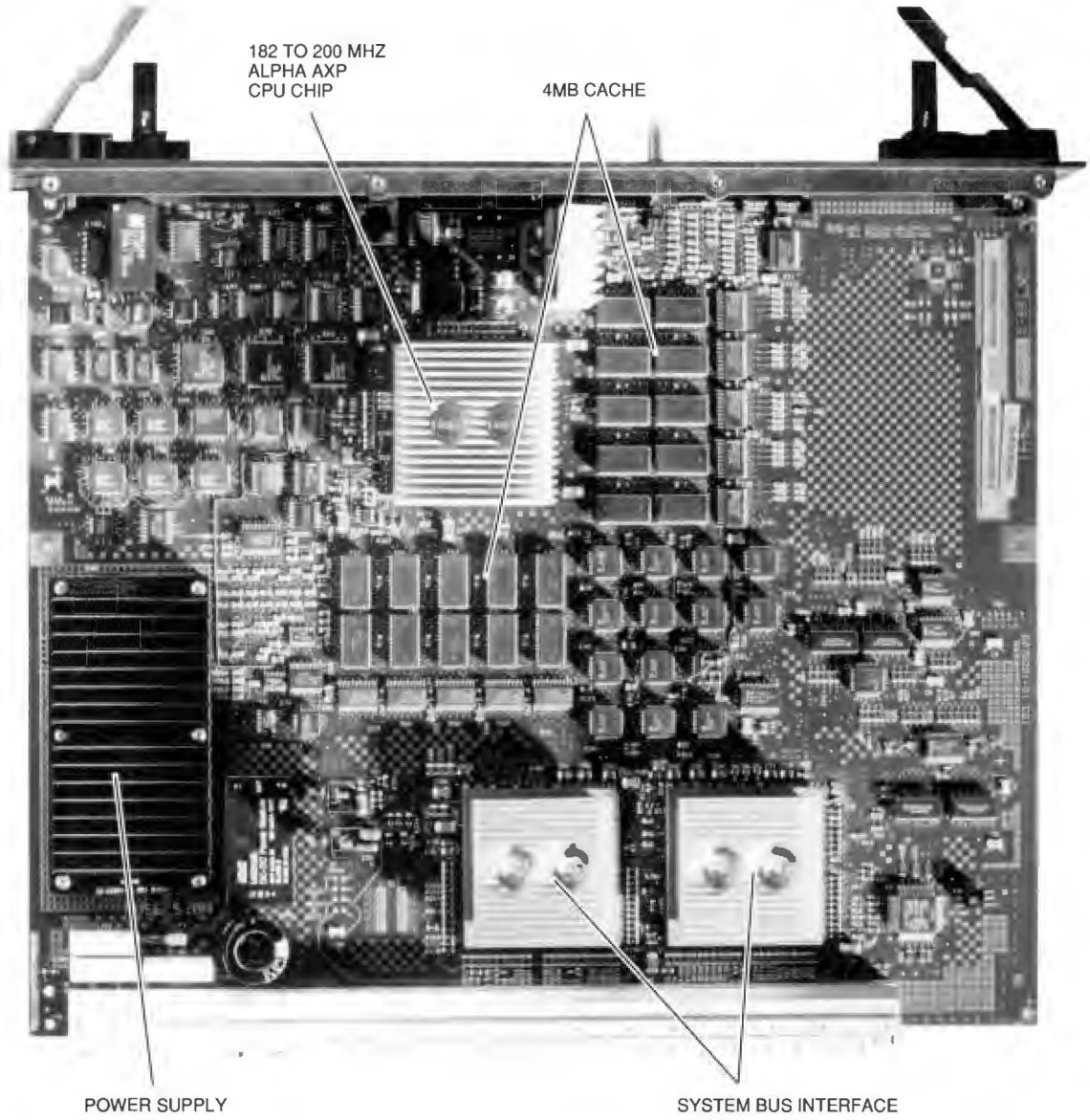



*Figure 5 Processor Module, Major Components Highlighted*

modules provide the interface between the high-speed parallel ports and specific standard I/O buses. To date, interfaces have been designed for the XMI, which is used as the I/O bus on the VAX 6000 and VAX 9000 systems, and for the Futurebus+, which is an IEEE standard high-performance bus definition.

The I/O port controller and specific bus adapter architecture was adopted to allow a flexible bus strategy that can evolve over time, as well as to accommodate the physical separation of processor and I/O subsystems necessary in an expandable system with multiple I/O channels. The I/O port controller cable(s) will function to a maximum cable length of 3 meters. This length allows I/O expansion cabinets to be placed on either side of the main system cabinet.

The aggregate bandwidth of the I/O port controller is 256MB per second. Each parallel port is capable of operating at a maximum of 135MB per second for data flowing from the I/O subsystem to memory and at 88MB per second for data flowing from memory to the I/O subsystem.

The I/O port controller module with its four parallel ports is a standard part of all DEC 7000

systems and resides in a dedicated system backplane slot. Various system configurations are available that contain between one and four XMI I/O buses. The Futurebus+ subsystems will be available when Futurebus+ components become available in the computer industry.

The I/O port controller provides a "mailbox" interface between the processor and I/O devices. A processor instruction cannot directly access a register in an I/O device, as was possible on previous VAX implementations. To use the "mailbox" interface, a processor creates a work descriptor packet in memory and then issues a command to the I/O port controller to execute the command. Command completion is asynchronous and the processor may choose to do other work while the command is executed. The "mailbox" interface between processors and I/O devices was created to allow relatively slow I/O devices to interface to a high-speed, nonpended system bus. If a processor were allowed to access the I/O device directly, the system bus would be stalled for large portions of time.

Clearly the mailbox communications method is more complicated than traditional direct access. Fortunately the mailbox is used only when a processor needs to directly access an I/O device. The I/O device can directly access main memory (or possibly a CPU cache) with all necessary buffering done by the I/O port controller. Most modern high-performance I/O adapters use high-level, packet-based protocols, which require very little direct access of the I/O adapter by the processor.

A typical CPU-initiated I/O transaction to an intelligent disk controller on an XMI bus to read from the disk would have the following steps.

- The CPU places a disk controller command packet requesting a disk read into system memory.
- The CPU sets up an I/O mailbox structure with a command to inform the disk controller that there is a command packet in memory, writes a register in the I/O port controller to inform it that there is a mailbox transaction to complete, and then spins on a done bit in the mailbox structure.
- The I/O port controller fetches the mailbox structure from memory, generates an XMI write command to the disk controller, and sets the done bit in the mailbox structure. The CPU sees the assertion of the done bit and goes on to other work.
- The disk controller receives the mailbox data and then generates an XMI request to read its command packet from memory.
- The I/O port controller reads the specified command packet from memory 64 bytes at a time and sends it back to the disk controller 32 bytes at a time.
- The disk controller decodes the command packet, reads the requested data from disk, and starts writing to system memory in 32-byte segments.
- The I/O port controller buffers the 32-byte writes from the disk controller into 64-byte segments and writes the data to system memory.
- The disk controller signals an interrupt on the XMI to indicate that the requested operation is complete, which is received by the I/O port controller. The I/O port controller signals an interrupt to the CPU.

## Console and Diagnostics

Like many previous VAX systems, the DEC 7000 system employs an embedded console. The console function is performed by code run on the processors within the system rather than by a dedicated, detached front-end processor.

Unlike the strategy for previous VAX systems, a unified console and diagnostic strategy was adopted for the DEC 7000 and 10000, VAX 7000 and 10000, and DEC 4000 systems. A single code base not only provides the basic console functions but also extends diagnostic support for manufacturing and field firmware upgrade support. This unified strategy has reduced the total development effort and promoted a common "look and feel" across the different systems.

The console development also differed from that of previous VAX systems. The primary implementation language was C, with only various architecture-specific code in Alpha AXP (or VAX) assembly language. The console and processor diagnostic code was simulated prior to the arrival of hardware. This simulation greatly simplified early hardware debug; the console had basic functionality after a single debug session.

At power-up, each processor acts independently to execute processor-specific diagnostics and console initialization. The processors then select a console primary, which then proceeds to test and configure the memory and I/O subsystems. The console primary also retains control of the console terminal line; console secondaries communicate

with the primary through memory-resident messages. After initialization, diagnostic or other console tasks can be assigned to any processor in the configuration. One benefit of this arrangement is that system diagnostics and exercisers can be run in parallel.

Like previous DECsystem consoles (that is, systems based on MIPS Co. chips), the DEC 7000 console provides a set of services, or callbacks, to the operating system. These services can be used to control automatic bootstrapping across operating system crashes as well as primitive I/O services used by the operating system during bootstrap and system crash. The latter simplifies the operating system device support by providing simple read/write functions common to all devices.

A feature of the power of the console is the field firmware update utility. Field upgrade of all system firmware (console and I/O adapters) is accomplished by the DEC 7000 firmware update utility (LFU). LFU is really a dedicated console image which is distributed on CDROM. The system console is used to boot LFU, which is then used to update all system firmware.

## *System Packaging*

The DEC 7000 system cabinet is 1700 mm high by 800 mm wide by 865 mm deep. The cabinet houses the system backplane, up to two I/O subsystems, and disk arrays or batteries for the system battery-backup function. Expansion is possible by using one or two I/O expander cabinets, each of which houses up to two additional I/O subsystems and additional disk arrays. Further mass storage expansion is possible with Digital's standard line of mass storage cabinets connected by CI, DSSI, or SI interconnects.

The DEC 7000 cabinetry has been designed for easy system upgrade and servicing. The system backplane assembly, power system, and I/O subsystems are modular and easily replaced by field personnel. The process of future upgrades can be accomplished more quickly and reliably through the use of modular subassemblies.

As shown in Figure 6, the DEC 7000 main system cabinet contains a central air mover with logic assemblies above and below it. The air mover is a single motor with a large molded vane assembly and can pull air through both the upper and the lower logic assemblies. An air flow of approximately 900 cubic feet per minute with velocities up to 1800 linear feet per minute is maintained through the upper logic assembly, which contains the processor and memory subsystems. Although not necessary for the DECchip 21064, this large volume of air movement was designed into the machine to allow upgrades through several generations of processor chips. By using standard air-cooling techniques and customized module "boxes" that optimize local air flow, it is possible to cool processor chips of up to 70 watts in the DEC 7000 system cabinet.

Above the air mover are the system backplane and the modular power subsystem. Below the air

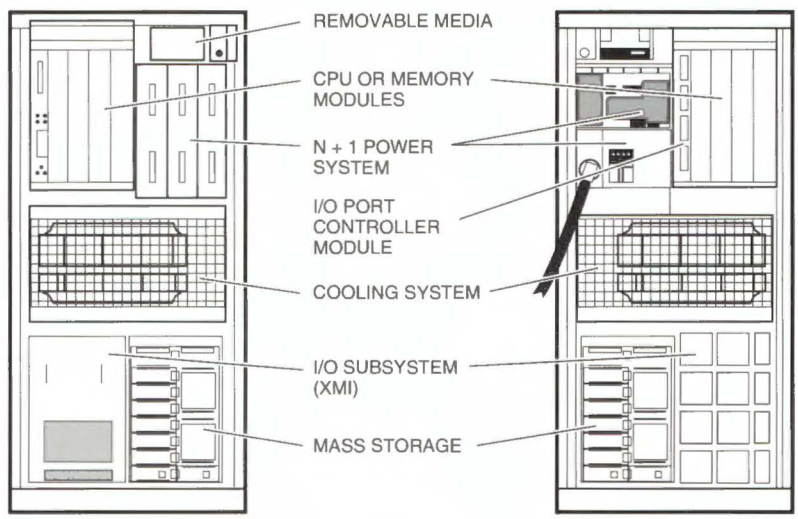



*Figure 6 DEC 7000 Main System Cabinet, Front (Left) and Rear (Right) Views*

mover are four modular spaces for I/O bus backplanes, disk drives, or batteries.

I/O, disk, and battery subsystems occupy varying amounts of the four modular spaces. The XMI subsystem occupies two spaces and is oriented front to back because of its rear-exit cabling scheme. The Futurebus+ subsystem occupies a single rear space. Disk subsystems consisting of up to six 5.25-inch (DSSI or SCSI [small computer system interface]) or fourteen 3.5-inch (SCSI only) drives may occupy any of the modular spaces. Batteries for the uninterruptible power system occupy two modular spaces, which may be oriented either front to back (for XMI-based systems) or side to side (for Futurebus+ systems).

The expander cabinet is identical to the main system cabinet, with two exceptions: disks may be packaged in the area occupied by the system backplane, and there is no control panel. Up to two XMI or Futurebus+ subsystems may be placed in an expander cabinet.

## Power Subsystem

The power subsystem of the DEC 7000 family has a highly modular, hierarchical design. The basic power system provides 48-volt direct current (VDC) to all subassemblies which in turn further regulate to necessary voltages. Each module in the system backplane contains on-board regulation. This feature will allow the system to easily evolve with changing voltage requirements as CMOS technology moves to lower voltages to reduce power consumption. Voltage tolerances can be tightly controlled since transmission drops are negated; a precise voltage level can be set at the time of module manufacture. The voltage and tolerance to a high-performance CMOS processor must be very tightly controlled in order to extract maximum performance. The XMI, Futurebus+, and disk subsystems all regulate the 48 VDC to lower voltages at a subsystem-wide level, not at the module level.

The 48-VDC modular power system consists of one to three parallel regulators, each of which produces 2400 watts of power. A maximally configured cabinet needs no more than two power regulators. An additional regulator can be configured into the system to provide an $n$+1 capability for higher availability.

The power system also includes a battery standby function that provides 48 VDC throughout the system in the event of an AC power failure. Unlike earlier VAX systems in which power was maintained only to system memory, the DEC 7000 keeps the entire system powered, including in-cabinet mass storage. Depending on the system configuration, power is maintained for a minimum of 20 minutes in an $n$+1 power configuration. $N$+1 power with full battery backup is standard on all DEC 10000 systems.

The DEC 7000 system employs a highly intelligent power subsystem with microprocessors in all 48-volt regulators, which report status to processor modules by means of a serial interconnect. System software can therefore monitor a wide range of power system operating parameters, including voltage output, AC input, efficiency, and battery charge state. In a large configuration with optional expander cabinets, the expander cabinet power systems also communicate with the system processors to provide system-wide power status.

## Performance

The DEC 7000 and DEC 10000 systems are the fastest uniprocessor and multiprocessor, microprocessor-based computer systems in the world as of their introduction date (10 November 1992) and as defined by SPEC89 and SPEC92 benchmark data. For compute-intensive benchmarks, the DEC 10000 is approximately 10 percent faster than the DEC 7000, based entirely on the difference in processor clock speed.

The base performance of the DEC 7000 and DEC 10000 systems is determined by the speed of the processor chip and is heavily influenced by cache, memory, and I/O subsystems. The design goal for the DEC 7000 and DEC 10000 systems was to extract the maximum possible performance from the DECchip 21064 by providing an electrical and physical environment capable of supporting 200-MHz processor operation as well as large caches, a large and fast memory subsystem, and multiple I/O subsystems.

While full system-level performance data is still being collected, the very high speed processor performance measured on the SPEC benchmarks combined with the very high performance cache, memory, and I/O subsystems of the DEC 7000 and DEC 10000 systems should yield very impressive overall system performance. See Table 1.

## Design Process

The DEC 7000 system was specified, designed, and tested by a group of approximately 200 people in Boxboro, Massachusetts. The system design team

**Table 1 DEC 7000 and DEC 10000 System Performance Measurements**

| | DEC 7000 | DEC 10000 |
|---|---|---|
| SPECmark89 | 167.4 | 184.1 |
| SPECint89 | 95.1 | 104.5 |
| SPECfp89 | 244.2 | 268.6 |
| SPECint92 | 96.9 | 106.5 |
| SPECfp92 | 182.1 | 200.4 |
| SPECthroughput89 (4 CPUs) | 604.4 | 654.6 |
| LINPACK double-precision | | |
| 100×100 (MFLOPS) | 38.6 | 42.5 |
| 1000×1000 (MFLOPS) | 102.1 | 111.6 |

was responsible for all aspects of the design except the DECchip 21064 microprocessor.

Conceptual work on a system to follow the VAX 6000 family was started in early 1989, although at that time design work was focused on implementations using VAX and MIPS R4000 processors. In the latter part of 1989, the decision was made to pursue the Alpha AXP strategy, and earlier concepts were reworked to incorporate much higher levels of performance to accommodate the proposed Alpha AXP chip.

In October–December 1989, a core team of approximately 10 engineers was assembled to firmly define system architecture and to produce specifications for all subassemblies. By July 1990 all specifications were complete, and implementation was started. The first processor module was powered up in June 1991, followed by a full system power-up in September 1991. The VMS operating system was booted on a DEC 7000 system on September 9, 1991, and OSF was booted in November 1991.

A minimal DEC 7000 system includes 430,000 gates of logic contained in gate arrays, whereas a minimal VAX 6000 Model 200 includes 94,000 gates. Despite more than four times the gate count, the design portion of the DEC 7000 program was completed in approximately 9 months as compared to 12 months for the VAX 6000 program. This reduction in design time was achievable in part because of the maturing of the engineering population (many of the DEC 7000 engineers had worked on various VAX 6000 implementations), as well as advances in design tool technology and the availability of significantly more powerful computers for design simulation. At its peak, the DEC 7000 program was consuming 1500 VAX units of performance, or VUPs, of compute power (primarily multiprocessor VAX 6000 Model 500 systems) and used over 325,000 hours of CPU time for simulations.

## Conclusion

The DEC 7000 and DEC 10000 systems are the second generation of highly configurable and expandable systems produced by Digital Equipment Corporation. These are the first systems expressly designed to accommodate multiple-processor architecture types. As computer technology moves forward at an ever-increasing pace, this type of design will be demanded by computer users and will be necessary to manage engineering costs.

The DEC 7000 and DEC 10000 system platform will accommodate new VAX and Alpha AXP processors for several years. Over that time, this platform will span a performance range of greater than 50:1. It will provide computer users with a stable system environment that should help minimize the changes caused by the continued development of new processor chips. While this level of flexibility incurs additional initial engineering and product costs, it does provide a very cost-effective way to deal with the inexorable forward march of technology.

## Acknowledgments

The following engineers formed the system architecture team of the project that produced the DEC 7000 and DEC 10000 and VAX 7000 and VAX 10000 products: Frank Bomba, Reinhard Schumann, Mike Callander, Steve Polzin, Kathy Harrington, Dave Mayo, Catharine van Ingen, Vicky Triolo, Bob Dickson, Dave O'Keefe, Jim Leahy, Hansel Collins, Jim Stegeman, Darrel Donaldson, Dave Hartwell, Charlie Barker, Mark Stefanski, Brian Allison. Various parts of this text originated within engineering specifications written by this team.

## References

1. *Digital Technical Journal,* vol. 2, no. 2, featuring papers on the VAX 6000 Model 400 (Spring 1990).
2. G. Uhler et al., "The NVAX and NVAX+ High-performance VAX Microprocessors," *Digital Technical Journal,* vol. 4, no. 3 (Summer 1992): 11–23.
3. D. Dobberpuhl et al., "A 200-MHz 64-bit Dual-issue CMOS Microprocessor," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 35–50.
4. A.J. Smith, "Cache Memories," *Computing Surveys,* vol. 14, no. 3 (September 1982).

*Nancy P. Kronenberg*
*Thomas R. Benson*
*Wayne M. Cardoza*
*Ravindran Jagannathan*
*Benjamin J. Thomas III*

# *Porting OpenVMS from VAX to Alpha AXP*

*The OpenVMS operating system, developed by Digital for the VAX family of computers, was recently moved from the VAX to the Alpha AXP architecture. The Alpha AXP architecture is a new RISC architecture introduced by Digital in 1992. This paper describes solutions to several problems in porting the operating system, in addition to performance benefits measured on one of the systems that implements this new architecture.*

The VAX architecture is an example of complex instruction set computing (CISC), whereas the Alpha AXP architecture is based on reduced instruction set computing (RISC). The two architectures are very different.[1] CISC architectures have performance disadvantages as compared to RISC architectures.[2] Digital ported the OpenVMS system to the Alpha AXP architecture mainly to deliver the performance advantages of RISC to OpenVMS applications. This paper focuses on how Digital's OpenVMS AXP operating system group dealt with the large volume of VAX assembly language and with system kernel modifications that had VAX architecture dependencies.

The OpenVMS AXP group had two important requirements in addition to the primary goal of increasing performance: first, to make it easy to move existing users and applications from OpenVMS VAX to OpenVMS AXP systems; second, to deliver a high-quality first version of the product as early as possible. These requirements led us to adopt a fairly straightforward porting strategy with minimal redesigns or rewrites. We view the first version of the OpenVMS AXP product as a beginning, with other evolutionary steps to follow.

The Alpha AXP architecture was designed for high performance but also with software migration from the VAX to the Alpha AXP architecture in mind. Included in the Alpha AXP architecture are some VAX features that ease the migration without compromising hardware performance. VAX features in the Alpha AXP architecture that are important to OpenVMS system software are: four protection modes, per-page protection, and 32 interrupt priority levels. The Alpha AXP architecture also defines a privileged architecture library (PAL) environment, which runs with interrupts disabled and in the most privileged of the four modes (kernel). PALcode is a set of Alpha AXP instructions that executes in the PAL environment, implementing such basic system software functions as translation buffer (TB) miss service. On OpenVMS AXP systems, PALcode also implements some OpenVMS VAX features such as software interrupts and asynchronous traps (ASTs). The combination of hardware architecture assists and OpenVMS AXP PALcode made it easier to port the operating system to the Alpha AXP architecture.

The VAX architecture is 32-bit; it has 32 bits of virtual address space, 16 32-bit registers, and a comprehensive set of byte, word (16-bit), and longword (32-bit) instructions. The Alpha AXP architecture is 64-bit, with 64 bits of virtual address space, 64-bit registers (32 integer and 32 floating-point), and instructions that load, store, and operate on 64-bit quantities. There are also longword load, store, and operate instructions, and a canonical sign-extended form for a longword in a 64-bit register.

The OpenVMS AXP system has anticipated evolution from 32-bit address space size to 64-bit address space by changing to a page table format that supports large address space. However, the OpenVMS software assumes that an address is the same size as a longword integer. The same assumption can exist in applications. Therefore, the first version of the OpenVMS AXP system supports 32-bit address space only.

Most of the OpenVMS kernel is in VAX assembly language (VAX MACRO-32). Instead of rewriting the VAX MACRO-32 code in another language, we developed a compiler. In addition, we required inspection and manual modification of the VAX MACRO-32 code to deal with certain VAX architectural dependencies. Parts of the kernel that depended heavily on the VAX architecture were rewritten, but this was a small percentage of the total volume of VAX MACRO-32 source code.

## *Compiling VAX MACRO-32 Code for the Alpha AXP Architecture*

Simply stated, the VAX MACRO-32 compiler treats VAX MACRO-32 as a source language to be compiled and creates native OpenVMS AXP object files just as a FORTRAN compiler might. This task is far more complex than a simple instruction-by-instruction translation because of fundamental differences in the architectures, and because source code frequently contains assumptions about the VAX architecture and the OpenVMS Calling Standard on VAX systems.[3,4] The compiler must either transparently convert these VAX dependencies to their OpenVMS AXP counterparts or inform the user that the source code has to be changed.

### *Source Code Annotation*

We extended the VAX MACRO-32 source language to include entry-point declarations and other directives for the compiler's use, which provide more information about the intended behavior of the program. Entry-point declarations were introduced to allow declaration of: (1) the register semantics for a routine when the defaults are not appropriate and (2) the specialized semantics of frameless subroutines and exception routines to be declared.

The differing register size between the VAX and the Alpha AXP architectures influenced the design of the compiler. On the VAX, MACRO-32 operates on 32-bit registers, and in general, the compiled code maintains 32-bit sign-extended values in Alpha AXP 64-bit registers. However, this code is now part of a system that uses true 64-bit values. As a result, we designed the compiler to generate 64-bit register saves of any registers modified in a routine, as part of the "routine prologue code." Automatic register preservation has proven to be the safest default but must be overridden for routines that return values in registers, using appropriate entry-point declarations.

Other directives allow the user to provide additional information about register state and code flow to improve generated code. Another class of directives instructs the compiler to preserve the VAX behavior with respect to granularity of memory writes or atomicity of memory updates. The Alpha AXP architecture makes atomic updates and guaranteed write granularity sufficiently costly to performance that they should be enabled only when required. These concepts are discussed in the section Major Architectural Differences in the OpenVMS Kernel.

### *Identifying VAX Architecture and Calling Standard Dependencies*

As mentioned earlier, the compiler must either transparently support VAX architecture-dependent constructs or inform the user that a source change is necessary. A good example of the latter case is reliance on VAX JSB/RSB (jump to subroutine and return) instruction return address semantics. On VAX systems, a JSB instruction leaves the return address on top of the stack, which is used by the RSB instruction to return.[3] System subroutines often take advantage of this semantic in order to change the return address. This level of stack control is not available in a compiled language. In porting the OpenVMS system to the Alpha AXP architecture, we developed alternative coding practices for this and many other nontransportable idioms.

The compiler must also account for the differences in the OpenVMS Calling Standard on the VAX and Alpha AXP architectures. Although transparent to high-level language programmers, these differences are very significant in assembly language programming.[4] To operate correctly in a mixed language environment, the code generated by the VAX MACRO-32 compiler must conform to the OpenVMS Calling Standard on the Alpha AXP architecture.

On VAX systems, a routine refers to its arguments by means of an argument pointer (AP) register, which points to an argument list that was built in memory by the routine's caller. On Alpha AXP systems, up to six routine arguments are passed to the called routine in registers; any additional arguments are passed in stack locations. Normally, the VAX MACRO-32 compiler transparently converts AP-based references to their correct Alpha AXP locations and converts the code that builds the list to

initialize the arguments correctly. In some cases, the compiler cannot convert all references to their new locations, so an emulated VAX argument list must be constructed from the arguments received in the registers. This so-called "homing" of the argument list is required if the routine contains indexed references into the argument list or stores or passes the address of an argument list element to another routine.

The compiler identifies these coding practices during its process of flow analysis, which is similar to the analysis done by a standard high-level language optimizing compiler. The compiler builds a flow graph for each routine and tracks stack depth, register use, condition code use, and loop depth through all paths in the routine flow. This same information is required for generating correct and efficient code.

### *Access to Alpha AXP Instructions and Registers*

In addition to providing migration of existing VAX code, the VAX MACRO-32 compiler includes support for a subset of Alpha AXP instructions and PALcode calls and access to the 16 integer registers beyond those that map to the VAX register set. The instructions supported either have no direct counterpart in the VAX architecture or are required for efficient operation on a full 64-bit register value. These "built-ins" were required because the OpenVMS AXP system uses full 64-bit values for some operations, such as manipulation of 64-bit page table entries (PTEs).

### *Optimization*

The compiler includes certain optimizations that are particularly important for the Alpha AXP architecture. For example, on a VAX system, a reference to an external symbol would not be considered expensive. On an Alpha AXP system, however, such a reference requires a load from the linkage section to obtain the address of the symbol prior to loading the symbol's value. (The linkage section is a data region used for resolving external references.[4]) Multiple loads of this address from the linkage section may be reduced to a single load, or the load may be moved out of a loop to reduce memory references.

Other optimizations include the elimination of memory reads on multiple safe references, register state tracking for optimal register-based memory references, redundant register save/restore removal, and many local code generation optimizations for particular operand types. Peephole optimization of local code sequences and low-level instruction scheduling are performed by the back end of the compiler.

In some instances, the programmer has knowledge of register state or other code behavior that cannot be inferred from the source code alone. Compiler directives are provided for specifying register alignment state, structure base address alignment, and likely flow paths at branch points.

Certain types of optimization typically performed by a high-level language compiler cannot be performed by the VAX MACRO-32 compiler, because assumptions made by the MACRO-32 programmer cannot be broken. For example, the order of memory reads may not be changed.

## *Major Architectural Differences in the OpenVMS Kernel*

This section concentrates on architectural changes that affect synchronization, memory management, and I/O. These are not the only architectural differences that cause significant changes in the kernel. However, they are the major differences and are representative of the effort involved in porting the OpenVMS system to the Alpha AXP architecture.

For the most part, it was possible to isolate architecture-dependent changes to a few major subsystems. However, differences in the memory reference architecture had a pervasive impact on users of shared data and many common synchronization techniques. Other differences such as those related to memory management, context switching, or access to I/O devices were confined mostly to the relevant subsystems.

### *Effects of Architectural Differences in Memory Subsystems*

The following differences between the VAX and Alpha AXP memory reference architectures affected synchronization:[1,3]

- Load/store architecture rather than atomic modify instructions
- Longword and quadword writes with no byte write instructions
- Read/write ordering not guaranteed

Load/store memory reference instructions are characteristic of most RISC designs. However, the other differences are less typical. The reasons for all

three differences were hardware simplification and opportunities for increased hardware performance.[1] These differences result in significant changes in system software and in many opportunities for subtle errors, which can be detected only under heavy load. Adapting to these architectural changes without greatly impacting performance was one of the major challenges that faced the group in porting the OpenVMS system to the Alpha AXP architecture.

A load/store architecture such as Alpha AXP cannot provide the atomic read-modify-write instructions present in the VAX architecture. Thus, instruction sequences are necessary for many operations performed by a single, atomic VAX instruction, such as incrementing a memory location. The consequence is a need for increased awareness of synchronization. Instead of depending on hardware to prevent interference between multiple threads of execution on a single processor, explicit software synchronization may be required. Without this synchronization, the interleaving of independent load-modify-store sequences to a single memory location may result in overwritten stores and other incorrect results.

The lack of byte writes imposes additional synchronization burdens on software. Unlike VAX and most RISC systems, an Alpha AXP system has instructions to write only longwords and 64-bit quadwords, not bytes or words. Thus to write bytes, the software must include a sequence of instructions that reads the encompassing longword, merges in the byte, and writes the longword to memory. As a consequence, software must be concerned not only with shared access to the same memory cell by multiple threads, but also with access to independent but adjacent variables. Synchronization awareness is now extended from shared data to data items that are merely near each other.

The OpenVMS AXP operating system group avoided the above-mentioned problems introduced by the architectural changes in one of three ways:

- Explicit software synchronization was added between threads.
- Data items were relocated to aligned longwords or quadwords.
- Alpha AXP load locked and store conditional instructions were used.

The obvious solution of adding explicit synchronization in the form of a software lock is not always appropriate for several reasons. First, the problem may be independent data items that happen to share a longword. Synchronizing this sort of access in unrelated code paths is prone to error. Explicit synchronization may also have an unacceptable performance impact. Finally, deadlocks are a possibility if one thread interrupts another.

Ensuring that data items are in aligned longwords or quadwords both improves performance and eliminates interactions between unrelated data. This technique is used wherever possible but cannot be used in two major cases. One case occurs when the replication factor is too large. Expanding an array of thousands of bytes to longwords may simply not be acceptable. The other major problem case is data structures that cannot be changed because of external constraints. For example, data may represent a protocol message or a structure primarily resident on disk. Separate internal and external forms of such data structures could exist, but the performance cost of continuous conversions may not be acceptable.

Often the easiest and highest-performance way to solve synchronization problems is to use sequences of load locked and store conditional instructions. The load locked instruction loads an aligned longword or quadword while setting a hardware flag that indicates the physical address that was loaded. The hardware flag is cleared if any other thread, processor, or I/O device writes to the locked memory location. The store conditional instruction stores an aligned longword or quadword if and only if the hardware lock flag is still set. Otherwise, the store returns an error indication without modifying the storage location. These instructions allow the construction of atomic read-modify-write sequences to update any datum that is contained within a single aligned quadword. These sequences of instructions are significantly slower than normal loads and stores due to the necessity of waiting for the write to reach a point in the memory hierarchy where consistency can be guaranteed. In addition, their use may inhibit many compiler optimizations because of restrictions on the instructions between the load and store. Although faster than most other synchronization methods, this mechanism should be used sparingly.

The lack of guaranteed read/write ordering between multiple processors is another complication for the programmer trying to achieve proper synchronization. Although not visible on a single processor, this lack of ordering means that one

processor will not necessarily observe memory operations in the order in which they were issued by another processor. Thus, many obvious synchronization protocols will not work when writes to the synchronization variable and to the data being protected are observed to occur out of order. A memory barrier instruction is provided in the architecture to ensure ordering. However, the negative impact of this instruction on system performance requires that it be used only when necessary.

As described in the previous section, we used various mechanisms to solve the synchronization problems. Having multiple solutions allowed us to choose the one with the least performance impact for each case. In some cases, explicit synchronization was already in place due to multiprocessor synchronization requirements. In other cases, we expanded data structures at a cost of modest amounts of memory to avoid expensive synchronization when referencing data.

### *Privileged Architecture Changes*

Unlike the pervasive architectural changes described in the previous section, the privileged architecture differences had a more limited impact. The primary remaining areas of change are the new page table formats and the details of process context switching. This section describes memory management as a representative example. However, many limited changes still amount to modifying virtually every source module in the OpenVMS kernel, even if only to add compiler directives.

*Memory Management* Not surprisingly, the memory management subsystem required the most change when moving from the VAX to the Alpha AXP architecture. Aside from the obvious 64-bit addressing capability, two significant differences exist between the page table structures on the VAX and the Alpha AXP architectures. First, Alpha AXP does not have an architectural division between shared and process private address space. Second, the Alpha AXP three-level page table structure shown in Figure 1 allows the sharing of arbitrary subtrees of the page table structure and the efficient creation of large, sparse address spaces. In addition, future Alpha AXP processors may have larger page sizes.

To meet our schedule goals, we decided initially to emulate the VAX architecture's 32-bit address space as closely as possible. This decision required creating a 2-gigabyte (GB) process private address region (i.e., VAX P0 and P1) and a 2GB shared address region (i.e., VAX S0 and S1) for each process. This is easily accomplished by giving each process a private level 1 page table (L1PT) that contains two entries for level 2 page tables (L2PTs). One of these L2PTs is shared and implements the shared system region, whereas the other is private and implements the process private address regions. Although the smallest allowed page size of 8 kilobytes (KB) results in an 8GB region for each level 2 page table, we use only 2GB of each region to keep within our 4GB 32-bit limit. As shown in Figure 1, the L2PTs are chosen to place the base address of the shared system region at FFFFFFFF80000000 (hexadecimal), the same as the sign-extended address of the top half of the VAX architecture's 32-bit address space.

Although changes were extensive in the memory management subsystem, many were not conceptually difficult. Once we dealt with the new page table structure, most changes were merely for alternative page sizes, new page table entry formats, and changes to associated data structures. We did decide to keep the OpenVMS VAX concept of mapping process page tables as a single array in shared system space for our initial implementation. Although not viable in the long term due to the potentially huge size of sparse process page tables, this decision minimized changes to code that references process page tables. Having process page tables visible in shared system space turned out to be a significant complication in paging and in address space creation, but the schedule benefits derived from avoiding changes to other subsystems were considered worthwhile. We expect to change to a more general mechanism of self-mapping process page tables in process space for a subsequent OpenVMS AXP release.

Retaining the VAX appearance of process page tables allowed us to meet our goals of minimum change outside of the memory management subsystem. Unprivileged code is unaffected by the memory management changes unless it is sensitive to the new page size. Even privileged code is generally unaffected unless it has knowledge of the length or format of PTEs.

*Optimized Translation Buffer Use* Thus far, we may have given the impression that architectural changes always create problems for software. This was not universally true; some changes offered us

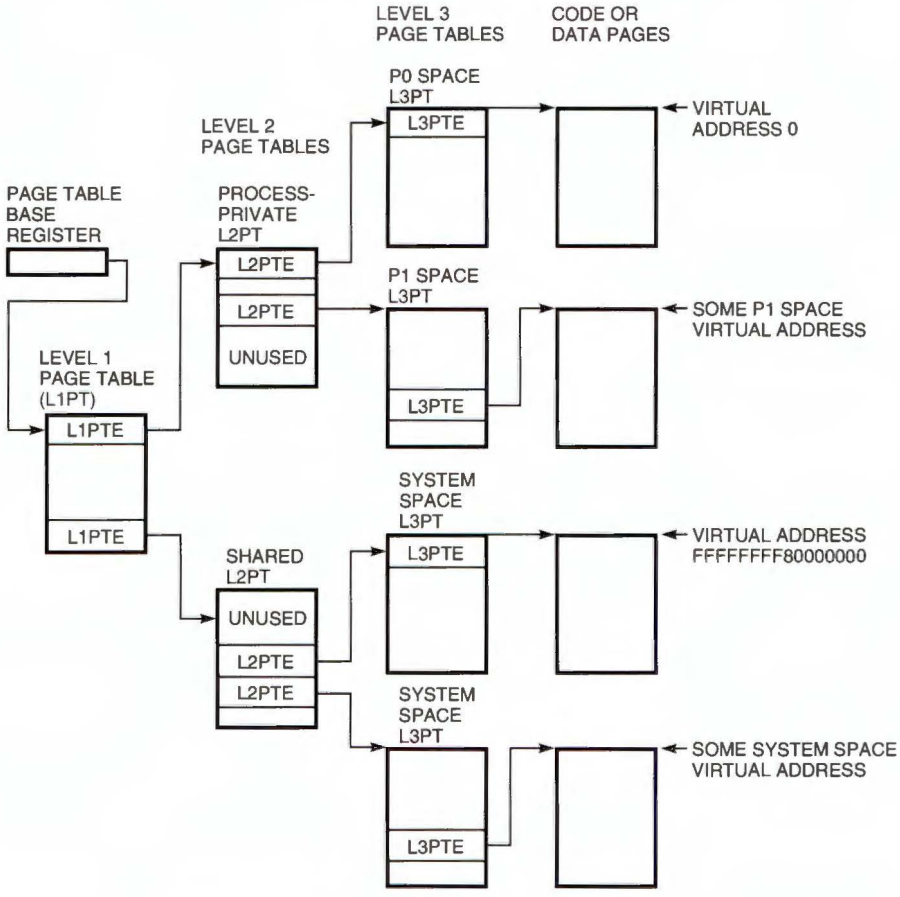



*Figure 1 OpenVMS AXP Page Table Structure*

opportunities for significant gains. One such change was an Alpha AXP translation buffer feature called granularity hints. TBs are key to performance on any virtual memory system. Without them, it would be necessary to reference main memory page tables to translate every virtual address to a physical address. However, there never seems to be enough TB entries. The Alpha AXP architecture allows a single TB entry to optionally map a virtually and physically contiguous block of properly aligned pages, all with identical protection attributes. These pages are marked for the hardware by a flag in the PTE.

Given granularity hints, near-zero TB miss rates for the kernel became attainable. To this end, we enhanced the kernel loading mechanisms to create and use granularity hint regions.

The OpenVMS AXP kernel is made up of many separate images, each of which contains several regions of memory with varying protections. For example, there is read-only code, read-only data, and read-write data. Normally, a kernel image is loaded virtually contiguously and relocated so that it can execute at any address. To take advantage of granularity hints, kernel code and data are loaded in pieces and relocated to execute from discontiguous regions of memory. Only a very few TB entries are actually used to map the entire nonpaged kernel, and the goal of near-zero TB misses was reached.

The benefits of granularity hints became immediately obvious, and the mechanism has since been expanded. Now, the OpenVMS AXP system also uses the code region for user images and libraries. This feature extends the benefits not only to images supplied by the OpenVMS system, but to customer applications and layered products as well. Of course, usage of this feature is only reasonable for images and libraries used so frequently that the permanent allocation of physical memory is

warranted. However, most applications are likely to have such images, and the performance advantage can be impressive.

## I/O

Unlike the architectural changes, the new I/O architecture structures an area that previously was rather uncontrolled. The project goal was to allow more flexibility in defining hardware I/O systems while presenting software with a consistent interface. These seem like contradictory aims, but both must be met to build a range of competitive, high-performance hardware that can be supported with a reasonable software effort.

The Alpha AXP architecture presents a number of differences and challenges that impacted the OpenVMS AXP I/O system. These changes spanned areas from longword granularity to device control and status register (CSR) access to how adapters may be built.

*CSR Access* A fundamental element of I/O is the access of CSRs. On VAX systems, CSR access is accomplished as simply another memory reference that is subject to a few restrictions. Alpha AXP systems present a variety of CSR access models.

Early in the project, the concept of I/O mailboxes was developed as an alternative CSR access model. The I/O mailbox is basically an aligned piece of memory that describes the intended CSR access. Instead of referencing CSRs by means of instructions, an I/O mailbox is constructed and used as a command packet to an I/O processor. The mailbox solves three problems: the mailbox allows access to an I/O address space larger than the address space of the system; byte and word references may be specified; and the system bus is simplified by not having to accommodate CSR references that may stall the bus. As systems get faster, these bus stalls are increasingly larger impediments to performance.

Mailboxes are the I/O access mechanism on some, but not all, systems. To preserve investment in driver software, the OpenVMS AXP operating system implemented a number of routines that allow all drivers to be coded as if CSRs were accessed by a mailbox. Systems that do not support mailbox I/O have routines that emulate the access. These routines provide insulation from hardware implementation details at the cost of a slight performance impact. Drivers may be written once and used on a number of differing systems.

*Read/Write Ordering* An I/O device is simply another processor, and the sharing of data is a multiprocessing issue. Since the Alpha AXP architecture does not provide strict read/write ordering, a number of rules must be followed to prevent incorrect behavior. One of the easiest changes is to use the memory barrier instructions to force ordering. Driver code was modified to insert memory barriers where appropriate.

The devices and adapters must also follow these rules and enforce proper ordering in their interactions with the host. An example is the requirement that an interrupt also act like a memory barrier in providing ordering. In addition, the device must ensure proper ordering for access to shared data and direct memory access.

*Kernel Processes* Another important way in which the Alpha AXP architecture differs from the VAX architecture is the lack of an interrupt stack. On VAX systems, the interrupt stack is a separate stack for system context. With the new Alpha AXP design, any system code must use the kernel stack of the current process. Therefore, a process kernel stack must be large enough for the process and for any nested system activity. This burden is unreasonable. A second problem is that the VAX I/O subsystem depends on absolute stack control to implement threads. As a result, most of the I/O code is in MACRO-32, which is a compiled language on the OpenVMS AXP system that does not provide absolute stack control.

These facts resulted in the creation of a kernel threading package for system code at elevated interrupt priority levels. This package, called kernel processes, provides a set of routines that support a private stack for any given thread of execution. The routines include support for starting, terminating, suspending, and resuming a thread of execution.

The private stack is managed and preserved across the suspension with no special measures on the part of the execution thread. Removing requirements for absolute stack control will facilitate the introduction of high-level languages into the I/O system.

## Performance

As stated earlier, the main purpose of the project was to deliver the performance advantages of RISC to OpenVMS applications. We adopted several methods, including simulation, trace analysis, and a variety of measurements, to track and improve

operating system and application performance. This section presents data on the performance of OpenVMS services and on the SPEC Release 1 benchmark suite.[5] Note that all Alpha AXP results are preliminary.

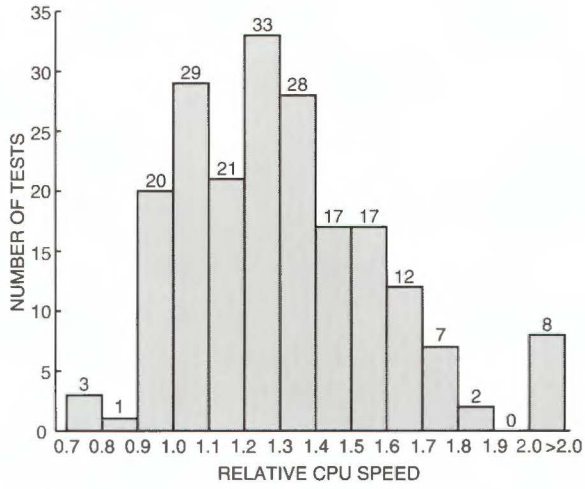
## *Performance of OpenVMS Services*

To evaluate the performance of the OpenVMS system, we used a set of tests that measure the CPU processing time of a range of OpenVMS services. These tests are neither exhaustive nor representative of any particular workload. We use relative CPU speed (i.e., VAX CPU time divided by Alpha AXP CPU time) as a metric to truly compare CPU performance. For I/O-related services, a RAM disk was used to eliminate I/O latencies.

The tests were run on a VAX system and an Alpha AXP system that are the same except for the CPU. Table 1 shows the configuration details of the two systems. Figure 2 shows the distribution of the relative CPU speed for the OpenVMS services measured. Most tests ran between 0.9 and 1.7 times faster on the Alpha AXP system than on the VAX system. Table 2 contains the results for a representative subset of the measured OpenVMS services.

## *Application Performance*

Applications vary in their use of operating system services. Most applications spend the majority of their time performing application-specific work and a small fraction of their time using operating system services. For these applications, performance depends mainly on the performance of hardware, compilers, and run-time libraries. We




Notes:
1. The relative CPU speed equals the CPU time on a VAX system divided by the CPU time on an Alpha AXP system.
2. A relative CPU speed greater than 1.0 implies that the Alpha AXP system is faster.
3. The total number of tests is 198.

*Figure 2 Distribution of Relative CPU Speed for OpenVMS Services*

**Table 1 Configuration Details for OpenVMS Services Test Environment**

| | VAX System | Alpha AXP System |
|---|---|---|
| Model number | VAX 7000 Model 610 | DEC 7000 Model 610 |
| Clock rate | 91 MHz | 182 MHz |
| Memory size | 64MB | 64MB |
| On-chip cache size | 1KB virtual I-cache<br>8KB physical I- and D-caches | 8KB physical I-cache<br>8KB physical D-cache |
| Backup cache size | 4MB I- and D-caches | 4MB I- and D-caches |
| Translation buffer | 96 entries | 12 ITB entries<br>32 DTB entries |
| Page size | 512 bytes | 8KB |
| Number of registers | 16 32-bit GPRs | 32 64-bit integer<br>32 64-bit floating-point |
| OpenVMS version | Pre-release V5.5-2 | Pre-release V1.0 |

Key:

| | |
|---|---|
| I | Instruction |
| D | Data |
| ITB | Instruction translation buffer |
| DTB | Data translation buffer |
| GPR | General-purpose register |

**Table 2 Relative CPU Speed for a Subset of OpenVMS System Services and Primitives**

| OpenVMS System Service or Primitive | Relative CPU Speed |
|---|---|
| Memory Management Services | |
| Create virtual address space | 1.03 |
| Delete virtual address space | 1.44 |
| Expand address region | 1.58 |
| Page fault without I/O (soft page fault) | 1.05 |
| Logical Name Services | |
| Translate a logical name | 1.75 |
| Event Flag Services | |
| Set an event flag | 1.45 |
| Clear an event flag | 1.35 |
| Process Control Services | |
| Create a process and activate an image | 1.17 |
| File System Services (File on a RAM Disk) | |
| File open | 1.34 |
| File close | 1.21 |
| File create | 1.24 |
| File delete | 1.31 |
| Record Management System (RMS) Services (File on a RAM Disk) | |
| Get record from a sequential file | 0.98 |
| Put record into a sequential file | 0.96 |

Note that the relative CPU speed equals the CPU time on a VAX system divided by the CPU time on an Alpha AXP system. A relative CPU speed greater than 1.0 implies that the Alpha AXP system is faster.

used the SPEC Release 1 benchmarks as representative of such applications. Table 3 shows the details of the VAX and Alpha AXP systems on which the SPEC Release 1 suite was run, and Table 4 contains the results. The SPECmark89 comparison shows that the OpenVMS AXP system outperforms the OpenVMS VAX system by a factor of 3.59.

The performance of OpenVMS services and the SPECmark results are consistent with other studies of how operating system primitives and SPECmark results scale between CISC and RISC.[6] Overall, the results are very encouraging for a first-version product in which redesigns were purposely limited to meet an aggressive schedule.

## Conclusions

Some OpenVMS VAX features such as symmetric multiprocessing and VMScluster support were deferred from the first version of the OpenVMS AXP system. Beyond this, we anticipate taking significant steps to better exploit the hardware architecture, including evolving to a true 64-bit operating system in a staged fashion. Also, detailed analysis of performance results shows the need to alter internal designs to better match RISC architecture. Finally, a gradual replacement of VAX MACRO-32 source with a high-level language is essential to support a 64-bit virtual address space and is an important element for increasing performance.

The OpenVMS AXP system clearly demonstrates the viability of making dramatic changes in the fundamental assumptions of a mature operating system while preserving the investment in software layered on the system. The future challenge is to continue operating system evolution in order to provide more capabilities to applications while maintaining that essential level of compatibility.

## Acknowledgments

The work described in this paper was done by members of the OpenVMS AXP operating system group. This work would have been impossible without the help of many software and hardware engineering groups at Digital. Thanks to Bradley Waters, who measured OpenVMS performance, and to John Shakshober and Sandeep Deshmukh, who obtained the SPEC Release 1 benchmark results. We also thank Barbara A. Heath and Kathleen D. Morse for their comments, which helped in preparing this paper.

## References

1. R. Sites, "Alpha AXP Architecture," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 19–34.

2. D. Bhandarkar and D. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," *Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-IV)* (New York, NY: The Association for Computing Machinery, 1991): 310–319.

3. T. Leonard, ed., *VAX Architecture Reference Manual* (Bedford, MA: Digital Press, 1987).

**Table 3 Configuration Details for the SPEC Release 1 Benchmark Test Environment**

| | VAX System | Alpha AXP System |
|---|---|---|
| Hardware | | |
| Model number | VAX 7000 Model 610 | DEC 7000 Model 610 |
| Clock rate | 91 MHz | 182 MHz |
| Backup cache size | 4MB I- and D-caches | 4MB I- and D-caches |
| Memory size | 128MB | 256MB |
| Software | | |
| Operating system and version | OpenVMS V5.5-2 Field Test | OpenVMS V1.0 |
| Compilers and version | VAX C V3.2<br>VAX FORTRAN V5.7 with HPO V1.3 (high-performance option) | Pre-release C compiler<br>Pre-release FORTRAN compiler |
| Other software | KAP V1.0 for VAX C and FORTRAN | KAPF/KAPC V1.49 native KAP for Alpha AXP systems |

Key:
I Instruction
D Data

Note that DECram, a memory-resident disk device, was used to create and manage memory-resident disks.

**Table 4 SPEC Release 1 Benchmark Results**

| SPEC Benchmark Name and Number | VAX 7000 Model 610 SPECratio | DEC 7000 Model 610 SPECratio | Relative Performance |
|---|---|---|---|
| 001.gcc | 34.9 | 67.5 | 1.93 |
| 008.espresso | 28.8 | 94.7 | 3.29 |
| 013.spice 2g6 | 30.9 | 87.7 | 2.84 |
| 015.doduc | 42.1 | 126.3 | 3.00 |
| 020.nasa7 | 67.2 | 293.0 | 4.36 |
| 022.li | 34.7 | 100.2 | 2.89 |
| 023.eqntott | 38.4 | 127.6 | 3.32 |
| 030.matrix300 | 138.8 | 1219.7 | 8.79 |
| 042.fpppp | 48.8 | 193.8 | 3.97 |
| 047.tomcatv | 61.6 | 276.5 | 4.49 |
| SPECint89 | 34.0 | 95.1 | 2.80 |
| SPECfp89 | 57.6 | 244.2 | 4.24 |
| SPECmark89 | 46.6 | 167.4 | 3.59 |

Note that relative performance represents the ratio of DEC 7000 Model 610 performance to VAX 7000 Model 610 performance.

4. *OpenVMS Calling Standard* (Maynard, MA: Digital Equipment Corporation, October 1992).

5. *Spec Newsletter,* vol. 4, no. 1 (March 1992).

6. T. Anderson, H. Levy, B. Bershad, and E. Lazowska, "The Interaction of Architecture and Operating System Design," *Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-IV)* (New York, NY: The Association for Computing Machinery, 1991): 108–120.

## General References

R. Goldenberg and S. Saravanan, *VMS for Alpha Platforms Internals and Data Structures,* Preliminary edition of vols. 1 and 2 (Maynard, MA: Digital Press, 1993, forthcoming).

J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach* (San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990).

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

*David S. Blickstein*
*Peter W. Craig*
*Caroline S. Davidson*
*R. Neil Faiman, Jr.*
*Kent D. Glossop*
*Richard B. Grove*
*Steven O. Hobbs*
*William B. Noyce*

# The GEM Optimizing Compiler System

*The GEM compiler system is the technology Digital is using to build state-of-the-art compiler products for a variety of languages and hardware/software platforms. Portable, modular software components with carefully specified interfaces simplify the engineering of diverse compilers. A single optimizer, independent of the language and the target platform, transforms the intermediate language generated by the front end into a semantically equivalent form that executes faster on the target machine. The GEM system supports a range of languages and has been successfully retargeted and rehosted for the Alpha AXP and MIPS architectures and for several operating environments.*

In the past, Digital has made major investments in optimizing compilers that were specifically directed at one hardware platform, namely VAX computers. When Digital began broadening its hardware offerings to include reduced instruction set computer (RISC) architectures, it became clear that new optimization technology was needed, as well as a new strategy for leveraging investments in compiler technology across an increasing number of hardware platforms.

This paper presents a technical description of the GEM compiler technology that Digital uses to generate compiler products for a wide range of hardware and software combinations. We begin with an explanation of the GEM strategy of leveraging investments by using portable, modular software components to build compiler products. The bulk of the paper describes the GEM optimizer and code generator technologies, with a focus on how they address challenges posed by the Alpha AXP architecture.[1] We then move to a discussion of compiler engineering and conclude with an overview of some planned enhancements to the software.

## GEM Compiler Architecture

Because of the many hardware platforms available, often with multiple operating systems and a variety of languages offered on those platforms, building a compiler from scratch for each combination is no longer feasible. To simplify the engineering of diverse compilers, GEM compiler products share a basic architecture. The compiler is divided into several major components, in effect, the fundamental building blocks from which a compiler is constructed. The interfaces among these components are carefully specified. The major components of a GEM compiler are the front end, the optimizer, the code generator, and the compiler shell. The logical division of GEM components and the range of GEM support is shown in Figure 1. Note that the *host* is the computer on which the compiler runs, and the *target* is the computer on which the generated object runs.

The *front end* performs lexical analysis and parsing of the source program. The primary outputs are *intermediate language* (IL) graphs and symbol tables, which are both standardized. In an IL graph, each node, referred to as a *tuple,* represents an operation. Front ends for all source languages translate to the single standard IL. All language-specific code is encapsulated in the front end. All knowledge of the source language is communicated in the IL or through callbacks to the front end. Knowledge of the target hardware is represented in tables and in a minimal amount of conditional code.

The *optimizer* transforms the IL generated by the front end into a semantically equivalent form that will execute faster on the target machine. A significant technical achievement is that a single optimizer is used for all languages and target platforms.

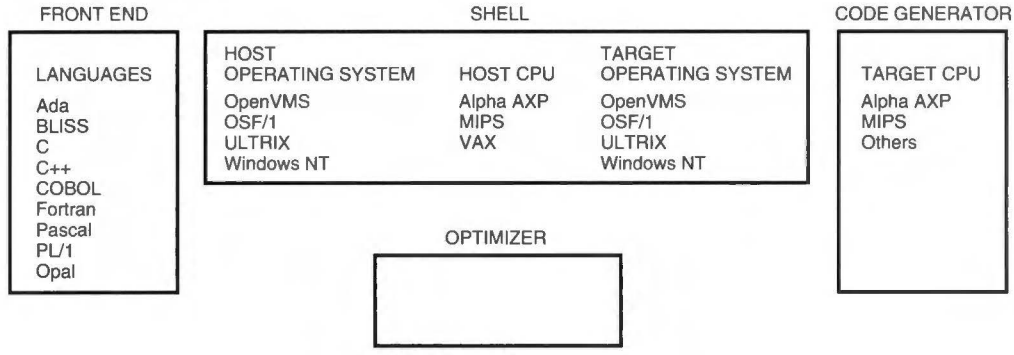



*Figure 1 GEM Components and Supported CPUs, Operating Systems, and Languages*

The *code generator* translates the IL into a list of *code cells,* each of which represents one machine instruction for the target hardware. Virtually all the target machine instruction-specific code is encapsulated in the code generator.

The *shell* is a collection of common compiler functions such as listing generators, object file emitters, and command line processors. Basically, the shell is a portable interface to the external environment in which the compiler is used. This interface allows the other components to remain independent of the operating system.

There are numerous benefits to this modular approach:

- Adding a new feature to a common component enhances many products.
- Source language compatibility is ensured among all compilers that use the same front end.
- Standardized interfaces enable us to plug in a new front end to build a compiler for a new language, or a new shell to allow the compiler to run on a new host.
- When a new language is added, it can be offered quickly on many platforms.
- When a new target CPU or operating system is supported, many languages are immediately available to that target.

## Order of Processing

When compiling a program, the overall order of processing must be carefully arranged so that each component of the compiler can see a large portion of the program at one time. When processing one portion of a program, certain information about other relevant parts of the source program can be useful.

Figure 2 illustrates the overall process of compiling a program. Since GEM compilers include interprocedural optimizations, as much of the program as possible should be presented to the optimizer at the same time. For this reason, GEM compilers allow the user to process multiple source files in a single compilation. The front end parses these source files and constructs the symbol table and a compact form of IL in memory before invoking the GEM back end. The portion of the user's program thus compiled is called a compilation unit.

The GEM back-end interprocedural optimization phase is the first to operate on the program. This phase analyzes the routines within a compilation unit to develop a call graph that shows which routines might call which other routines. Interprocedural optimizations are applied to the routines as a group.

Next, the global optimizer and the code generator process each routine in a bottom-up order, resulting in a translation of the program to code cells that represent operations at machine level. This bottom-up order is convenient for certain optimizations, as discussed in the Optimization section. The first action of the global optimizer is to translate the routine's IL from the compact form provided by the front end to an expanded form used by the optimizer and the code generator. Since only one routine at a time is stored in expanded form, a much larger data structure can be used to store the results of the optimizer analysis. The expansion from compact form also expands certain shorthand forms, which are convenient for a front end, into explicit operations in the expanded IL, much like a macro expansion facility in a source language.

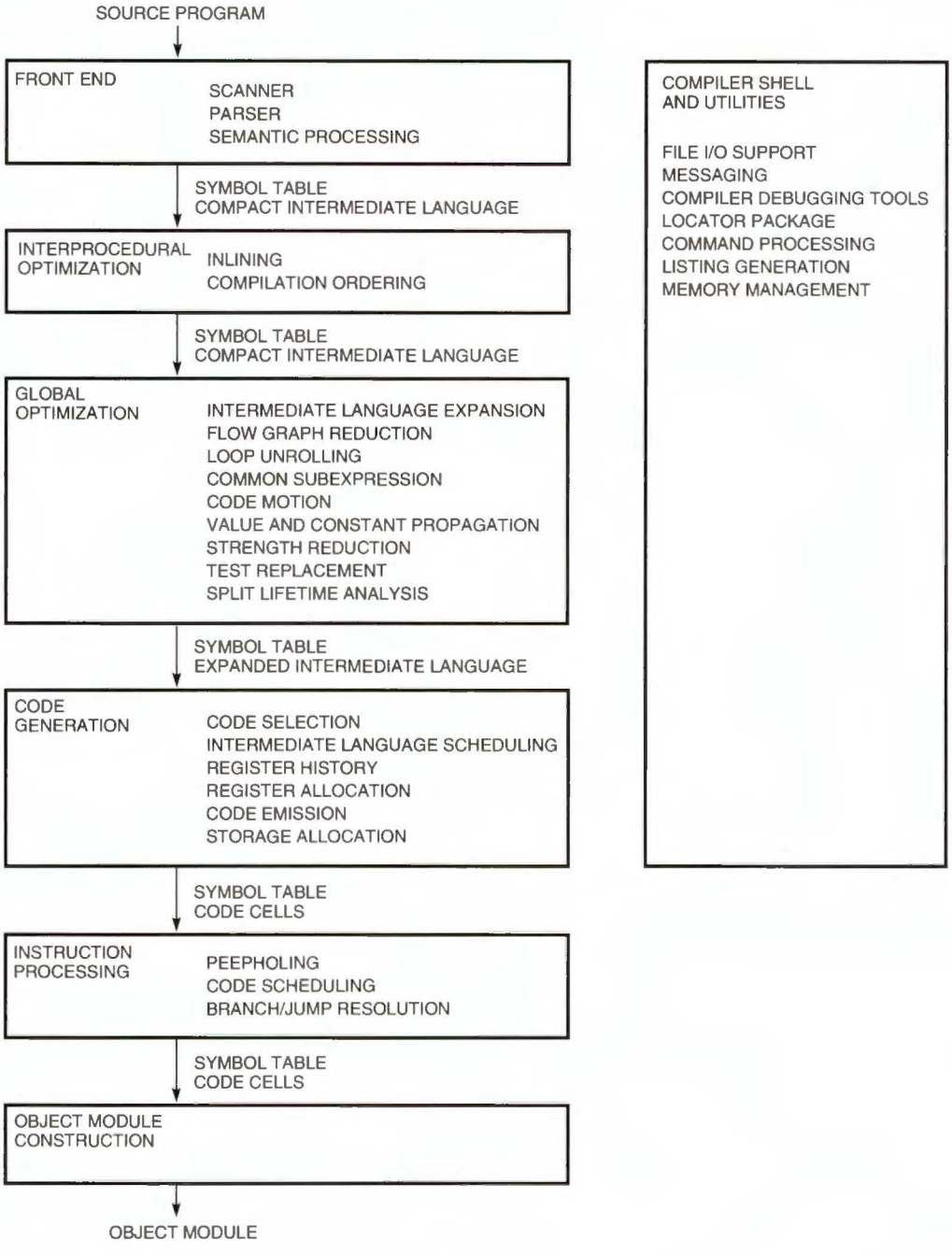



*Figure 2 GEM Compiler Order of Processing*

Once all the routines have been processed by the global optimizer and the code generator, a complete description of the program is available at the machine instruction level. Certain machine-specific optimizations, such as peephole optimizations and instruction scheduling, are performed on this program description. Finally, the optimized machine instructions are converted to the appropriate object language for the target operating system.

## Optimization

The GEM compiler system's optimizer is state-of-the-art and independent of the language and the target platform. The input to the optimizer is the IL and symbol table for multiple routines; the output is the semantically equivalent IL and symbol table, both modified to run faster on the target platform.

GEM optimizations include interprocedural optimizations, modern optimizations for superscalar RISC architectures such as the Alpha AXP architecture, plus a robust implementation of the classical global optimizations. In addition, GEM's code generator includes a number of optimization features that help it produce extremely high local code quality.

## Design Principles

Certain general design approaches or principles were applied throughout the optimizer. For instance, choices had to be made in the design of the IL; the front end could either provide a higher-level description of program features or rely on the back end to derive the higher-level description from an analysis of a lower-level description. In cases where accurate, well-defined algorithms for deriving those higher-level features exist, GEM chooses to derive the descriptions.

Describing source code loops is a key example of the implementation of this design principle. Most source languages have explicit syntax for writing loops, and the front end could translate these languages into a higher-level IL that designates those loops. Instead, GEM uses a lower-level IL with primitives such as conditional branch and label operators. The advantage of this approach is that GEM recognizes all loops, even those constructed with GOTO statements.

A general design approach that emerged from experience gained during the GEM project is the use of enabling or expanding transformations to support fundamental optimizations. Often, representing operations in the IL in a way that hides certain implicit operations is a compact and efficient approach. However at times, making these implicit operations explicit allows a particular optimization routine to operate on them. A good solution to this problem is to initially represent the operations in the IL in the compact form. Then, before applying optimizations that could benefit from seeing the implicit operations, apply expanding transformations to convert the IL into a longer form in which all operations are explicit.

Out of concern for the time required to compile large programs, GEM also established the design principle that the order of complexity as a function of the number of IL operations should be as close to linear as possible.

## Data Access Model and Side Effects Interface

Since GEM compilers translate all source languages to a common IL and symbol table format, the semantics of these languages must be specified precisely. Many optimizations require an exact understanding of which symbols are being written or read by operations in the IL, and which operations might affect the results computed by other operations.

The GEM team developed a detailed specification known as the *data access model,* which defines those operations that can write to memory and those that can read from memory. Each of these memory-accessing operations can explicitly designate the symbol being accessed when it is known. The model also requires the front end to specify when symbols may be aliased with parameters and when they may be *pointer aliased.* A pointer-aliased symbol may be accessed through pointers or other operations that do not specify the symbol that they access.

The model can indicate that the pointer-aliased property is *derivable,* i.e., a symbol is pointer aliased only if an operation that stores its address is present in the IL. A special IL operator marks such operations. When the derivation of this property is deferred, the optimizer can avoid marking symbols pointer aliased.

The data access model provides a standard way for a front end to indicate how IL operations affect or depend upon symbols. However, some front ends can provide additional language-specific discrimination of operations that cannot be allowed to interfere with one another. For example, a strongly typed language like Pascal may stipulate that an assignment to a floating-point target must refer to different storage than an integer read, even when the assignment target is accessed indirectly through a pointer.

To represent language-specific rules while adhering to the philosophy that the back end should have no knowledge of the source language, GEM compilers employ a unique interface with the front end, called the *side effects interface.* The front end provides a set of procedures that GEM can call during

optimization to ask which IL operations have side effects and which IL operations depend upon those side effects.

### *Interprocedural Optimization*

GEM's interprocedural optimization phase starts by walking over the IL for all routines to build the call graph. The call graph is a directed multigraph in which the nodes are routines, and the edges are calls from one routine to another. The graph is not a tree because recursion is allowed. A special virtual routine node represents all unknown routines that might call or be called by a routine in this compilation.

GEM walks the graph to determine which local symbols that are potential targets of up-level access are actually referenced in a called routine. When up-level references do occur, GEM can also determine the most efficient way to pass that context from the routine that declares the variable to the routine that references it.

On the same walk, GEM analyzes the use of symbols whose pointer-aliased property is derivable. If operations that store the address of such a symbol are present, then the symbol is marked as pointer aliased. The front end's indication is also retained so that this analysis can be repeated after address storing operations are eliminated.

The most significant interprocedural optimization that GEM performs is procedure inlining. Inlining is a well-known method for reducing procedure call overhead and for increasing the effectiveness of global optimizations by enlarging the scope of the operations seen at one time. Inlining has additional benefits on superscalar RISC architectures, like the Alpha AXP system, because the optimization allows the compiler to schedule the instructions of the two routines together.

GEM's inliner reviews all calls in the call graph and uses heuristic algorithms to determine which calls should be inlined for maximum speed without unreasonable increases in code size or compilation time. The heuristics consider the number and kind of IL operations, the number of symbols referenced, and the kinds of optimization that would likely be enabled or disabled by inlining.

When callers pass constants as actual parameters, better optimization is likely to result from inlining because the corresponding formal parameter will have a known constant value. On the other hand, when two sections of the same array are passed as arguments, and the corresponding formals are described as not aliased with one another, eliminating the formal parameters through inlining discards valuable alias information.[2,3]

Also, the order in which inlining decisions are made can be important. In a chain of calls in which A calls B and B calls C, the call from A to B might be the most desirable inlining candidate. However, if the call from B to C is inlined first, the size of B may increase, making it a less attractive candidate for inlining into A. Consequently, GEM uses its heuristics to preevaluate all calls and then orders the calls by desirability. GEM inlines the most desirable candidate first, and then reevaluates the caller's properties, possibly adjusting its position in the ordered list.

In many C programs, the address of a variable (especially a struct) is passed to a called routine that refers to the variable through a pointer formal parameter. After inlining, a symbol's address is stored in a pointer and indirect references are made through the pointer. Later, while optimizing the routine, GEM's value propagation often eliminates the pointer variable. Finally, when one or more pointer-storing operations have been eliminated, GEM reevaluates the pointer-aliased property of derivable local symbols, and the variable that was once passed by address is no longer pointer aliased.

After interprocedural analysis, the routines of the user's program pass through the optimizer and code generator one at a time. GEM's interprocedural phase chooses a bottom-up routine order in the call graph. Except for recursive cycles, this order causes GEM to generate the code for a called routine before generating the caller's code. GEM takes advantage of this property by recording the scratch registers that were actually used in a called routine and adjusting register usage at its call sites.[4] GEM also determines whether or not the called routine requires an argument count.

### *Intermediate Language Peepholes*

GEM uses a peephole optimizer to improve the IL. In addition to performing the many obvious simplifications such as multiplying by one or adding zero, the optimizer performs other transformations. Integer division by a constant is expanded into a multiply by a reciprocal operation, which can be efficiently implemented with a UMULH instruction. String operations on short fixed-length strings are converted into integer operations, to benefit from

various optimizations performed only on scalars. Also, integer multiply operations by a constant are converted into an equivalent set of shift and add or subtract operations.

IL peepholes sometimes expose new optimization opportunities by expanding complex operations into more explicit components. Also, other optimizations such as value propagation may create new opportunities to apply peepholes. To take advantage of these opportunities, GEM compilers apply these IL peepholes multiple times during the optimization of a routine.

### *Data-flow Analysis*

In previous Digital compilers, the use of data-flow analysis was limited largely to the elimination of common subexpressions (CSEs), value propagations, and code motions. We generalized the data-flow analysis technique to perform a wider variety of optimizations including field merging, induction variable detection, dead store elimination, base binding, and strength reduction.

The process of detecting CSEs is divided into the tasks of

- Knowing when two expressions would compute the same results given identical inputs. Within GEM compilers, such expressions are said to be *formally equivalent.*
- Verifying that the inputs to formally equivalent subexpressions are always identical. Such expressions are said to be *value equivalent.* This verification is accomplished by using the side effects mechanism.
- Determining when an expression dominates a value equivalent expression.[5] This information guarantees that GEM will have computed the dominating expression whenever the dominated expression is needed.

Code motions introduce the additional task of finding those places in the flow graph to which an expression could be legally moved such that

- The moved expression would be value equivalent to the original expression, and
- The moved expression would execute less often than the original expression.

The following sections describe how GEM detects base-binding and strength-reduction candidates by substituting slightly different equivalence functions.

### *Base Binding*

On RISC machines, a variable in memory is referenced by loading the address into a base register and then using indirect addressing through the base register. To reduce the number of address loads, sets of variables that are closely allocated share base registers. GEM considers two address expressions formally equivalent if they differ by an amount less than the range of the hardware instruction offset field. The CSE detection algorithm determines which address expressions are formally equivalent and thus can share a base register, and the code motion algorithm moves the base register loads out of loops.

### *Induction Variables*

Some of GEM's most valuable optimizations require the identification of inductive expressions and induction variables, which is done during data-flow analysis. An expression in a loop is inductive if its value on a particular iteration is a linear function of the trip count. The simplest forms of inductive expressions are the control variables of counted loops. Expressions that are linear functions of induction variables are also inductive.

GEM's implementation of data-flow analysis uses a technique for determining what variables are modified between basic blocks in the flow graph.[6,7] The variables modified between a basic block and its dominator are represented as a set called the *IDEF set.* The mapping from variables to set elements is done using the side effects interface.

The algorithm for detecting induction variables starts by presuming that all variables modified in the loop are induction variable candidates. It then disqualifies variables not redefined as a linear function of themselves with a coefficient equal to one. The loops that GEM chooses to analyze have a loop top that dominates all nodes within the loop. The IDEF set for a loop top is exactly those variables that are modified within the loop and thus serves as the starting value for the induction variable candidate set, again using the side effects mapping of variables to set elements. During the walk of the loop, whenever a disqualifying store is encountered, the contents of the candidate set are updated. Thus, at the end of the walk, the remaining variables in the set are known to be true induction variables.

### *Strength Reduction of Induction Variables*

Strength reduction is the process of replacing an expensive operation with a less expensive operation. The most basic example of strength reduction on induction is as follows:

If the original source program was

```
      DO 20 I = 1,10
20    PRINT I*4
```

strength reduction would reduce the multiply to an add as follows:

```
      I' = 4
      DO 20 I = 1,10
      PRINT I' 20
      I' = I' + 4
```

Note that the most common array references are of the form A(I), which implies a multiplication of I by the stride of the array. Thus, strength reduction yields a significant performance improvement in array-intensive computations.

To detect strength-reduction candidates, we redefine formal and value equivalence as follows:

- Two inductive expressions are formally equivalent if, given identical inputs, they differ only by a constant.
- Two formally equivalent inductive expressions are value equivalent if their inputs are value equivalent or are direct references to induction variables.

Thus, strength-reduction candidates appear loop invariant, and two expressions that are value equivalent can share a single strength reduction. Code motion yields the initial value of the strength reduction.

### *Split Lifetime Analysis*

The GEM optimizer analyzes the usage of certain variables to determine if the stores and fetches of a variable can be partitioned, i.e., split, into disjoint variables or *lifetimes.*

For example, consider the following program segment:

```
1: V = X * Y
2: Z = Z * V

3: V = R + S
4: T = T + V
```

The references to V can be divided into two disjoint lifetimes V′ and V′′ without changing the semantics of the program as in:

```
1: V' = X * Y
2: Z  = Z * V'

3: V" = R + S
4: T  = T + V"
```

V′ and V′′ can be treated as two completely independent variables. This has several useful applications.

- V′ and V′′ can be assigned to different registers, each with shorter lifetimes than the original variable V. The allocator can thus pack registers and memory more tightly.
- V′ and V′′ can be scheduled independently. For example, the computation of Z in line 2 could be scheduled after the redefinition of V in line 3.
- A lifetime that begins with a fetch is an uninitialized variable. GEM issues a diagnostic in such cases.
- Any lifetime with only stores is effectively "dead," and thus, the stores can be eliminated.
- When a lifetime of an induction variable contains an equal number of stores and fetches, the variable is used only to compute itself. Thus, the whole lifetime can be eliminated. This is called induction variable elimination.
- GEM uses split lifetime information to optimize the flushing and reloading of register variables around routine calls.
- GEM uses split lifetime information to determine what variables are potentially referenced by exception handlers.
- Lifetimes often need to be extended around loop tops and loop bottoms. Split lifetime analysis has full information in many cases in which the code generator's lifetime computation must make pessimistic assumptions. Thus, analyzed variables are allocated more efficiently inside loops.

The technique GEM uses for split lifetime analysis is based on the VAX Fortran SPLIT phase.[8] The technique includes several extensions in the areas of induction variables, unselected variables (the original algorithm analyzed only a fixed number of variables), and exception handling.

## *Code Generation*

The GEM code generator matches code templates to sections of IL trees.[9] The code generator has a set of approximately 600 code patterns and uses dynamic programming to guide the selection of a least-cost covering for each statement tree in the IL graph produced by the global optimizer.

Each code pattern specifies a set of interpretive code-generation actions to be applied if the template is selected. The code-generation actions create temporaries, determine their lifetimes, allocate registers and stack locations, and actually emit sequences of instructions. These actions are applied during the following four separate code-generation passes over the IL graph for a procedure:

- Context. During the context pass, the code generator creates data structures that describe each temporary variable. The information computed includes the lifetime, usage counts, and a weight scaled by loop depth.
- Register history. During the register history pass, the code generator does a dominator-order walk of the flow graph to identify potential redundant loads of values that could be available in registers.
- Temp name. During the temp name pass, the code generator performs register allocation using the lifetime and weight information computed during the context pass. The code generator also uses register history to allocate temporaries that hold the same value in the same register. If successful, this action eliminates load and move instructions.
- Code. During the code pass, the code generator emits instructions and code labels. The resulting code cells are an internal representation at the assembly code level. Each code cell contains a single target machine instruction. The code cells have specific registers and bound offsets from base registers. References to labels in the code stream are in a symbolic form, pending further optimization and final offset assignment after instruction peephole optimization and instruction scheduling.

### *Template Matching and Result Modes*

Code template enumeration and selection occurs during the context pass. The enumeration phase scans IL nodes in execution order (bottom-up) and labels each node with alternative patterns and costs. When a root node such as a store or branch tuple is reached, the lowest-cost template for that node is selected. The selection process is then applied recursively to the leaves for the entire tree.[10]

The IL tree pattern of a code-generation template consists of four pieces of information:

- A pattern tree that describes the arrangement of IL nodes that can be coded by this template. The interior nodes of the pattern tree are IL operators; the leaves are either result mode sets or IL operators with no operands.
- A predicate on the tree nodes of the pattern. The predicate must be true in order for the pattern to be applicable.
- A result mode that encodes the representation of a value computed by the template's generated code.
- An integer that represents the cost of the code generated by this template.

The result modes are an enumeration of the different ways the compiler can represent a value in the machine.[11] GEM compilers use the following result modes:

- Scalar, for a value, negated value, and complemented value
- Boolean, for low-bit, high-bit, and nonzero values
- Flow, for a Boolean represented as control flow
- Result modes for different sizes of integer literals
- Result modes for delayed generation of addressing calculations
- Result modes indicating that only a part of a value has been materialized, i.e., the low byte, or that the materialized value has used a lower-cost solution

As templates are matched to portions of the IL tree, each node is labeled with a vector of possible solutions. The vector is indexed by result mode, and the lowest-cost solution for each result mode is recorded on the forward bottom-up walk. When a root node is encountered, the lowest-cost template in its vector of solutions is chosen. This choice then determines the required result mode and solution for each leaf of the pattern, recursively.

### *GEM Code Generator Action Language*

The GEM code generator uses and extends methods developed in the BLISS compilers, the Carnegie-Mellon University Production-Quality Compiler-Compiler Project, and Digital's VAX Pascal compiler.[12,13] One key GEM innovation is the use of a formalized action language to give a unified description of all actions performed in the four code-generation passes. The same formal action descriptions are interpreted by four different interpreters. For example, the Allocate_TN action is used to allocate long-lived temporaries that may be in a register or in memory. This action creates a data structure describing the temporary in the context pass, allocates a register during the temp name pass, and provides the actual temporary location for code emission.

Tree-matching code generators were originally developed for complex instruction set computer (CISC) machines, like the PDP-11 and VAX computers. The technique is also an effective way to build a retargetable compiler system for current RISC architectures. The overall code-generation structure and many of the actions are target independent. Some IL trees use simple, general code patterns, whereas special cases use more elaborate patterns and result modes.

## *Register Allocation*

GEM compilers use a simple linear model to characterize register lifetimes. The context, temp name, and code passes process the basic blocks and the IL nodes of each block in execution order. Each code pattern has a certain number of lifetime ticks to represent points at which a temporary value is created or used. The lifetime of a temporary is then the interval defined by its starting lifetime tick and ending lifetime tick.

Simple expression temporaries have a linear lifetime contained within a basic block. User variables and CSEs may require that lifetimes be extended to cover loop tops and loop bottoms. The optimizer inserts special begin and end markers to delimit the precise lifetimes of variables created by the split lifetime phase.

The code generator uses a number of heuristics to allocate registers to avoid copying. If a new lifetime begins at exactly the same tick as another lifetime ends, this may indicate that they should share a register. Otherwise, the allocator uses a round-robin allocation to avoid packing registers too tightly, which would inhibit scheduling. The Move_Value action is used to copy one register to another and provides a hint that the source and destination should be allocated to the same register.

Actual allocation of registers and stack temporaries occurs in the temp name pass. The allocator uses a bin-packing technique to allocate each compiler and user variable to a register or to memory.[14] The allocator first attempts to assign variables to registers; lifetimes that conflict cannot be assigned to the same register. The allocator uses a density function to control the process. A new candidate can displace a previous variable that has a conflicting lifetime if this action increases the density measure. After the allocation of temporaries to registers is completed, any unallocated or spilled temporaries are allocated to stack locations.

## *Scheduling*

To take advantage of high instruction-issue rates in Alpha AXP systems, compilers must carefully schedule the object code, interleaving instructions from several parts of the program being compiled. Performing instruction scheduling only once after registers have been allocated places artificial constraints on the ordering, as illustrated in the following code example:

```
ldq     r0, a(sp)    ; Copy a to b
stq     r0, b(sp)
ldq     r0, c(sp)    ; Copy c to d
stq     r0, d(sp)
```

If the load of c and store of d were to use some other register, the code could be rescheduled to save three cycles on the DECchip 21064 processor, as shown in the following code:

```
ldq     r0, a(sp)    ; Copy a to b
ldq     r1, c(sp)    ; Copy c to d
stq     r0, b(sp)
stq     r1, d(sp)
```

On the other hand, scheduling only before register allocation does not incorporate decisions made by the code generator. Therefore, instruction scheduling in GEM compilers occurs twice, before and after registers are allocated. This practice is fairly common in contemporary RISC compiler systems. In most other systems, scheduling is performed only on machine code. GEM has two different schedulers—one that schedules machine code and one that schedules IL.

## *Intermediate Language Scheduling*

IL scheduling is performed one basic block at a time. First, a forward pass over the block gathers information needed to control the scheduling, and then a backward pass builds the new ordered list of tuples. During the forward pass, the compiler builds dependence edges to represent the necessary ordering relationships between pairs of tuples. Tuples that would require an excessive number of edges, such as CALL tuples, are considered markers. No tuples can be reordered across a marker.

The compiler uses the data access model to determine whether two memory-access tuples conflict. Also, if two tuples have address operands with the same value (using data-flow information) but different offset attributes, the tuples must access different memory. Thus, no dependence edge is needed, and more rescheduling is possible.

The general code for an expression tuple places the result into a compiler-generated temporary, and the general code for a store into a register variable moves the value from a temporary into the variable. Many GEM code patterns for expression tuples allow *targeting,* where the expression is computed directly into the variable instead of into a temporary. These code patterns are valid only if there are no fetches of the variable between the expression tuple and the store operation. Similarly, a fetch tuple need not generate any code (called *virtual*), if no stores exist between the fetch and its consumer. For example,

```
T = A-1; A = B+1; C = T;
```

might generate the GEM IL

```
1$: FETCH(A)
2$: SUB(1$, [1])
3$: FETCH(B)
4$: ADD(3$, [1])
5$: STORE(A, 4$)
6$: STORE(C, 2$)
```

In this example, SUB operates directly on the register allocated for A, and ADD targets its result to the register allocated for A. The obvious dependence edge is from FETCH(A) to STORE(A,...). However, IL scheduling must be careful not to invalidate the code patterns, which would happen if it moved FETCH(A) between ADD and STORE(A) or STORE(A) between FETCH(A) and SUB. To ensure valid code patterns, the first pass moves the head of dependence edges backward from targeted stores to the expression tuple that does the targeting. Similarly, the first pass moves the tail of dependence edges forward from virtual fetches to their consumers. In this example, the edge runs from 2$ to 4$ and prevents either of the illegal reorderings.

In addition to building dependence edges, the first pass computes heuristics for each tuple, to be used by the second, i.e., scheduling, pass. One heuristic, the anticipated execution time (*AET*), estimates the earliest time at which the tuple could execute. The AET for tuple T is either the maximum AET of any tuple that must precede T, or the maximum AET plus the latency of T's operands. If some of the tuples that must precede T require the same hardware resources, the AET may be optimistic. Nevertheless, the AET is a useful guide to the scheduling pass.

The first pass also computes the minimum number of registers (separately for integer and floating-point registers) needed to evaluate the subexpression rooted at a particular tuple. The value of this heuristic is the Sethi-Ullman number, i.e., the number of registers needed to evaluate the subexpressions in the optimal order, keeping their intermediate values, plus the additional registers to evaluate the tuple itself.[15] If the second pass schedules tuples with a lower count later in the program, the register usage will be kept low. Without such a mechanism, scheduling before register allocation tends to cause excessive register pressure.

CSEs can be treated similarly to subexpressions in this computation, but with two complications. The first pass cannot predict the last use of the CSE and therefore treats each use as the last one. The scheduler ignores any register usage associated with CSEs that are not both created and used within the block being scheduled. This action allows the register allocator to place the CSEs in memory, if the scheduled code has better uses for registers.

The second pass of the IL scheduler works backward over the basic block. The scheduler removes all the tuples up to the last marker and makes available those that have no dependence edges to tuples that must follow. The scheduler then selects an available tuple and places it in the scheduled output, updates the state of each modeled functional unit, and makes available new tuples whose dependences are now satisfied. When the marker is scheduled, the scheduler continues to remove the preceding group of tuples from the block until the entire block has been scheduled.

The scheduler keeps track of the number of scheduled cycles and the estimated number of live registers. When choosing among tuples, the scheduler prefers one whose subtree can be evaluated within the available registers, or, failing that, one whose subtree can be evaluated with the fewest registers. When several tuples qualify, the scheduler chooses the one with the greatest AET.

Limiting register pressure, while not important for all programs, is important in blocks with a lot of available parallelism. With this feature, IL scheduling is a significant contributor to the high performance of GEM-compiled programs.

## *Instruction Peepholing*

After code has been generated or code cells have been created directly, the instruction processing phases are run as a group. Instruction peepholing performs a variety of localized transformations, typically by matching patterns of adjacent instructions and replacing them with better patterns. From the perspective of instruction scheduling, the most

interesting function of the instruction peepholer is to perform a set of branch reductions. The peepholer also replicates short sequences of code to facilitate instruction scheduling and to eliminate the instruction pipeline effects of branches.

A control flow processing phase follows the instruction peepholing phase. Currently, this phase determines labels that are backward branch targets for alignment purposes. This action occurs before instruction scheduling, because instruction alignment is important for the DECchip 21064 Alpha AXP processor, in which instructions must be aligned on quadword boundaries to exploit dual instruction issue. In the near future, the control flow processing phase will collect register information for each basic block to allow additional scheduling transformations.

### *Instruction Scheduling*

The instruction scheduler is the next phase. At this point, all register binding and code modifications other than branch/jump resolution have occurred. The scheduler does a forward walk over the basic blocks in each code section to determine the alignment of the first instruction in each block.

For each basic block, the instruction scheduler does two passes that are effectively the inverse of the passes that the IL scheduler performs, namely a backward walk to determine instruction-ordering requirements and path length to the end of the block, and a forward pass that actually schedules the code.

The backward ordering pass uses an AET computation similar to the one used by the IL scheduler. The instruction scheduler knows the actual instructions to be scheduled and has a more detailed machine model. For the DECchip 21064 processor, for example, the instruction scheduler has detailed asymmetric bypassing information and information about multiple issue. For architectures that have branch delay slots, the AET computation is biased so that instructions likely to be able to fill branch delay slots will occur immediately before branch operations.

The forward scheduling pass does a cycle-by-cycle model of the machine, including modeling multiple issue. The reasons for choosing this approach rather than an approach that just selects an ordering of the instructions are as follows:

- For machines with significant issue limitations, e.g., nonpipelined functional units or multiple issue pairing rules, packing the limiting resource well is often preferable to obtaining a good schedule. A cycle model allows other instructions to "float" into the no-issue slots, while allowing the critical resource to be scheduled well.
- Modeling the machine allows easy determination of where stalls are occurring, which in turn allows instructions from the current block or from successor blocks to be moved into no-issue slots.
- Modeling the machine in a forward direction captures the fact that processors are typically "greedy" and issue all the instructions that they can issue at a given time.
- The cycle model allows a variety of dumps, which can be useful both to users of the compiler system and to developers who are trying to improve the performance of generated code.

The forward pass does a topological sort of the instructions. The scheduler moves instructions that have either a direct dependence or an antidependence (e.g., register reuse) to a data structure called the issuing ring for future issue.

The scheduler represents the instructions available for issuing as a list of data structures known as heaps, which are priority queues. Each heap on the list contains instructions with a similar "signature." For example, a heap might contain all store instructions. When looking for the next instruction to issue, the scheduler examines the top instruction in each heap. Within each heap, instructions are typically ordered by their AET values, with occasional small biases for different instruction properties, such as loads that may have a variable execution time longer than the projected time.

The heaps are, in turn, ordered in the list according to how desirable it is that a particular heap's top instruction be issued. All nonpipelined instruction heaps are first on the list, followed by all semipipelined heaps and, last, all fully pipelined ones. A semipipelined resource may prevent particular instructions from issuing in certain future cycles but can issue every cycle. For example, stores on some machines interact with later loads.

Instructions that use multiple resources are represented in the heap ordering. For example, floating-point multiplies on the MIPS R3000 machine use both the multiplier and some of the same resources as additions. As a result, the heap that holds multiplies is always kept ahead of the heap that holds adds. This ordering scheme works well for both machines with a significant number of nonpipelined units, such as the MIPS processors,

and machines that have largely pipelined functional units with only particular combinations of multiple issue allowed, like the DECchip 21064 processors.

Note that, other than the architecture-specific computation for AET and per-processor implementation data tables, the scheduler is completely target independent. For example, currently, processor implementation tables exist for the MIPS R3000 and R4000 processors, the DECchip 21064 processor, and Alpha AXP processors that are under development.

## *Field Merging Example*

Generating efficient code for the extraction and insertion of fields within records is particularly challenging on RISC architectures, like Alpha AXP, that provide only 32-bit (longword) or 64-bit (quadword) memory operations.

Often, a program will fetch or store several fields that are contained in the same longword. Without optimization, each fetch would load the longword from memory, and each store would both load and store the longword. However, it is possible to perform a collection of field fetches and stores with a single load and store to memory. As another example, two bit tests within the same longword could be done in parallel as a mask operation.

In the IL generated by the front end, each field operation is generated as a separate IL operation. Thus, the real task of optimizing field accesses is to identify IL operations that can be combined.

In the initial IL, a field fetch or store is represented as an IL operator. The underlying problem is that the redundant loads and stores are not visible in this representation. The first part of the solution involves expanding the field fetch or store into lower-level operators. The IL generated by the front end for two field extractions as shown in (a) of Figure 3 is expanded into the IL shown in (b) of Figure 3. With the loads exposed as fetches, dataflow analysis is now capable of finding the common subexpressions of 1$ and 3$.

Similarly, each field store expands into a fetch of the background longword, an insertion of the new data into the proper position, and a store back. Given two field stores, value propagation can eliminate the second fetch, and then dead-store elimination can eliminate the first store.

In some cases, a program operates on the field and thus eliminates the extract and insert operations. For example, the following example generates the machine code shown in Figure 4.

```
typedef struct node {
        char n_kind;
        char n_flags;
        struct node *xl_car;
        struct node *xl_cdr;
} NODE;

#define MARK  1
#define LEFT  2

void demo(ptr)
  NODE *ptr;
{
    while (ptr) {
        if (ptr->n_kind == 0) {
            ptr->n_flags |= MARK;
            ptr->n_flags &= ~LEFT;
        }
        ptr = ptr->xl_cdr;
    }
}
```

The unoptimized code would contain a load and an extract for each reference to n_kind or n_flags, plus an insert and a store for the latter two references. The optimizer has eliminated two of the

```
1$:     FETCHX(RECORD, [0], [1])  ; Fetch the #1 (low-order) bit
                                  ; from memory
2$:     FETCHX(RECORD, [1], [1])  ; Fetch the #2 bit from memory
```

(a) Pre-field merging IL

```
1$:     FETCH(RECORD)             ; Fetch the longword
2$:     EXTV(1$, [0], [1]);       ; Extract the #1 from the longword

3$:     FETCH(RECORD)             ; Fetch the longword
4$:     EXTV(1$, [1], [1]);       ; Extract the #2 from the longword
```

(b) Post-field merging IL

*Figure 3 Field Merging Example*

```
demo::
    BEQ    ptr, L$5
    NOP
L$7:
    LDL    R0, (R16)     ; Load n_kind and n_flags
    AND    R0, 255, R1   ; Extract n_kind
    BNE    R1, L$9
    MOV    256, R17
    BIS    R0, R17, R17 ; Set MARK (in place)
    MOV    -513, R1
    AND    R17, R1, R17 ; Clear LEFT (in place)
    STL    R17, (R16)    ; Store back
L$9:
    LDL    ptr, 8(R16)
    BNE    ptr, L$7
L$5:
    RET    R26
```

*Figure 4 Machine Code with Field Merging*

three loads, two of the three extracts, both inserts, and one of the two stores.

## Branch Optimization Examples

Branch instructions can hurt the performance of high-performance systems in several ways. In addition to consuming space and causing time to be expended while issuing the instruction, branches can disrupt the hardware pipeline. Also, branches can inhibit optimizations such as code scheduling. Therefore, the GEM compiler system uses several strategies to avoid branches in the IL and generated code or to eliminate some bad effects of branch instructions.

Some branches appear as part of a well-defined pattern that need not inhibit optimizations. GEM uses special operators for these cases. A simple example is the MAX function. For Alpha AXP systems, MAX can be implemented using the CMOVxx instructions, avoiding branch instructions entirely. For other architectures, the main benefit is that the branch does not appear in the IL. A more complicated example involves the so-called "flow-Boolean" operators. Consider the C code example,

```
x = (p && *p) ? *y : *z;
```

which generates the following GEM IL:

```
1$: FETCH(P)
2$: NONZERO(1$)
3$: ANDSKIP(2$)
4$: FETCH(1$)
5$: NONZERO(4$)
6$: LANDC(3$, 5$)
7$: SELTHEN(6$)
8$: FETCH(Y)
9$: FETCH(8$)
10$: SELELSE(9$)
11$: FETCH(Z)
12$: FETCH(11$)
13$: SELC(7$, 10$, 12$)
14$: STORE(X, 13$)
```

The ANDSKIP and LANDC tuples implement the conditional-AND operator. If tuple 2$ is false, tuples 4$ and 5$ are skipped, and the result of the LANDC is false. Otherwise, the LANDC uses the result of tuple 5$.

Similarly, the SELTHEN, SELELSE, and SELC tuples implement the select operator. If tuple 6$ is true, then tuples 8$ and 9$ compute the result, and tuples 11$ and 12$ are skipped. If tuple 6$ is false, then tuples 8$ and 9$ are skipped, and tuples 11$ and 12$ compute the result.

These operators allow programs to represent branching code within the standard basic-block framework but require branches in the generated code, to avoid undesired side effects of the skipped tuples. In some cases, though, GEM can determine that the skipped tuples have no side effects and then converts the operators to an unconditional form, allowing the generated code to be free of branches.

GEM performs other transformations on the IL to eliminate branches and thus enable further optimizations. For example, GEM transforms

```
if (expr) var = 1; else var = 0;
```

into

```
var = ((expr) != 0)
```

Alpha AXP implementations typically include a branch prediction mechanism. Correctly predicted

branches take several cycles less time than mispredicted branches. The fastest conditional branch is one that is correctly predicted not to be taken. GEM uses several strategies to arrange branches for best performance.

GEM selects an order for the basic blocks of a program that may differ from the order in the source program. For each basic block that ends with an unconditional branch, GEM places the target block next, unless that block has already been placed. Similarly, if a basic block within a loop ends with an unconditional branch, a target block within that loop is placed next, if possible. For example,

```
while (--i > 0) {
  if (a[i] != b[i]) return a[i]-b[i];
  a[i] = 0;
}
```

To eliminate the unconditional branch when the loop iterates, GEM transforms the pretested loop into a posttested loop. Since the return statement is outside the loop, the generated code looks like

```
if (--i > 0)
  do {
    if (a[i] != b[i]) goto label;
    a[i] = 0;
  } while (--i > 0);
 ...

label:  return a[i]-b[i];
```

GEM can also unroll loops and thus reduce the number of times the branch back must be executed. More important, GEM often allows operations from different iterations to be scheduled together. Unrolling by four transforms the above loop into a cleanup loop and the main loop into code that resembles

```
do {
    if (a[i] != b[i]) goto label;
    a[i] = 0;
    if (a[i-1] != b[i-1]) goto label;
    a[i-1] = 0;
    if (a[i-2] != b[i-2]) goto label;
    a[i-2] = 0;
    if (a[i-3] != b[i-3]) goto label;
    a[i-3] = 0;
} while (i -= 4);
```

This code executes four fall-through branches and one taken branch, whereas the original code executed four fall-through branches and four taken branches.

Certain code patterns generate code that is likely not to be executed. For example, when the compiler believes that a 16-bit value in memory is apt to be naturally aligned, but may be unaligned, it generates the instructions shown in Figure 5 to load the value, given the address in r0. The code runs quickly for the aligned case, because the branch is correctly predicted to fall through, but gets the correct value for unaligned data, as well. A similar code pattern handles stores.

## Compiler Engineering

Engineering compilers for a large combination of languages and platforms required a considerable number of innovations in the area of project engineering. In this section we describe some of the project methods and tools GEM uses.

### Opal Intermediate Language Compiler

The task of a GEM compiler is to translate a program presented by the front end in the form of an IL graph and symbol table into machine code. In the early stages of GEM development, no front

```
; 3-instruction inline sequence if aligned
;
        ldq_u   r1, (r0)
        extwl   r1, r0, r1
        blbs    r0, 10$
  20$:

; out-of-line sequence to load and merge
;
  10$: ldq_u   r28, 1(r0)
        extwh   r28, r0, r28
        or      r1, r28, r1
        br      r31, 20$
```

*Figure 5 Potentially Unaligned Load Code*

ends existed to generate IL graphs and symbol tables. To fill this requirement, a syntactic specification of the IL and symbol table was designed and an IL assembler called Opal was built to compile this syntax. Opal uses GEM components such as the shell and thus supports a robust set of features including listing generation, object files, include files, debug support, and language editor diagnostics.

Even with the availability of front ends, Opal remains a vital project tool: it allows GEM developers to exercise new features before front-end support is available; front-end developers use Opal to experiment with different IL alternatives; and the Opal syntax serves as the output format of the IL dumper.

### *Attribute and Operator Signature Tables*

GEM tables give a complete description of all GEM data structures, including IL operators and symbol table nodes. The operator signature table contains the operator type, result type, number of operands, and legal operand types for IL operators. The attribute tables describe each component in a node including location, abstract GEM data type, legal values, node type for pointers, and special print formats. When a new attribute is added to the GEM specification, the attribute is described once in the tables and automatically the Opal compiler understands the syntax and semantics, the GEM dump utility is able to dump the attribute, and the GEM integrity checker is able to verify the structure.

### *Automatic KFOLD Builder*

The GEM compiler needs to evaluate constant expressions at compile time, which is referred to as constant folding. GEM's intermediate language has many IL operators and data types. A constant folder is thus a complicated routine with many cases, and the compile-time and run-time results must be identical.

After writing our first, incomplete, handcrafted constant folder, we searched for a method to automate the process. No source language supported all the operators and data types of the GEM IL. The key insight was that there is one language in which IL programs can be written precisely and tersely: the GEM IL itself. Since GEM already embodies knowledge of the code sequences to evaluate every IL operator, no other encoding is needed.

The automatic KFOLD builder is a specialized GEM compiler that uses the standard GEM back end but has a front end that compiles only one program. The KFOLD builder scans the GEM operator signature table and constructs a procedure that contains a many-way conditional branch to select a case based on the IL operator specified in the argument list. Each case fetches operand values from the argument list, applies the operator, and returns the result. Since most GEM IL tuples operate on several data types, additional subcases may be based on the operator type or result type. We have already recovered the investment in developing the automatic KFOLD builder, and it significantly eases the task of retargeting GEM.

## *Conclusion*

This paper describes the current GEM compiler system. However, a portable, optimizing compiler provides many opportunities that we have not yet exploited. Some enhancements planned for future versions are:

- Additional IL operators and data types, to support more languages
- Support for additional architecture and operating system combinations
- Dependence analysis, to enable some of the following enhancements
- Loop transformations, to improve the use of the memory hierarchy
- Software pipelining, to increase parallelism in vectorizable loops
- Better reordering of memory references during instruction scheduling
- The scheduling of instructions into different basic blocks
- The relaxing of the linear restriction on the lifetime model, i.e., allowing holes in register lifetimes

The GEM compiler system has met demanding technical and time-to-market goals. The system has been successfully retargeted and rehosted for the Alpha AXP and MIPS architectures and several operating environments. GEM supports a wide range of languages and provides high levels of optimization for each. The current version of GEM generates efficient code for Alpha AXP systems, and the implementation is robust and flexible enough to support future improvements.

## Acknowledgments

The authors wish to acknowledge the contributions of the following individuals to the design and implementation of the GEM compilers: Ron Brender, Patsy Griffin, Lucy Hamnett, Brian Koblenz, Dennis Murphy, Bob Peterson, Paul Winalski, Stan Whitlock (Fortran), Bevin Brett (Ada), and Farokh Morshed (C).

## References

1. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, MA: Digital Press, 1992).
2. K. Cooper, M. Hall, and L. Torczon, "The Perils of Interprocedural Knowledge," *Rice COMP TR90-132* (1990).
3. K. Cooper, M. Hall, and L. Torczon, "Unexpected Side Effects of Inline Substitution: A Case Study," *TOPLAS* (March 1992): 22–32.
4. F. Chow, "Minimizing Register Usage Penalty at Procedure Calls," *SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988): 85–94.
5. T. Lengauer and R. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *TOPLAS,* vol. 1, no. 1 (July 1979): 121–141.
6. J. Reif, "Symbolic Interpretation in Almost Linear Time," *Conference Records of the Fifth ACM Symposium on Principles of Programming Languages* (1978): 76–83.
7. J. Reif and R. Tarjan, "Symbolic Program Analysis in Almost-Linear Time," *SIAM Journal of Computing,* vol. 11, no. 1 (February 1981): 81–93.
8. K. Harris and S. Hobbs, "VAX Fortran," *Optimization in Compilers,* ed., F. Allen, B. Rosen, and F. Zadek (New York, NY: ACM Press, forthcoming).
9. R. Cattell, "Formalization and Automatic Derivation of Code Generators," Ph.D. thesis, CMU-CS-78-115, Carnegie-Mellon University, April 1978.
10. A. Aho and S. Johnson, "Optimal Code Generation for Expression Trees," *Journal of the ACM,* vol. 23, no. 3 (July 1976): 488–501.
11. B. Leverett, "Register Allocation in Optimizing Compilers," Ph.D. thesis, CMU-CS-81-103, Carnegie-Mellon University, February 1981.
12. W. Wulf et al., *The Design of an Optimizing Compiler* (New York, NY: American Elsevier Publishing Co., 1975).
13. B. Leverett et al., "An Overview of the Production-Quality Compiler-Compiler Project," *Computer,* vol. 13, no. 8 (August 1980): 38–49.
14. R. Johnsson, "An Approach to Global Register Allocation," Ph.D. thesis, Carnegie-Mellon University, December 1975.
15. R. Sethi and J. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Journal of the ACM,* vol. 17, no. 4 (October, 1970): 715–728.

## General Reference

P. Anklam et al., *Engineering a Compiler* (Bedford, MA: Digital Press, 1982).

*Richard L. Sites*
*Anton Chernoff*
*Matthew B. Kirk*
*Maurice P. Marks*
*Scott G. Robinson*

# *Binary Translation*

*Binary translation is a technique used to change an executable program for one computer architecture and operating system into an executable program for a different computer architecture and operating system. Two binary translators are among the migration tools available for Alpha AXP computers: VEST translates OpenVMS VAX binary images to OpenVMS AXP images; mx translates ULTRIX MIPS images to DEC OSF/1 AXP images. In both cases, translated code usually runs on Alpha AXP computers as fast or faster than the original code runs on the original architecture. In contrast to other migration efforts in the industry, the VAX translator reproduces subtle CISC behavior on a RISC machine, and both open-ended translators provide good performance on dynamically modified programs. Alpha AXP binary translators are important migration tools—hundreds of translated OpenVMS VAX and ULTRIX MIPS images currently run on Alpha AXP systems.*

When Digital started to design the Alpha AXP architecture in the fall of 1988, the Alpha AXP team was concerned about how to run existing VAX code and soon-to-exist MIPS code on the new Alpha AXP computers.[1,2] To take full advantage of the performance capability of a new computer architecture, an application must be ported by rebuilding, using native compilers. For a single program written in a standard programming language, this is a matter of recompile and run. A complex software application, however, can be built from hundreds of source pieces using dozens of tools. A native port of such an application is possible only when all parts of the build path are running on the new architecture.

Therefore, devising a way to run an existing (old architecture) binary version of a complex application on a new architecture is an important interim measure. Such a technique allows a user to get applications up and running immediately, with minimal porting effort. Once a user's everyday environment is established, applications can be rebuilt over time, using handwritten native code or partially native and partially old code.

## *Background*

Several techniques are used in the industry to run the binary code of an old architecture on a new architecture. Figure 1 shows four common techniques, from slowest to fastest:

- Software interpreter (e.g., Insignia Solutions' SoftPC)
- Microcoded emulator (e.g., PDP-11 compatibility mode in early VAX computers)
- Binary translator (e.g., Hunter System's XDOS)
- Native compiler

A software interpreter is a program that reads instructions of the old architecture one at a time, performing each operation in turn on a software-maintained version of the old architecture's state. Interpreters are not very fast, but they run on a wide variety of machines and can faithfully

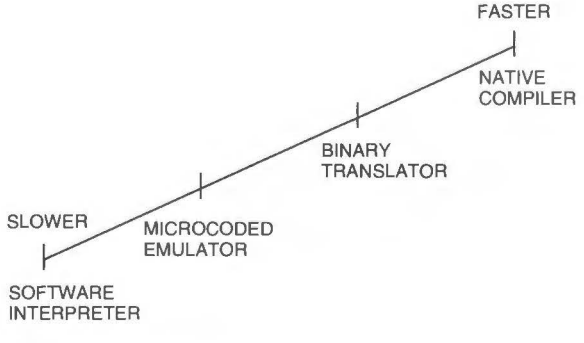



*Figure 1 Common Techniques for Running Old Code on New Computers*

reproduce the behavior of self-modifying programs, programs that branch to data, programs that branch to a checksum of themselves, etc. Caching interpreters gain speed by retaining predecoded forms of previously interpreted instructions.

A microcoded emulator operates similarly to a software interpreter but usually with some key hardware assists to decode the old instructions quickly and to hold hardware state information in registers of the micromachine. An emulator is typically faster than an interpreter but can run only on a specific microcoded new machine. This technique cannot be used to run existing code on a reduced instruction set computer (RISC) machine, since RISC architectures do not have a microcoded hardware layer underlying the visible machine architecture.

A translated binary program is a sequence of new-architecture instructions that reproduce the behavior of an old-architecture program. Typically, much of the state information of the old machine is kept in registers in the new machine. Translated code faithfully reproduces the calling standard, implicit state, instruction side effects, branching flow, and other artifacts of the old machine. Translated programs can be much faster than interpreters or emulators, but slower than native-compiled programs.

Translators can be classified as either (1) bounded translation systems, in which all the instructions of the old program must exist at translate time and must be found and translated to new instructions,[3,4,5] or (2) open-ended translation systems, in which code may also be discovered, created, or modified at execution time. Bounded systems usually require manual intervention to find 100 percent of the code; open-ended systems can be fully automatic.

To run existing VAX and MIPS programs, an open-ended system is absolutely necessary. For example, some customer programs write license-check code (VAX instructions) to memory, and branch to that code. A bounded system fails on such programs.

A native-compiled program is a sequence of new-architecture instructions produced by recompiling the program. Native-compiled programs usually use newer, faster calling conventions than old programs. With a well-tuned optimizing compiler, native-compiled programs can be substantially faster than any of the other choices.

Most large programs are not self-contained; they call library routines, windowing services, databases, and toolkits, for example. These programs also directly or indirectly invoke operating system services. In simple environments with a single dominant library, it can be sufficient to rewrite that library in native code and to interpret user programs, particularly user programs that actually spend most of their time in the library. This strategy is commonly used to run Windows and Macintosh programs under the UNIX operating system.

In more robust environments, it is not practical to rewrite all the shared libraries by hand; collections of dozens or even hundreds of images (such as typical VAX ALL-IN-1 systems) must be run in the old environment, with an occasional excursion into the native operating system. Over time, it is desirable to rebuild some images using a native compiler while retaining other images as translated code, and to achieve interoperability between these old and new images. The interface between an old environment and a new one typically consists of "jacket" routines that receive a call using old conventions and data structures, reformat the parameters, perform a native call using new conventions and data structures, reformat the result, and return.

The Alpha AXP Migration Tools team considered running old VAX binary programs on Alpha AXP computers using a simple software interpreter, but rejected this method because the performance would be too slow to be useful. We also rejected the idea of using some form of microcoded emulator. This technique would compromise the performance of a native Alpha AXP implementation, and VAX compatibility would be nearly impossible to achieve without microcode, which is inconsistent with a high-speed RISC design.

We therefore turned to open-ended binary translation. We were aware of the earlier Hewlett-Packard binary translator, but its single-image HP 3000 input code looked much simpler to translate than large collections of hand-coded VAX assembly language programs.[6] One member of the team (R. Sites) wrote a VAX-to-VAX binary translator in October 1988 as proof-of-concept. The concept looked feasible, so we set the following ambitious product goals:

1. Open-ended (completely automatic) translation of almost all user-mode applications from the OpenVMS VAX system to the OpenVMS AXP system
2. Open-ended translation of almost all user-mode applications from the ULTRIX system to the DEC OSF/1 system

3. Run-time performance of translated code on Alpha AXP computers that meets or exceeds the performance of the original code on the original architecture

4. Optional reproduction of subtle old-architecture details, at the cost of run-time performance, e.g., complex instruction set computer (CISC) instruction atomicity for multithreaded applications and exact arithmetic traps for sophisticated error handlers

5. If translation is not possible, generation of explicit messages that give reasons and specify what source changes are necessary

While we were creating the VAX translator, we discovered that the process of building flow graphs of the code and tracking data dependencies yielded information about source code bugs, performance bottlenecks, and dependencies on features not available in all Alpha AXP operating systems. This analysis information could be valuable to a source code maintainer. Thus, we added one more product goal:

6. Optional source analysis information

To achieve these goals, the Alpha AXP Migration Tools team created two binary translators: VEST, which translates OpenVMS VAX binary images to OpenVMS AXP images, and mx, which translates ULTRIX MIPS images to DEC OSF/1 AXP images. However, binary translation is only half the migration process. As shown in Figure 2, the other half is to build a run-time environment in which to execute the translated code. This second half of the process must bridge any differences between old and new operating systems, calling standards, exception handling, etc. For open-ended translation, this part of the process must also include a way to run old code that was not discovered (or did not exist) at translate time. The translated image environment (TIE) and mxr run-time environment support the VEST and mx translators, respectively, by reproducing the old operating environments. Each environment supports open-ended translation by including a fallback interpreter of old code, and extensive run-time feedback to avoid using the interpreter except for dynamically created code. Our design philosophy is to do everything feasible to stay out of the interpreter, rather than to increase the speed of the interpreter. This approach gives

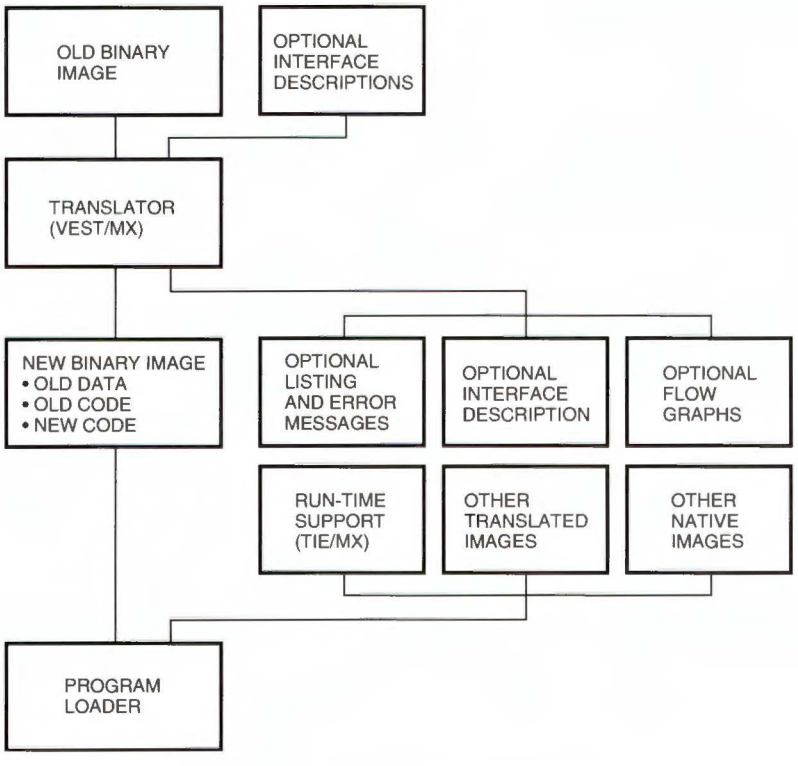



*Figure 2 Binary Translation and Execution Process*

better performance over a wider range of programs than using pure interpreters or bounded translation systems.

The remainder of this paper discusses the two binary translator/run-time environment pairs available for Alpha AXP computers: VEST/TIE and mx/mxr. To establish a basis for the discussion, the reader must understand the following terms: datum, alignment, instruction atomicity, granularity, interlocked update, and word tearing. Definitions of these terms appear in the References and Note section.[7]

## *VEST: Translating a VAX Image*

Translating a VAX image involves two main steps: analyzing VAX code and generating Alpha AXP code. The translated images produced are OpenVMS AXP images and may be run just like native images.[8] Translated images run with the assistance of the translated image environment, which is discussed later in this paper. The VEST binary translator is written in C++ and runs on VAX, MIPS, and Alpha AXP machines. The TIE is written in the OpenVMS system programming languages, BLISS and Alpha assembler.

### *Analysis*

To locate VAX code, VEST starts disassembling code at known entry points and recursively traces the program's flow of control. Entry points come from main and global routines, debug symbol table entries, and optional information files (including run-time feedback from the TIE).

As VEST traces the program, it builds a flow graph that consists of basic blocks (i.e., straight-line code sequences) annotated with information derived from parsing instructions. VEST then performs several analyses on the flow graph to propagate context information to each basic block and eliminate unnecessary operations. Context information includes condition code usage, register contents, stack depth, and a variety of other information that allows VEST to generate optimized code.

Analysis is important for achieving good performance. For example, no condition codes exist in the Alpha AXP architecture. Without analysis it would be necessary to compute condition codes for each VAX instruction even if the codes were not used. Furthermore, several forms of analysis were invented to allow correct translation. For example, VEST automatically determines if a subroutine does a normal return.

Code analysis can detect many problems, including some that indicate latent bugs in the source image. VEST can detect, for example, uninitialized variables, improperly formed VAX CASE instructions, stack depth mismatches along two different paths to the same code (the program expects data to be at a certain stack depth), improperly formed returns from subroutines, and modifications to a VAX call frame. A latent bug in the source image should be fixed, since the translated image may demonstrate incorrect behavior due to that bug.

Analysis also detects the use of unsupported OpenVMS features including unsupported system services. The source image must be modified to eliminate the use of these features.

Some problems reported by VEST result from code that is hackish in nature. For example, we found code that expects a call mask at an entry point to be executed as a no-op instruction so that the code preceding the subroutine can simply execute the call mask, rather than go through the overhead of a VAX jump (JMP) instruction. VEST reproduces the behavior of the VAX program, even if this behavior is a result of luck.

A VEST-generated flow graph is displayed in Figure 3. Dashed lines represent code paths followed if a conditional branch is taken. Solid lines indicate fall-through paths. A problem is highlighted by a wide, dashed pointer whose bottom end indicates the basic block in which the problem was uncovered. Full blocks show the path that reveals the error; empty blocks show basic blocks that are not in the error path. In Figure 3, a path exists by which register 3 (R3) may be used without being set if the VAX BNEQ (branch if the register does not equal zero) instruction in the second basic block is true the first time through the code sequence.

### *Code Generation*

The VEST translator generates code by converting each VAX instruction into zero or more Alpha AXP instructions. The architecture mapping is straightforward because there are more Alpha AXP registers than VAX registers. The VAX architecture has only 15 registers, which are used for both floating-point and integer operations. The Alpha AXP architecture has separate integer and floating-point registers. VAX R0 through R14 are mapped to Alpha AXP R0 through R14 for all operations except floating point. R12, R13, and R14 retain their VAX designations as argument pointer, frame pointer, and

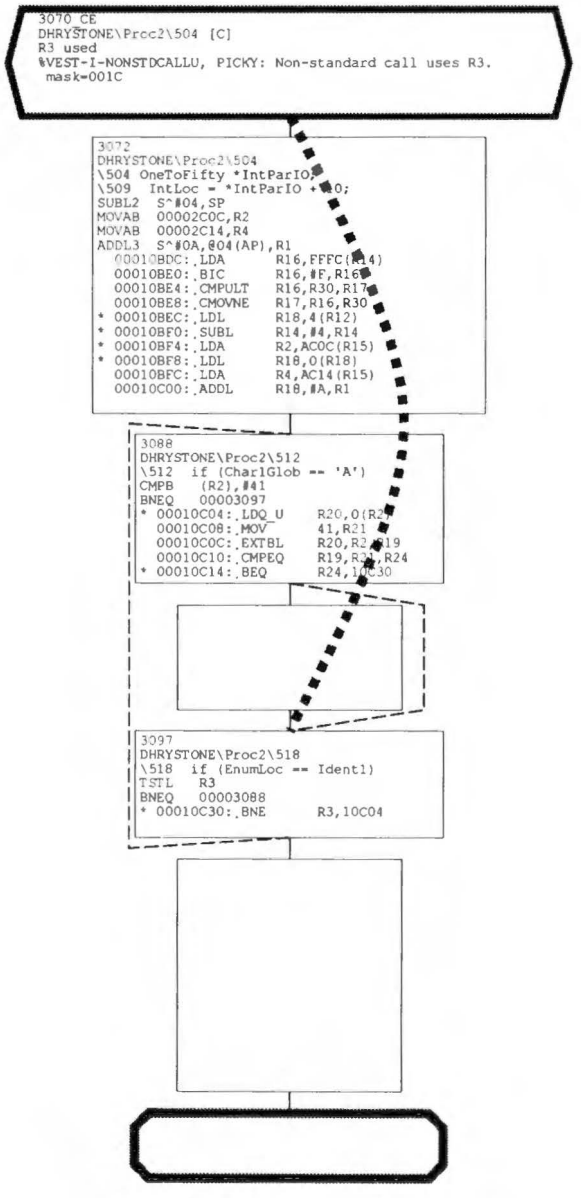



*Figure 3 VEST-generated Flow Graph Showing Uninitialized Variable*

stack pointer, and R15 is used to resolve PC-relative references. Floating-point operations are mapped to F0 through F14.

The VAX architecture has condition codes that may be referenced explicitly. In translated images, condition codes are mapped into R22 and R23. Similar to the HP 3000 translator, R23 is used as a fast condition code register for positive/negative/zero results.[6] R22 contains all four condition code bits and is calculated only when necessary. All remaining Alpha AXP registers are used as scratch registers or for OpenVMS AXP standard calls.

VEST connects simple branches directly to their translated targets. VEST performs backward symbolic execution of VAX instructions to resolve as many computed branch targets as feasible. If more than one possible computed target exists, a run-time lookup is done on the VAX target address. If the lookup fails to find a translated target, a fallback VAX interpreter is used, as described in the TIE section Failure to Find All Code during Translation. Unlike bounded translation systems, which must achieve 100 percent resolution of computed targets, the VEST and mx binary translators require no manual intervention.

## *Translated Images*

A translated image has the same format as an OpenVMS AXP image and contains the original OpenVMS VAX image as well as the Alpha AXP instructions that were generated for the VAX code. The run-time VAX interpreter TIE needs the original VAX instructions as a fallback. (Also, some error handlers look up the call stack for pointers to specific VAX instructions.) The addresses of statically allocated data in the translated image are identical to their VAX addresses. The image contains a VAX-to-Alpha AXP address mapping table for use during lookups and may contain an instruction atomicity table, described in the VAX Instruction Guarantees section.

Translated images use the OpenVMS VAX calling standard. Native images use different conventions, but translated images interoperate with native or translated shareable images. Automatic jacketing services are provided in the TIE to convert calls using one set of conventions into the other. In many cases, jacketing services permit substitution of a native shareable image for a translated shareable image without modification. However, a jacket routine is sometimes required. For example, on OpenVMS AXP systems, the translated FORTRAN run-time library, FORRTL_TV, invokes the native Alpha AXP library DEC$FORRTL for I/O-related subroutine calls. DEC$FORRTL has a different interface than FORRTL has on an OpenVMS VAX system. For these calls, FORRTL_TV contains handwritten jacket routines.

## *Files Used*

Translating an image requires only one file—a VAX executable image. Several optional files make translation more effective.

1. Image information files (IIFs). VEST automatically creates IIFs to provide information about shareable image interfaces. The information includes the addresses of entry points, names of routines, and resource utilization.
2. Symbol information files (SIFs). VEST automatically generates SIFs to control the global symbol table in a translated shared library, facilitating interoperation between translated and native images.
3. Hand-edited information files (HIFs). The TIE automatically generates HIFs, which may be hand-edited to supply information that VEST cannot deduce. HIFs contain directives to tell VEST about undetected entry points, to force it to change specific assumptions about an image during translation, and to provide known interface properties to be propagated into an IIF.

## VEST Performance Considerations

In evaluating translated code performance, we recognized that there was a significant trade-off between performance and the accuracy of emulating the VAX architecture. VEST permits users to select several architectural assumptions and optimizations, including:

- D-float precision. The Alpha AXP architecture provides hardware support for D-float with only 53-bit mantissas, whereas the VAX architecture provides 56-bit mantissas. The user may select translation with either 53-bit hardware support (faster) or 56-bit software support (slower).
- Alignment. Alpha AXP instructions support only naturally aligned longword (32-bit) and quadword (64-bit) memory operations. Unaligned memory operations cause alignment faults, which are handled transparently by software at significant run-time expense. The user may direct VEST to assume that data references are unaligned whenever alignment information is unavailable.
- Instruction atomicity. Multitasking and multiprocessing programs may depend on instruction atomicity and memory operation characteristics similar to those of the VAX architecture. VEST uses special code sequences to produce exact VAX memory characteristics. VEST and the TIE cooperate to ensure VAX instruction atomicity when instructed to do so. This mechanism is described in detail in the section Special Considerations for Instruction Atomicity.

## Untranslatable Images

Some characteristics make OpenVMS VAX images untranslatable, including:

- Exception handler issues. Images that depend on examining the VAX processor status longword (PSL) during exception handling must be modified, because the VAX PSL is not available within exception handlers.
- Direct reference to undocumented system services. Some software contains references to unsupported and undocumented system services, such as an internal-to-VMS service, which parses image symbol tables. VEST highlights these references.
- Exact VAX memory management requirements. Images that depend on exact VAX memory management behavior do not function properly and must be modified. These images include those that depend on VAX page size or that expect certain objects to be mapped to particular addresses.
- Image format. Programs that use images as data are not able to read OpenVMS AXP images without modifications, because the image formats are different.

## TIE Design Overview

The run-time translated image environment TIE assists in executing translated OpenVMS VAX images under the OpenVMS AXP operating system. Figure 4 and Table 1 show the contents of the TIE.

### Problems Solved at Run Time

Complications may occur when translated OpenVMS VAX images are run under the OpenVMS AXP operating system. This section discusses the following related topics: the failure to find all code during translation, VAX instruction guarantees, instruction atomicity, memory update, and preserving VAX exceptions.

*Failure to Find All Code during Translation*
When the VEST binary translator encounters a branch or subroutine call to an unknown destination, VEST generates code to call one of the TIE lookup routines. The lookup routines map a VAX

instruction address to a translated Alpha AXP code address. If an address mapping exists, then a transfer to the translated code is performed. Otherwise, the VAX interpreter executes the destination code. When the VAX interpreter encounters a flow of control change, it checks for returns to translated code.

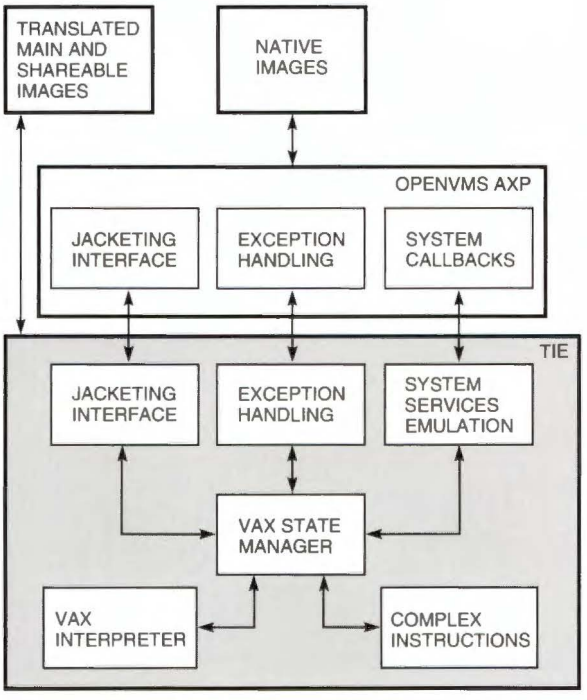



*Figure 4 VEST Run-time Environment*

If the target of the flow change is translated code, the interpreter exits to this code. Otherwise, the interpreter continues to interpret the target.

Lookup operations that transfer control to the interpreter also record the starting VAX code address in an HIF file. The VAX image can then be retranslated with the HIF information, resulting in an image that runs faster.

Lookup routines are also used to call native Alpha AXP (nontranslated) routines. The TIE supplies the required special autojacketing processing that allows interoperation between translated and native routines with no manual intervention. At load time, each translated image identifies itself to the TIE and supplies a mapping table used by the lookup routines. The TIE maintains a cache of translations to speed up the actual lookup processing.

Every translated image contains both the original VAX code and the corresponding Alpha AXP code. When a translated image identifies itself, the TIE marks its original VAX addresses with the page protection called fault on execute (FOE). An Alpha AXP processor that attempts to execute an instruction on one of these pages generates an access violation fault. This fault is processed by a TIE condition handler to convert the FOE page protection into an appropriate destination address lookup operation. For example, the FOE might occur when a translated routine returns to its caller. If the caller was interpreted, then its return address is a VAX code address instead of a translated VAX (Alpha AXP code) address. The Alpha AXP processor attempts

**Table 1 TIE Contents**

| | |
|---|---|
| VAX-to-Alpha AXP Address Mapping (VAX State Manager) | Used to find computed destinations and other cases where VEST did not find the original VAX code. Each translated image has a mapping table included. |
| VAX Instruction Atomicity Controller (VAX State Manager) | Achieves VAX instruction atomicity for asynchronous events. This allows data sharing between the single asynchronous execution context (AST) provided by OpenVMS and non-AST level routines. |
| VAX Instruction Interpreter | Executes VAX instructions not found by VEST. |
| VAX Complex Instructions | Some VAX instructions do not have code generated in-line by VEST. Those instructions are processed in the TIE. Examples are MOVC3 and MOVC5 that move byte strings. |
| OpenVMS VAX Exception Processing | Certain aspects of OpenVMS AXP exception processing are necessarily different from OpenVMS VAX. For example, the VAX computers have two scratch registers, but Alpha AXP computers have 15. Translated condition handlers are passed the VAX equivalents. |
| Routines for Differences between OpenVMS VAX and OpenVMS AXP System Services | Some operating system interfaces were rearchitected. The TIE intervenes to make the differences transparent. |

to execute the VAX code and generates a FOE condition. The TIE condition handler converts this into a JMP lookup operation.

*VAX Instruction Guarantees* Instruction guarantees are characteristics of a computer architecture that are inherent to instructions executed on that architecture. For example, on a VAX computer, if instruction 1 writes data to memory and then instruction 2 writes data to memory, a second processor must not see the write from instruction 2 before the write from instruction 1. This property is called strict read-write ordering.

The VEST/TIE pair can provide the illusion that a single CISC instruction is executed in its entirety, even though the underlying translation is a series of RISC instructions. VEST/TIE can also provide the illusion of two processors updating adjacent memory bytes without interference, even though the underlying RISC instructions manipulate four or eight bytes at a time. Finally, VEST/TIE can provide exact memory read-write ordering and arithmetic exceptions, e.g., overflow. All these provisions are optional and require extra execution time.

Tables 2 and 3 show the visibility differences between various guarantees on VAX and Alpha AXP systems as well as for translated VAX programs.

*Special Considerations for Instruction Atomicity* The VAX architecture requires that interrupted instructions complete or appear never to have started. Since translation is a process of converting one VAX instruction to potentially many Alpha AXP instructions, run-time processing must achieve this guarantee of instruction atomicity. Hence, a VAX instruction atomicity controller (IAC) was created to manipulate Alpha AXP state to an equivalent VAX state. When a translated asynchronous event

**Table 2 Single Processor Guarantees**

| Single Processor Guarantees Characterized by What an Observer Sees on the Same Processor That Executes the Data Change | | | |
|---|---|---|---|
| **Topic** | **VAX** | **Translated VAX** | **Native Alpha AXP** |
| Instruction Atomicity | An entire VAX instruction | An entire translated VAX instruction with /PRESERVE=INSTRUCTION _ATOMICITY and TIE's instruction atomicity controller, else a single Alpha AXP instruction | A single Alpha AXP instruction |

**Table 3 Multiple Processor Guarantees**

| Multiple Processor Guarantees Characterized by What an Observer on a Different Processor Sees versus the One Executing the Data Change | | | |
|---|---|---|---|
| **Topic** | **VAX** | **Translated VAX** | **Native Alpha AXP** |
| Byte Granularity | Yes, hardware ensures this | Yes, with /PRESERVE=MEMORY _ATOMICITY | Yes, via LDx_L, merge, STx_C sequence |
| Interlocked Update | Yes, for aligned datum using interlock instructions | Yes, for aligned datum using VAX interlock instructions | Yes, via LDx_L, modify, STx_C sequence |
| Word Tearing | Aligned longword writes change all bytes at once<br><br>Other writes are allowed to change one byte at a time | Aligned longword or quadword writes change all bytes at once | Aligned longword or quadword writes change all bytes at once |

processing routine is called, the IAC is invoked. The IAC examines the Alpha AXP instruction stream and either backs up the interrupted program counter to restart at the equivalent VAX instruction boundary or executes the remaining instructions to the next boundary. Many VAX programs do not require this guarantee to operate correctly, so VEST emits code that is VAX instruction atomic only if the qualifier /PRESERVE=INSTRUCTION_ATOMICITY is specified when translating an image.

VEST-generated code consists of four sections that are detected by the IAC. These sections have the following functions:

- Get operands to temporary registers
- Operate on these temporary registers
- Atomically update VAX results that could generate side effects (i.e., an exception or interlocked access)
- Perform any updates that cannot generate side effects (e.g., register updates)

The VAX interpreter achieves VAX instruction atomicity by using the atomic move, register to memory (AMOVRM) instruction. The AMOVRM instruction is implemented in privileged architecture library (PAL) subroutines and updates a contiguous region of memory containing VAX state without being interrupted. At the beginning of each interpreted VAX instruction, a read and set flag (RS) instruction sets a flag that is cleared when an interrupt occurs on the processor. AMOVRM tests the flag, and if set, performs the update and returns a success indication. If the flag is clear, the AMOVRM instruction indicates failure, and the interpreter reprocesses the interrupted instruction.

*Issues with Changing Memory* VAX instruction atomicity ensures that an arithmetic instruction does not have any partially updated memory locations, as viewed from the processor on which that instruction is executed. In a multiprocessing environment, inspection from another processor could result in a perception of partial results.

Since an Alpha AXP processor accesses memory only in aligned longwords or quadwords, it is therefore not byte granular. To achieve byte granularity, VEST generates a load-locked/store-conditional code sequence, which ensures that a memory location is updated as if it were byte granular. This sequence is also used to ensure interlocked access to shared memory. Longword-size updates to aligned locations are performed using normal load/store instructions to ensure longword granularity.

Many multiprocessing VAX programs depend on byte granularity for memory update. VEST generates byte-granular code if the condition /PRESERVE=MEMORY_ATOMICITY is specified when translating an image. In addition, VEST generates strict read-write ordering code if the qualifier /PRESERVE=READ_WRITE_ORDERING is specified when translating an image.

*Preserving VAX Exceptions* Alpha AXP instructions do not have the same exception characteristics as VAX instructions. For instance, an arithmetic fault is imprecise, i.e., not synchronous with the instruction that caused it. The Alpha AXP hardware generates an arithmetic fault that gets mapped into an OpenVMS AXP high-performance arithmetic (HPARITH) exception. To retain compatibility with VAX condition handlers, the TIE maps HPARITH into a corresponding VAX exception when calling a translated condition handler. Most VAX languages do not require precise exceptions. For those that do, like BASIC, VEST generates the necessary trap barrier (TRAPB) instructions if /PRESERVE=FLOATING_EXCEPTIONS is specified when translating an image.

## *OpenVMS AXP and OpenVMS VAX Differences*

*Functional Differences* Most OpenVMS AXP system services are identical to their OpenVMS VAX counterparts. Services that depend on a VAX-specific mechanism are changed for the Alpha AXP architecture. The TIE intervenes in such system services to ensure the translated code sees the old interface.

For example, the declare change mode handler ($DCLCMH) system service establishes a handler for VAX change mode to user (CHMU) instructions. The handler is invoked as if it were an interrupt service routine required to use the VAX return from interrupt or exception (REI) instruction to return to the invoker's context. On OpenVMS AXP systems, the handler is called as a normal procedure. To ensure compatibility, the TIE inserts its own handler when calling OpenVMS AXP $DCLCMH. When a CHMU is invoked on Alpha AXP computers, the TIE handler calls the handler of the translated image, using the same VAX-specific mechanisms that the handler expects.

*Exception Handling* OpenVMS AXP exception processing is almost identical to that performed in the OpenVMS VAX system. The major difference is that the VAX mechanism array needs to hold the value of only two temporary registers, R0 and R1, whereas the Alpha AXP mechanism array needs to hold the value of 15 temporary registers, R0, R1, and R16 through R28.

*Complex Instructions* Translating some VAX instructions would require many Alpha AXP instructions. Instead, VEST generates code that calls a TIE subroutine. Subroutines are implemented in two ways: (1) handwritten native emulation routines, e.g., MOVC5, and (2) VEST-translated VAX emulation routines, e.g., POLYH.

Together, VEST and TIE can translate and run most existing user-mode VAX binary images. As shown in Table 4, performance of translated VAX programs slightly exceeds the original goal. Performance depends heavily on the frequency of use of VAX features that are not present in Alpha AXP machines.

## ULTRIX MIPS Translation

mx is the translator that converts ULTRIX MIPS programs to DEC OSF/1 AXP programs. The mx project started after VEST was functional, and we took advantage of the VEST common code base for much of the analysis and Alpha AXP code assembly phases of the translator. In fact, about half of the code in mx is compiled from the same source files as those used for VEST, with some architectural specifics supplied by differing include files. The code-sharing aspects of C++ have proven quite valuable in this regard.

mxr is the run-time support system for translated programs. It provides services similar to TIE, emulating the ULTRIX MIPS environment on a DEC OSF/1 AXP system. mxr is written in C++, C, and Alpha assembler.

## Challenges

Creating a translator for the MIPS R2000/R3000 architecture presented us with a host of new opportunities, along with some significant challenges. The basic structure of the mx translator is much simpler than that of VEST. Both the source and the target architectures are RISC machines; therefore, the two instruction sets have a considerable similarity. Many instructions translate one for one. The MIPS architecture has very few instruction side effects or subtle architectural details, although

**Table 4 Translated VAX Performance, Normalized to Native-compiled OpenVMS AXP Code**

| Program | VAX Time on VAX 6610 (83.3 MHz) | VEST Translated Time on DEC 7000 AXP (167 MHz)* | Native Time on DEC 7000 AXP (167 MHz) |
|---|---|---|---|
| SPECmark89 | | | |
| gcc | 1.9 | –† | – |
| expresso | 3.1 | 2.7 | 1.0 |
| spice2g6 | 2.8 | 1.8 | 1.0 |
| doduc | 2.9 | 3.0 | 1.0 |
| nasa7 | 4.4 | 6.2 | 1.0 |
| li | 2.7 | 4.2 | 1.0 |
| eqntott | 3.3 | 2.2 | 1.0 |
| matrix300 | 8.8 | 4.2 | 1.0 |
| fpppp | 3.8 | 2.7 | 1.0 |
| tomcatv | 5.3 | 2.9 | 1.0 |
| Geometric Mean (without gcc) | 3.8 | 3.1 | 1.0 |

Notes:
The larger the number, the slower the performance. These performance numbers were measured on derated field test hardware and software at various times during 1992; production results will vary somewhat. The SPEC benchmarks are written in FORTRAN and C; no conclusions should be drawn about other classes of programs written in other languages.

*The DEC 7000 system was running at a derated speed compared to production DEC 7000 systems.

†Timing information for this run is not available.

those that are present are particularly tricky. Furthermore, the format of an executable program under the ULTRIX system collects all code in a single contiguous segment and makes it easy for mx to reliably find close to 100 percent of the code in the MIPS application. The system interfaces to the ULTRIX and DEC OSF/1 systems are similar enough that most ULTRIX system calls have functionally identical counterparts under the DEC OSF/1 system.

The challenges in mx stem from the fact that the source architecture is a RISC machine. For example, DEC OSF/1 AXP is a 64-bit computing environment, i.e., all pointers used to communicate with the operating system are 64 bits wide. This environment does not present a problem when the pointer is passed in a register. However, when a pointer (or a long data item, such as a file size) is passed in memory, it must be converted between the 32-bit representation, used by the ULTRIX system, and the 64-bit AXP representation, even when the semantics of the operating system call are the same on both systems.

A significant challenge is the fact that our users' expectations for performance of translated programs are much higher than for VEST. Reasoning that the source and target machines are similar, users also expect mx to achieve a translated program performance better than that of the source program, since Alpha AXP processors are faster. Thus, as our performance goal, we set out to produce a translated program that runs at about the same speed as the original program would run on a MIPS R4000 machine with a 100-megahertz (MHz) internal clock rate.

## *Mapping the Architectures*

At first glance, it appears that we could simply assign each MIPS register to a corresponding Alpha AXP register, because each machine has 32 general-purpose registers. The translated code would then have two scratch registers, since the MIPS architecture does not allow user-level programs to use registers K0 and K1, which are reserved for the operating system kernel.

Unfortunately, translation requires more than two scratch registers. The Alpha AXP architecture does not have byte or halfword (16-bit) loads or stores, and the code sequences for performing these operations require four or five scratch registers. Furthermore, mx requires a base register to locate mxr without having to load a 64-bit address constant at each call. Finally, the MIPS architecture has more than 32 registers, including the HI and LO registers used by the multiply and divide instructions, and a floating-point condition register, whose layout and contents do not correspond to the Alpha AXP floating-point condition register.

In mx, we assign registers using standard compiler techniques. To assign registers to 33 MIPS resources (the 32 general registers plus one 64-bit register to hold both HI and LO), certain registers are permanently mapped, and other MIPS registers are kept in either AXP registers or memory. The MIPS argument-passing registers A0 through A3 are permanently assigned to Alpha AXP registers R16 through R19, which are the argument registers in the DEC OSF/1 AXP calling standard. This correspondence simplifies the work needed when mxr must take arguments for an ULTRIX system call and pass them to a DEC OSF/1 system call. Similarly, the argument return registers V0 and V1 are mapped to the Alpha AXP argument return registers R0 and R1. The return address registers and stack pointer registers of the two machines are also mapped. MIPS R0 is mapped to Alpha AXP R31, where both registers contain the same hard-wired zero value. We reserve Alpha AXP registers R22 through R24 as scratch registers and also use them when interfacing to mxr. We reserve Alpha AXP R14 as a pointer to an mxr communication area. Finally, we reserve three more registers as scratch registers for use by the code generator.

The remaining 16 Alpha AXP registers are available to be assigned to the remaining 23 MIPS resources. After the code is analyzed and we have register usage information, the 16 most frequently used MIPS registers get mapped to the remaining 16 Alpha AXP registers, and the remaining registers are assigned to memory slots in the mxr communication area. When a MIPS basic block uses one of the slotted registers, mx assigns it to one of the scratch registers. If the first reference reads the old contents of the register, mx generates a load instruction from the communications area. If the value of the MIPS resource changes in the basic block, the scratch register is stored in the communication area before the end of the block. As in most compilers, if we run out of registers, a spill algorithm chooses a value to save in the communication area and frees up a register.

Alpha AXP integer registers are 64 bits wide, whereas MIPS registers are only 32 bits wide. We chose to keep all 32-bit values in Alpha AXP integer

registers as sign-extended values, with the high 32 bits equal to bit 31. This approach occasionally requires mx to generate additional code to create canonical 32-bit integer results, but the 64-bit compare operations do not need to change the values that they are comparing.

The floating-point architecture is more complex. Each of the 32 MIPS floating-point registers is 32 bits wide. Only the even registers are used for single precision, and a double-precision number is kept in an even-odd register pair. We map each pair of MIPS floating-point registers onto a single 64-bit Alpha AXP floating-point register. Also, one Alpha AXP floating-point register represents the condition code bit of the MIPS floating-point control register. Thus, the mx code generator can use 14 scratch registers. mx goes to considerable effort to find paired loads and stores in the MIPS code stream, and to merge them into one Alpha AXP floating-point operation.

MIPS single-precision operations cause problems with floating-point correspondence. Since on MIPS machines, the single-precision number is kept in only the even register of the register pair, the even and odd registers in a pair are independent when single-precision (or integer) operations are done in the floating-point unit. On Alpha AXP machines, computation must be done on a value extended to double format in the whole 64-bit register. We defined two forms for values in Alpha AXP floating-point registers: computational form, in which computation is done, and canonical form, which mimics the MIPS even and odd registers. If a MIPS program loads an even register and uses this register as a single-precision value, mx loads the value from memory to be used computationally. If a MIPS program loads only an even register but does not use this register in the basic block, mx puts the 32-bit value into half of the Alpha AXP floating-point register. This permits correct behavior in the pathological case where half of a floating-point number is loaded in one place, and the other half is loaded in some other basic block. If a register is used as a single-precision number in a basic block without first being loaded, the code generator inserts code to convert it from canonical to computational floating-point form. If a single-precision value has been computed in a block and is live at the end of the block, it is converted to canonical form.

mx inserts a register mapping table into the translated program that indicates which MIPS resources are statically mapped to which Alpha AXP registers, and which MIPS resources are normally kept in memory. This table allows mxr to find the MIPS resources at run time.

### *Finding Code*

As with the VEST translator, mx finds code by starting at entry points and recursively tracing down the flow of control. mx finds entry points using the executable file header, the symbol table (if present), and feedback from mxr (if present). Finally, mx performs a linear scan of the entire text section for unexamined words. mx analyzes any data that looks like plausible code but does not connect this data into the main flow graph. Plausible code consists of a series of valid MIPS instructions terminated by an unconditional transfer of control.

While finding code and connecting the basic blocks into a flow graph, mx looks for the code sequence that indicates a switch statement, i.e., a multi-way branch, usually through an element of a table. mx finds the branch table and connects each of the possible targets as successors of the branch.

### *Code Analysis*

Our static analysis of hundreds of MIPS programs indicates that only 10 instructions account for about 85 percent of all code. These instructions are LW, ADDIU, SW, NOP, ADDU, BEQ, JAL, BNE, LUI, and SLL. The corresponding sequences of Alpha AXP code range from zero operation codes, or opcodes, (for NOP, since the Alpha AXP architecture does not require NOPs anywhere in the code stream) to two opcodes (for SLL).

Code analysis for source programs is much more important in mx than in VEST, because the coding idioms for many common operations differ between the Alpha AXP and MIPS processors. The simple technique of mapping each MIPS instruction to a sequence of one or more Alpha AXP instructions loses much of the context information in the original program.

For example, the idiom used to load a 32-bit constant into a register on MIPS machines is to generate a load upper immediate (LUI) opcode, placing a 16-bit constant in the high-order 16 bits of a register. This operation is followed by an OR immediate (ORI) opcode, logically ORing a 16-bit zero-extended value into the register. The LUI corresponds exactly to the Alpha AXP load address high (LDAH) opcode. However, the Alpha AXP

architecture has no way of directly ORing a 16-bit value into a register and cannot even load a zero-extended 16-bit constant into a register. When the high-order bit of the 16-bit constant is 1, the shortest translation for the ORI is three instructions. The mx translator scans the code looking for such idioms, and generates the optimal two-instruction sequence of Alpha AXP code that performs the 32-bit load. No opcode exists that corresponds to the ORI, but the results in the registers are correct.

When we started writing the mx translator, we listed a number of code possibilities that we thought we would never see. In retrospect, this was a misguided assumption. For example, we have seen programs that branch into the delay slot of other instructions, requiring us to indicate that the delay slot instruction is a member of two different basic blocks—the block it ends, and the one it starts. We have observed programs that put software breakpoint (BREAK) instructions in the branch delay slot, and thus BREAK ends a basic block without being the last instruction. Some compilers schedule code so that half of a floating-point register is stored and then reused before the other half is stored. The general principle that we intuit from these observations is "if a code sequence is not expressly prohibited by the architecture, some program somewhere will use it."

## *Code Generation*

After the program is parsed and analyzed and the flow graph is built, the code generator is called. It builds the register mapping table and then, in turn, processes each basic block, generating Alpha AXP code that performs the same functions as the MIPS code.

At each subroutine entry, mx scans the code stream with a pattern-matching algorithm to see if the code corresponds to any of a number of standard MIPS library routines, such as strcpy. (Note that the ULTRIX operating system has no shared libraries, so library routines are bound into each binary image.) If a correspondence exists, the entire subroutine is recursively deleted from the flow graph and replaced with a canned routine to perform the subroutine's work on Alpha AXP processors. This technique contributes significantly to the performance of translated programs.

For each remaining basic block, the instructions are converted to a linked list of intermediate opcodes. At first, each opcode corresponds exactly to a MIPS opcode. The list is then scanned by an optimization phase, which looks for MIPS coding idioms and replaces them with abstract machine instructions that better reflect the idiom. For example, mx changes loads of immediate values to a non-MIPS hardware load immediate (LI) instruction; shift and add sequences to abstract operations that reflect the Alpha AXP scaled add and subtract sequences; and sequences that change the floating-point rounding mode (used to truncate a floating-point number to an integer) to a single opcode that represents the Alpha AXP convert operation with the chopped mode (/C) modifier.

MIPS code contains a number of common code sequences that cross basic block boundaries, but which can be compressed into a single basic block in Alpha AXP code. Examples of these are the min and max functions, which map neatly onto a single conditional move (CMOVxx) instruction in Alpha AXP code. The code generator looks for these sequences, merges the basic blocks, and creates an extended basic block, which includes pseudo-opcodes that indicate the MIPS code idiom.

After the optimizer completes the list of instructions, it translates each abstract opcode to zero or more Alpha AXP opcodes, again building a linked list of instructions. This process may permit further improvements, so the optimizer makes a second pass over the Alpha AXP code.

When processing a basic block, the code generator assumes that it has an unlimited number of temporary resources. Since this is not actually true, the code generator then calls a register assigner to allocate the real Alpha AXP temporary resources to the intermediate temporary registers. The register assigner will load and spill MIPS resources and generated temporary registers as needed.

Finally, the list of Alpha AXP instructions is assembled into a binary stream, and the instruction scheduler rearranges them to remove resource latencies and use the chip's multiple issue capability.

## *Image Formats*

The file format for input is the standard ULTRIX extended common object file format (COFF). In most ULTRIX MIPS programs, the text section starts at 00400000 (hexadecimal) and the data at 10000000 (hexadecimal). In virtually all programs, a large gap exists between the virtual address for the end of text and the start of the data section. When mx creates the output image, it places the generated Alpha AXP code after the MIPS code and

before the MIPS data. This allows the program to have one large text section. The Alpha AXP code begins at an Alpha AXP page boundary, so that we can set the memory protection on the MIPS code separately from the Alpha AXP code.

The translated image is not in DEC OSF/1 AXP executable format. Instead, it looks like a MIPS COFF file, but with the first few bytes changed to the string "#!/usr/bin/mxr".

### *Executing a Translated Program*

When a translated image is run on DEC OSF/1 AXP, its modified header invokes mxr first. mxr uses the memory map (mmap) system call to load the translated program at the same virtual address that it would have had under the ULTRIX operating system. mxr resets the protection of the MIPS code to read/no-write/no-execute, the Alpha AXP code to read/no-write/execute, and the data to read/write/no-execute.

mxr allocates a communication area and initializes Alpha AXP R14 to point to this area. The communication area contains save areas for MIPS resources, initialized pointers to mxr service routines, and other scratch space. mxr then constructs new command argument (argv) and environment vectors as 32-bit wide pointers (as the MIPS program expects), arranges to intercept certain signals from the DEC OSF/1 AXP system, and transfers control to the translated start address of the program.

When a system signal is delivered to the program, control goes to the signal intercept code in mxr. This code transforms the signal context structure from the DEC OSF/1 AXP system and constructs an ULTRIX MIPS style context, which it then passes to the translated signal handler.

Certain signals are processed specially. For instance, a program that attempts to transfer control to a location containing MIPS code rather than translated code gets a segmentation violation, since the MIPS code is not executable. This situation can occur if a routine modifies its return address to be a MIPS address constant. mxr will examine the target address and, if it corresponds to the start of a pretranslated MIPS basic block, divert the flow of control to the translated code for that block. If not, mxr enters the MIPS interpreter. The interpreter proceeds to emulate the MIPS code until a translated point is reached. mxr then resynchronizes its machine state and reenters the translated code.

### *Translation Goals and Classes of Programs Not Supported*

Our goal was to translate most user-mode MIPS programs compiled for a MIPS R2000 or R3000 machine running ULTRIX Release 4.0 (or later) to run identically on the DEC OSF/1 AXP system with acceptable performance. As shown in Table 5, performance of translated MIPS programs meets or exceeds the original goal.

**Table 5 Translated MIPS Relative Performance**

| Program | MIPS Time on DECstation 5000 Model 240 (40 MHz) | Translated Time on DEC 3000 AXP Model 500 (150 MHz) |
|---|---|---|
| SPECint92 | | |
| espresso | 2.4 | 1.1 (1.0)* |
| li | 1.6 | 1.2 (1.0) |
| eqntott | 1.6 | 2.1 (1.0) |
| compress | 2.7 | 1.0 (1.0) |
| sc | –† | – |
| gcc | 2.1 | 1.2 (1.0) |
| Geometric Mean (without sc) | 2.0 | 1.3 (1.0) |
| SPECfp92 | | |
| spice2g6 | – | – |
| doduc | 1.7 | 1.0 |
| mdljdp2 | 2.7 | 1.0 |
| wave5 | 1.1 | 1.0 |
| tomcatv | 3.0 | 1.0 |
| ora | 1.5 | 1.0 |
| alvinn | 1.6 | 1.0 |
| ear | 1.7 | 1.0 |
| mdljsp2 | 1.4 | 1.0 |
| swm256 | 2.3 | 1.0 |
| su2cor | 2.7 | 1.0 |
| hydro2d | 2.9 | 1.0 |
| nasa7 | 2.6 | 1.0 |
| fpppp | 2.2 | 1.0 |
| Geometric Mean (without spice2g6) | 2.0 | 1.0 |

Notes:
The larger the number, the slower the performance. These performance numbers were measured on derated field test hardware and software at various times during 1992; production results will vary somewhat. The SPEC benchmarks are written in FORTRAN and C; no conclusions should be drawn about other classes of programs written in other languages.

*The values in parentheses are from running once, then retranslating with the run-time feedback from the first run; this gave a significant performance difference only for the programs shown.

†Timing information for this run is not available.

Due to extreme technical obstacles, some classes of programs will never be supported by mx. We decided not to translate programs that use privileged opcodes or system calls or that need to run with superuser privileges. In cases where the file system hierarchy differs between the ULTRIX and DEC OSF/1 AXP systems, programs that expect files to be in particular places or in a particular format may fail. Similarly, programs that read /dev/kmem and expect to see an ULTRIX MIPS memory layout fail.

Certain other classes of programs are not currently supported, but are technically feasible. These include big endian MIPS programs from non-Digital MIPS environments, programs that use R4000 or R6000 instructions that are not present on the R3000 model, programs that need to be multiprocessor safe, and programs that require certain categories of precise exception behavior.

## Summary

Building successful turnkey binary translators requires hard work but not magic. We built two different translators, VEST and mx. In both cases, the old and new environments are, by design, quite similar in fundamental data types, memory addressing, register and stack usage, and operating system services. Translators between dissimilar architectures or operating systems are a different matter. Translating the code might be a reasonably straightforward task. However, emulating a run-time environment in which to execute the code might present insurmountable technical and business obstacles. Without capturing the environment, an instruction translator would be of no use.

The idea of binary translation is becoming more common in the computer industry, as various other companies start on their transitions to 64-bit architectures.

## Acknowledgments

Steve Hobbs originally suggested the binary translation path in the architecture task force discussions. Nancy Kronenberg and Bob Supnik added critical early support and later coordination. Jud Leonard set the engineering direction of doing careful static translation once, instead of on-the-fly dynamic translation at each execution. Butler Lampson boosted morale at a critical time. Jim Gettys has also been an important and vocal supporter.

The success of the translators would not have been possible without the enthusiastic support of the OpenVMS AXP and DEC OSF/1 AXP operating system groups, and the respective run-time library groups, especially Matt LaPine, Larry Woodman, Hai Huang, Dan Murphy, Nitin Karkhanis, Ray Lanza, Anton Verhulst, and Terry Grieb.

The Porting and Performance Engineering Group did extensive porting and testing of customer applications. The group members, especially Shamin Bhindarwala and Robi Al-Jaar, were sources of extremely valuable customer feedback. The Engineering System Group under Mike Greenfield also made extensive early use of the translators and provided valuable feedback.

The Alpha AXP Migration Tools team is relatively small for the substantial amount of work accomplished in the past two and one-half years. Every person has made several key contributions. In addition to the authors of this paper, the team members are: Kate Burleson, Peigi Cleminshaw, George Darcy, Catherine Frean, Bruce Gordon, Rick Gorton, Kevin Koch, Mark Herdeg, Giovanni Della Libera, Nikki Mirghafori, Srinivasan Murari, Jim Paradis, and Ashutosh Roy.

## References and Note

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

2. R. Sites, "Alpha AXP Architecture," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 19–34.

3. C. Hunter and J. Banning, "DOS at RISC," *Byte Magazine* (November 1989): 361–368.

4. *Echo Logic,* News Release (May 4, 1992).

5. L. Wirbel, "DOS-to-UNIX Compiler," *Electronic Engineering Times* (March 14, 1988): 83.

6. A. Bergh, K. Keilman, D. Magenheimer, and J. Miller, "HP 3000 Emulation on HP Precision Architecture Computers," *Hewlett-Packard Journal* (December 1987).

7. Datum is the term used to refer to a piece of information that has an address and a size.

   Alignment is the property of a datum of size $2^n$ bytes. This datum is aligned if its byte address has $n$ low-order zeros. A size or address not meeting this constraint implies that the datum is unaligned.

   Instruction atomicity is the property of instruction execution on single processor systems such that an interrupted instruction has been

completed or has never started, i.e., partial execution of an instruction is never observed.

Granularity is the property of memory writes on multiprocessor systems such that independent writes to adjacent aligned data produce consistent results. The terms byte, word, longword, quadword, and octaword granularity refer to writing 1-, 2-, 4-, 8-, and 16-byte size adjacent data.

Interlocked update is the property of memory updates (read-modify-write sequences) on multiprocessor systems such that simultaneous independent updates to the same aligned datum will be consistent. This property causes serialization of the independent read-modify-write sequences and is not guaranteed for an unaligned datum.

Word tearing is the property of aligned memory writes on multiprocessor systems such that a reader independent of the writer can see partial results of the write.

8. N. Kronenberg et al., "Porting OpenVMS from VAX to Alpha AXP," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 111–120.

*Jeffrey A. Coffler*
*Zia Mohamed*
*Peter M. Spiro*

# *Porting Digital's Database Management Products to the Alpha AXP Platform*

*The cornerstone software component of high-end production systems is a database management system. Digital has successfully ported the DEC Rdb for OpenVMS relational database management system and the DEC DBMS for OpenVMS network database management system to the Alpha AXP platform. Rdb and DBMS were perhaps the most complex layered products to be ported. The tight coupling of these two products to the OpenVMS VAX system made the port a challenging task. To avoid the future problem of integrating two source code bases, the porting team decided to use a common code base and to overlap current VAX development with the Alpha AXP port. The goal was to provide an easy migration path for software products to the Alpha AXP platform.*

Digital is one of a small number of vendors competing in the high-end, complex production systems market. Applications for this market support industries such as banking, stock exchanges, telecommunications, and information services. The Alpha AXP platform is ideally suited to meet the response time, throughput, and availability requirements of these applications, since it offers increased performance while maintaining the superb availability characteristics of VMScluster systems.

Although high-end production systems involve a collection of software packages, the cornerstone software component is a database management system. Digital offers two database management systems for high-end commercial systems: DEC Rdb for OpenVMS, a relational database management system, and DEC DBMS for OpenVMS, a network (CODASYL) database management system. Digital had to port the DEC Rdb for OpenVMS VAX and DEC DBMS for OpenVMS VAX database systems to the Alpha AXP platform as early as possible to continue to compete in this commercial arena. The resulting products are the DEC Rdb for OpenVMS AXP and DEC DBMS for OpenVMS AXP systems. (Since these two products for the Alpha AXP system are the same as those for the VAX system, hereafter, we will refer to the products as Rdb and DBMS.) Additionally, both software products drive many sales of Digital's OpenVMS operating system and transaction processing and information management products such as CDD, ACMS, and DEC RALLY, which integrate with the Rdb and DBMS systems.

Database management systems are among the most complex of all software products. Applications expect these systems to have 7 by 24 availability, sophisticated concurrency capabilities, fast data access, high-speed backup and restore mechanisms, and large buffer pools. To provide such functionality, the Rdb and DBMS products make extensive use of the OpenVMS VAX system, the VAX run-time libraries, and the BLISS and VAX MACRO-32 programming languages. The current release of the product set uses more than 100 system services or run-time library calls. The two products utilize almost every BLISS BUILTIN function, i.e., a machine-specific function call that generates in-line code. Combined, Rdb and DBMS comprise more than 30 different images. The products run in elevated processing modes, both executive and kernel, and include user-written system services.

Further compounding the complexity of porting the Rdb and DBMS software to the Alpha AXP platform is the fact that they are mature products; DBMS was released in 1981, Rdb in 1984. Because various system capabilities did not exist in the early 1980s, the two database management systems include code that is no longer required. For example, both products have code to move bytes from one data

type to another. Also, during image rundown, the products rely on undocumented, operating system behavioral patterns such as the asynchronous system trap (AST) delivery protocols. In addition, the Rdb software contains a modified version of the OpenVMS SORT routine.

Rdb and DBMS were initially designed to run only on the OpenVMS VAX operating system. Consequently, both products heavily utilize VAX-specific features for performance gains.[1] For example, Rdb generates VAX machine code routines as part of query execution plans; the machine code is carefully generated for maximum execution efficiency. This tight coupling of Rdb and DBMS to the OpenVMS VAX system made the port a challenging task.

Since the OpenVMS and BLISS groups were busy with their own porting projects, we in the Database Systems Group had to accomplish our port with little outside help. The task was noteworthy because, by necessity, the team had to port its product set to the Alpha AXP platform earlier than most of the other porting groups. At the same time, Rdb and DBMS were perhaps the most complex layered products that would be ported. Our goal was to port these two products in a timely fashion, so that Digital would truly succeed in providing an easy migration path for software products to the Alpha AXP platform.

In this paper, we first present a brief description of the architecture of the two database management system products. We next describe the guiding policies we formulated to allow the port to proceed as efficiently as possible. Then, we document porting issues that we resolved for the two products. Finally, we summarize our experiences related to this effort.

## Product Architecture

Digital is unique in the database industry in that we provide two different types of database management systems that layer on top of the same database kernel, which is called KODA. The KODA kernel provides journaling and recovery, locking, access methods (e.g., B-tree, hashing), record and page management, and buffer pool management.

The Rdb software provides language preprocessors, an interactive query front end, a callable interface, catalogue management, query optimization, and relational operations such as join, select, and project. Rdb supplies a relational interface to the database.

The DBMS product also provides language preprocessors, an interactive query front end, and other software necessary to define, create, and manage data in simple or complex databases. In contrast to Rdb, DBMS provides a CODASYL interface to the database.

Figure 1 shows the relationship of the Rdb and DBMS software products to the KODA database kernel.

## Porting Policies

Initially, we developed policies to guide our port to the Alpha AXP platform. These policies, which applied to the KODA, Rdb, and DBMS teams, were designed to simplify the port and to ease long-term maintenance requirements.

### Common Source Code Base

Our most important decision was to have a common source code base. That is, we wanted to have one set of source code that could be compiled and run on either a VAX or an Alpha AXP system. At the time we began our port, the OpenVMS group was the only other software group that had started their port, and they had chosen to have two distinct code bases. (The OpenVMS AXP porting schedule dictated the choice.) So with respect to code base, the path we chose was untested. We also decided to maintain common command procedures to compile, build, and link, and common regression tests between the VAX and Alpha AXP systems.

A primary reason for our code base decision was that we did not have the resources to manage two different code bases. Also, although two divergent code sources would have allowed for a stable code

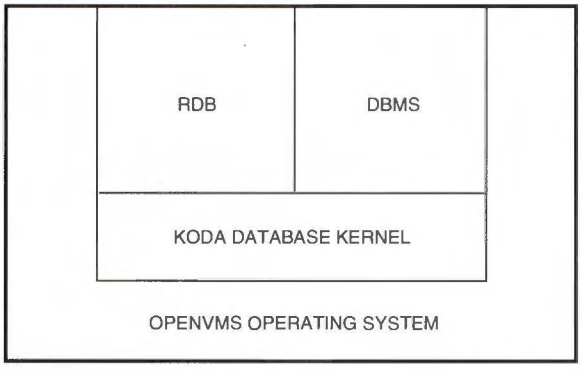



*Figure 1 Relationship of Rdb and DBMS to the KODA Database Kernel*

base with which to begin the Alpha AXP port, the group strongly wanted to avoid having to merge the two code bases at a future date. Consequently, since our preliminary investigation indicated that a single code base was feasible and that we could hide most of the platform dependencies through the superb macro capability of the BLISS language, we proceeded with the common source code implementation. The single code base allowed us to build and release Alpha AXP and VAX versions of our products at the same time.

### *Concurrent Releases*

Our release schedule complicated the process of adhering to the single code base policy. To meet the schedule, we had to overlap some of the Alpha AXP port with our current VAX releases. That is, the scenario we followed was NOT: work on a VAX release; complete all necessary code changes; stabilize the release; and then create a newer set of sources for the Alpha AXP port. Rather, for the beginning portion of the Alpha AXP port, we also had to change source code destined for a VAX release. Thus, if a module had to be changed for the earlier VAX release and the same module had already been ported for the Alpha AXP release, the engineer had to propagate the code change to the Alpha AXP source code.

To minimize the effect of double code changes, we first worked on those modules for the Alpha AXP release that were reasonably stable in the current VAX code stream. For example, the BLISS REQUIRE files that we use for data definitions were reasonably stable for the VAX release by the time the Alpha AXP port began. The modules that did not change for the VAX release were also good candidates for helping us to avoid making double code changes. When we finally began to port the bulk of the modules, they were mostly stable and, as a result, only bug fixes for the VAX release required that we manually modify the same module for the Alpha AXP release.

Furthermore, once we began work on the Alpha AXP release, we needed the capability of being able to compile, link, and test on both the Alpha AXP and VAX platforms. So we had to modify our development environment to allow us to identify the code change session as either an Alpha AXP or a VAX session.

### *No New Functionality*

The Alpha AXP release of the database management system product set contains no new functionality. On the first pass, we decided to port the VAX code without designing any new algorithms. We did clean up some code for style, convention, and performance, but basically, the Alpha AXP release remains functionally equivalent to the latest VAX release.

### *Correct and Fast Code Execution*

We did not prioritize our effort to first, be correct, and second, be fast. We decided that we must be correct *and* fast on certain key issues. For example, on VAX systems, our argument-passing mechanism utilized the argument pointer (AP). To minimize code changes, we could have used the ARGPTR construct in the BLISS cross compiler. However, ARGPTR is inefficient and, therefore, not appropriate for our needs. Consequently, we ensured that our new argument-passing design was efficient, even though doing so was time-consuming.

### *Minimizing Platform-specific Modules*

Code conditionalization, i.e., producing separate code for the VAX and the Alpha AXP platforms, requires various levels of code duplication. For example, the process may require the duplication of an entire module, routines within a module, or certain lines of code within a routine. To minimize the amount of code duplicated, we conditionalized on the smallest code segment possible, using a sensible approach. For example, when forced into using conditional code, we avoided duplicating modules by choosing to keep within a single module. Ideally, we conditionalized just a few lines. Wherever possible, BLISS macros were modified to hide the code conditionalization.

### *Rdb Is Rdb*

We wanted our database management products to "look and feel" the same on an Alpha AXP system as they did on a VAX system. So, to paraphrase from the OpenVMS operating system maxim, we wanted Rdb to be Rdb! That is, the ported Rdb should have the same utilities, the same data structures, the same data definition capabilities, the same data manipulation constructs, etc., as the DEC Rdb for OpenVMS VAX product. Incorporated in this desire for sameness was the fundamental point that we were not going to change the on-disk structures. DBMS was ported with the same goal in mind.

### *No Changes to On-disk Structures*

The KODA kernel stores records on database pages. Unfortunately, the database page is not naturally

aligned; page header fields and fields within the records are not aligned. Although aligning these fields would boost performance, to realign all the structures on the database page would require the database to be unloaded and then reloaded. Current customers cannot afford the downtime needed to perform the conversion, so we decided to maintain the same page/record structure. Furthermore, by maintaining the same on-disk structure for the VAX and Alpha AXP databases, we do not preclude future concurrent access to the database in a mixed-architecture VMScluster. Thus, our present design does not require an unload/reload operation, since performing that action would be too much of an impediment to migrating to the Alpha AXP platform. However, we do plan to investigate the potential performance boost from aligned pages/records and, if the gain is substantial, to offer some alignment solution. Note that this section refers only to data structures tied to on-disk structures. We did align all in-memory structures, and we elaborate on this topic in the next section.

## Porting Details

In this section we describe a general set of issues and solutions that applied to all the groups involved in porting the database management system software to the Alpha AXP platform. We then explain some of the more interesting issues and solutions pertaining to each group.

### Common Issues

A collection of general porting issues applied to the Rdb, DBMS, and KODA groups. For example, all groups needed the capability to conditionalize code in a module, so that the compiler on an Alpha AXP system would produce one set of object code, and the compiler on a VAX system would produce another set. Common issues were:

- Varianted code
- Data alignment and field resizing
- Argument-passing mechanism
- BUILTIN functions
- VAX testing
- The CALLG mechanism and AP references
- VAX MACRO-32 modules
- Message file support

*Varianted Code* To simplify conditional code, we added a set of literals, for example KOD$K_VAX or KOD$K_ALPHA, that can be used in all our BLISS modules. We could then use these literals to conditionalize code. The code example shown in Figure 2 illustrates the conditionalizing of the PROBE instruction. The PROBE instruction checks the read/write access of a memory location. On Alpha AXP systems, the instruction is quite different from the corresponding instruction on VAX systems. However, BLISS easily handles this difference in a macro, which allows us to change the name and the order of the arguments, pass arguments by value instead of reference, and use an offset instead of a length. By developing such a macro, the actual source code did not have to change.

*Data Alignment and Field Resizing* On the first pass, we immediately modified all in-memory data structures so that they were naturally aligned. This step avoided incurring a significant performance penalty on the Alpha AXP platform. In addition, since no single Alpha AXP instructions exist that could be used to easily manipulate bytes or words, many of our in-memory byte (8-bit) and word (16-bit) fields were changed to longwords (32 bits) to reduce the object code size and improve performance.

```
$PROBER (BASE, LEN = 4, MODE = 0) =
    %IF     KOD$K_ALPHA
    %THEN   (BUILTIN PAL_PROBER;
             PAL_PROBER (BASE, LEN - 1, MAX (MODE, $PREV_MODE)))
    %ELSE   (BUILTIN PROBER;
             PROBER (%REF (MODE), %REF (LEN), BASE))
    %FI %,
```

*Figure 2 Conditionalized PROBE Instruction*

Once we aligned the in-memory data structures, two groups of data structures remained unaligned: those tied to the database root file, which records database parameters such as associated files and database settings, and the database pages that actually contain the data records. Since the database root file is relatively small (i.e., less than 100 blocks in size), it was aligned also. Thus, the root file is automatically re-created in a conversion that occurs when upgrading a database product to support both the Alpha AXP and VAX architectures. Since this conversion invariably takes place when converting to a newer version of either the Rdb or the DBMS product, the additional realignment of the root is a minor additional expense.

Thus far, we have not pursued any potential modifications of the page data structures, such as aligning them once they are fetched into memory. Note that these structures do not generate unaligned faults. Instead, they force the compiler to generate a few additional instructions to handle the odd alignment.

*Argument-passing Mechanism* The VAX and Alpha AXP argument-passing mechanisms are entirely different. Rather than using the standard BLISS mechanism, the existing code depended strongly on the VAX argument-passing mechanisms by using BLISS macros to reference arguments from the AP. This approach was not possible on Alpha AXP systems due to the lack of an AP register. (You could force the AP to be generated, but that process would be slow and would waste memory.) Therefore, we changed our procedure headings to declare a generic formal parameter list (e.g., P1 through P*N*) for both the Alpha AXP and the VAX systems and then developed another set of BLISS macros that allowed us to bind to the arguments based on the generated formal parameter list. Since this process involved changing every routine declaration, we developed a text-processing tool that would automatically change the routine headings and thereby avoid the expensive and error-prone task of manually changing each routine.

*BUILTIN Functions* Together, the KODA, Rdb, and DBMS code uses most of the BLISS BUILTIN functions. This fact presented a problem for the team porting the software to the Alpha AXP platform. Some VAX BUILTINs were not supported, some behaved differently, and some were eliminated as BUILTINs but emulated by Starlet, an OpenVMS support library. Again, we used BLISS macros to solve the problem. Essentially, our macros categorized the BUILTINs and then performed the appropriate expansion, based on the category. For example, the PROBE BUILTIN differed markedly between the VAX and Alpha AXP implementations, as indicated by Figure 2.

*VAX Testing* Another general problem that we had to guard against was the possibility that the Alpha AXP code changes would introduce bugs into the VAX versions of the products. Consequently, we adopted a policy whereby all Alpha AXP changes had to be tested on a VAX system. This policy ensured that we maintained a steady pattern of correct VAX behavior. Also, since the VAX environment was more stable than the Alpha AXP environment, testing on a VAX system helped tremendously in identifying and fixing bugs related to the port.

*The CALLG Mechanism and AP References* The Alpha AXP platform does not directly support CALLG, a VAX procedure calling mechanism, and references to the AP. The CALLG mechanism and AP references are slow since they are simulated and automatically allocate stack space to accommodate the largest possible argument list (i.e., 255). In situations where performance was not critical, for example, in an error handler, we replaced CALLG by a standard routine call on both the VAX and the Alpha AXP software versions. When performance was an issue, we used conditional code to retain the CALLG mechanism for the VAX code and to use a standard routine call in the Alpha AXP code. In instances where the CALLG mechanism is used to pass the argument list to the next routine, we constructed an argument vector and replaced CALLG by a special call linkage. The new mechanism passed the pointer to the argument vector by means of a single parameter or a global register. This solution guaranteed good performance on both VAX and Alpha AXP systems yet avoided any conditionalizing of the code.

*VAX MACRO-32 Modules* For a variety of reasons, we used VAX MACRO-32 to code some routines in the Rdb, DBMS, and KODA software. For example, basic operations such as record compression, record expansion, and buffer initialization are performed through calls to VAX MACRO-32 routines that are heavily optimized for efficient operation. Some routines are coded in VAX MACRO-32 for ease

of character manipulation. Also, we used VAX MACRO-32 to code machine instructions that were not available through a BLISS BUILTIN function.

We adopted various solutions for these VAX MACRO-32 routines. For those routines where performance was not an issue and BLISS generated acceptable code, we converted to BLISS code. For routines where performance was absolutely critical, we rewrote the routine in Alpha AXP MACRO-64 to utilize the additional registers. Finally, in some cases where we could not rewrite the routine in BLISS code and did not have the resources to convert to MACRO-64 code, we employed the Alpha MACRO cross compiler.

*Message File Support* Due to the structure of the database products, as shown in Figure 1, each component has separate message files. Both Rdb and DBMS have a message file that is separate from the KODA message file. Furthermore, the Rdb and DBMS software share the KODA message file.

The message files are merged during the build cycle, so that customers are not required to be aware of the modular layout of the code. As a result, KODA messages, when appended to Rdb's message file, print as Rdb messages (e.g., RDMS-F-msgcode, message text). However, the Rdb source code still references the KODA message codes with the KOD$_ message prefix.

Prior to the introduction of the Alpha AXP architecture, the KODA messages were defined with .LITERAL declarations in the message files. Since we occasionally link images with multiple message files, we wrote a program that would read an .OBJ file and write a new .OBJ file without writing the KODA literal declarations. This process would no longer work since Alpha AXP object files have a different format than VAX object files. As a result, we changed the mechanism to define the KOD$_ symbolic values to be compatible with both the VAX and Alpha AXP architectures.

First, we removed all .LITERAL declarations from the KODA message file. As a result, all KODA messages were defined strictly as RDMS or DBMS messages. Then, after passing the message source file through the message compiler to get the message object file, we invoked the ANALYZE/OBJECT facility to get a listing of the message symbol codes and values for each message. Finally, we wrote a small utility to read the ANALYZE/OBJECT output and generate a BLISS .B32 file, which is shown in Figure 3.

This BLISS program, when compiled and included in an executable image, defines the appropriate KOD$_ message codes and their associated values. This procedure is used on both the OpenVMS VAX and the OpenVMS AXP operating systems to generate the message files. Furthermore, since this group no longer writes programs that read object code, the resulting method is easier to maintain.

The following three sections discuss some problems encountered by each of the porting teams.

## *Porting the KODA Database Kernel*

Among the issues that the KODA group dealt with were those related to calling mechanisms, kernel-mode rundown handlers, and a bugcheck dump mechanism.

*Stack-switching/Stall Mechanism* The KODA database kernel performs its own multithreading activities. A single process can be actively attached to multiple databases in the context of a single instantiation of the software. For example, in the DBMS interactive query (DBQ) facility, the user can perform the following operation:

```
MODULE DBMKODMSG =
BEGIN

GLOBAL LITERAL KOD$_ABORT_WAIT          = %X'0028800C';
GLOBAL LITERAL KOD$_ACCVIO              = %X'002885EC';
GLOBAL LITERAL KOD$_AIJACTIVE           = %X'00288BA3';
GLOBAL LITERAL KOD$_AIJALLDONE          = %X'00288B33';
...

END
ELUDOM
```

*Figure 3 BLISS Code to Generate KOD Message Definitions*

```
dbq> ! Attach to first database as user1.
dbq> BIND DB1 ON STREAM 1
dbq>
dbq> ! Attach to second database as user2.
dbq> BIND DB2 ON STREAM 2
dbq>
dbq> ! Establish user1 context.
dbq> SET STREAM 1
```

This example has the user attached to two different databases, DB1 and DB2. To issue queries against either database, the user enters the SET STREAM command. In response, KODA establishes the correct data structures and stream context for this database session. This process involves switching data structures and stack context. Consequently, KODA manages its own stack for its executive mode code and data structures. This stack-switching mechanism is complex, and this code is intimately tied to the VAX procedure calling mechanism. For example, whenever a query must stall (e.g., while waiting for a lock request), KODA saves the current executive mode context and then switches back through the stream code out to user mode. This action allows the process to receive user-mode ASTs. This mechanism essentially saves a call frame so that after the user-mode stall has completed, KODA can set up the appropriate stack and return to the calling routine by means of the saved call frame.

The calling/return mechanism is entirely different for the VAX and Alpha AXP architectures. On Alpha AXP systems, for each routine, the compiler generates prologue code and epilogue code to manage the routine calling mechanism. Accordingly, the KODA stack mechanism had to rely on this new mechanism. In addition, for this level of support, the routine that was coded in BLISS for the VAX platform had to be coded in MACRO-64 on the Alpha AXP platform.

*Kernel-mode Rundown Handlers* Another example of KODA's close tie to OpenVMS behavior involved the use of KODA's kernel-mode rundown handler. On VAX systems, in the event of an abnormal failure, we must clean up certain data structures and release resources such as locks or channels. Furthermore, database recovery must start before the image rundown is completed, so that surviving processes cannot acquire locks on resources before the databases are recovered.

We accomplish this image cleanup through the use of a user-defined system service (i.e., a system service not defined by the OpenVMS system), which acts as a kernel-mode rundown handler. In addition to releasing database resources, the handler also cleaned up OpenVMS data structures such as the pending AST queue. These OpenVMS data structures changed significantly for the Alpha AXP architecture. For example, an Alpha AXP system has five pending AST queues instead of one. In addition, this handler routine would acquire the OpenVMS scheduler spinlock and perform "poor man's lockdown," which effectively pages the entire routine into memory (since the code cannot incur a page fault at elevated interrupt priority level, IPL). For Alpha AXP, code and data cannot be located in the same PSECT, so this trick was not possible. Instead, we used the $LKWSET macro to lock pages in memory and then to clean up the OpenVMS data structures.

After we completed and tested the code, the database and OpenVMS engineering teams decided that such intricacy was needlessly complex, and that the OpenVMS AXP software could clean up the data structures based on its image control block and related structures. This example shows how the OpenVMS AXP system offers different functionality than the OpenVMS VAX system, i.e., the port offered the opportunity to clean up existing mechanisms.

*Bugcheck Dump Mechanism* Complex, sophisticated software products are by nature difficult to debug. Most of these products utilize a data structure dumping mechanism whenever an internal software or hardware error is encountered. KODA has a mechanism called a bugcheck dump that performs this service. When an unexpected exception is generated, the bugcheck dump code prints all relevant data structures into a file. In addition, the dump includes a stack dump. On VAX systems, the bugcheck dump traces back down the stack using the saved call frames and prints out all the fields in each call frame, the routine name, and the arguments passed.

In particular, the method for printing the symbolic name of the routines is especially clever. After linking an image, we utilize a program that scans the symbol table (.STB file) produced by the linker. Then the program creates its own object file, which includes a relative offset of all the routines and their symbolic names. Finally, the image is relinked, and this new object file is included into the image in a particular PSECT. When tracing back down the call frames, the bugcheck dump also checks the special PSECT to locate and print the correct routine name. This dump is an invaluable tool in determining the causes of unexpected errors. Figure 4 includes two

```
Saved PC = 000408AF : DIO$FETCH_DBKEY + 0000004F
ARG# Argument [data...] ----------------------------------------------------
   1 002064B4: 0001FCFC 002064F4 0020650C 207C0000 000277C7 00010000 00020001
   2 00000001
        Handler = 00000000, PSW = 0000, CALLS = 1, STACKOFFS = 0
        Saved AP = 0020644C, Saved FP = 00206430, PC Opcode = E0
SR2 = 002646D0: 00000000 00000000 00006918 FFDAA3E8 FFF63770 00000000 00000000
SR3 = 0000BC41: 013A2048 C2FFFFFF FFFFF85E E00095B7 D512A4E0 40000000 18C00040
SR4 = 002646B0: 00000008 0020645C 002646A0 00000000 00000000 00000000 00000000
        20 bytes of stack data from 0020641C to 00206430:
002646B000000001002064B400000002  0000   '....4d .....0F&.'
                          001C7D08  0010   '.}..'

Saved PC = 00055241 : PSI$MODIFY_STITM + 00000033
ARG# Argument [data...] ----------------------------------------------------
   1 002064B4: 0001FCFC 002064F4 0020650C 207C0000 000277C7 00010000 00020001
   2 00000096
   3 002646D0: 00000000 00000000 00006918 FFDAA3E8 FFF63770 00000000 00000000
        Handler = 00000000, PSW = 0000, CALLS = 1, STACKOFFS = 0
        Saved AP = 00206490, Saved FP = 00206464, PC Opcode = DD
SR2 = 00256042: 00020096 0000005F 00000057 00000000 00000002 00010000 002E2A13
SR3 = 00264680: 00000000 00000001 00000008 002646A0 00264670 00000000 00000000
        24 bytes of stack data from 0020644C to 00206464:
002646D000000096002064B400000003  0000   '....4d .....PF&.'
                  001C7CF8002646C0  0010   '@F&.x|..'
```

*Figure 4 Bugcheck Dump*

routine calls from a stack trace, indicated by the lines of code that begin with "Saved PC."

Alpha AXP systems have no equivalent to the VAX call frames, so it is impossible to use the call frame mechanism to trace down through the stack. As mentioned previously, Alpha AXP routines utilize prologue and epilogue code for returning from routine calls. Procedure descriptors contain information such as entry address and register save information.

On Alpha AXP systems, another Digital group supplied a set of routines that allows tracing the call sequence. This set provided the basic capability to print the routine calling sequence that led to an abnormal exception. In addition, the Alpha AXP linker produced a symbol table file. However, we decided to simplify our bugcheck mechanism. Although we still search the symbol table file for all routine addresses, rather than create an Alpha AXP object file, we create a VAX MACRO-32 file that includes the routine name and address/offset. Then, we simply use the Alpha MACRO cross compiler to generate the Alpha AXP object, which gets linked into the image on the second pass. In fact, we changed our VAX bugcheck routine to produce a MACRO-32 file with routine name and offsets. This process is simpler than directly creating an object file, as we did previously.

Even though the routines provided this call traceback capability, we were missing the arguments passed to the routines, perhaps the most important part of the stack trace. The VAX mechanism captured this data, because very often a bugcheck results from one routine passing an improper argument to another routine. The Alpha AXP system does not provide a way to capture this information, because the routine calling sequence reuses registers R16 through R21 for passing arguments.

## *Porting Rdb*

Some issues handled by the Rdb porting group were associated with the dispatch code, Alpha AXP code generation, Rdb precompilers, and Rdb system relations.

*Dispatch Code* The dispatch code is the topmost layer of the Rdb software and is called directly by the user application by means of relational call interface (RCI) calls.[2] The main function of dispatch code is to direct the user request to the correct target Rdb executive (local or remote) for processing. On VAX systems, the dispatch code passes the user

arguments to the Rdb software using the CALLG linkage.[3] On Alpha AXP systems, CALLG linkage is very inefficient. Therefore, the dispatch code was changed to build a user argument vector in the same style as the VAX argument list, and the pointer to the argument vector was passed as a single parameter. The code in Rdb was changed to bind to the user arguments using the offset from the pointer to the argument vector.

Using two different calling mechanisms in the dispatch to pass user arguments was a careful design. On VAX systems, the existing CALLG mechanism was retained to ensure backward compatibility between different versions of the Rdb dispatch, Rdb layered products, and gateways. A new calling mechanism was used on Alpha AXP systems to ensure good performance, since every user request to the Rdb executive goes through the dispatch.

*Code Generator* Rdb uses compiled BLISS code and generated machine code to execute user requests. During request compilation, Rdb generates highly efficient routines using the target machine instructions. These routines perform basic data operations including data conversion, data movement between buffers, aggregation, and expression evaluation.

The design of the Rdb code generator to produce Alpha AXP machine code was undoubtedly the most complex porting task. Use of a mechanism other than code generation would have reduced the porting effort. However, at the time we began porting Rdb, it was not clear if an alternate mechanism would guarantee an acceptable level of performance. Good performance was considered critical to the success of Rdb on Alpha AXP systems. Therefore, we decided to add functionality to the Rdb code generator to produce Alpha AXP code. To generate efficient Alpha AXP code sequences, we observed specific guidelines.[4]

On Alpha AXP systems, code that references data items with increasing memory addresses executes more efficiently. Therefore, the algorithm was changed to first order the data items by increasing memory addresses and then generate code to process the data.

In Rdb, each data item has a null bit that indicates whether or not the value of the data item is known. As shown in Figure 5, to conserve space, the null bits of different data items are stored together like a bit vector within a record. Loading/storing a null bit is an expensive operation on Alpha AXP systems.[4] Therefore, the algorithm was modified to fetch a batch of null bits into a register. When all null bits in the register are processed, the batch is written and the next batch of null bits is fetched. This approach reduced the number of load and store instructions and made the code sequence much more efficient.

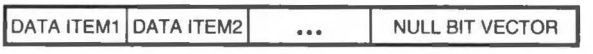



*Figure 5 Rdb Record Layout*

On Alpha AXP systems, the machine code routines generated by Rdb use four different addressing modes to access data items: absolute address, base register plus offset, integer register content, and floating-point register content. Each of the Alpha AXP registers R12 through R15 is used as a base register. Thus, any data stored within 256K ($4 \times 64K$) of memory space can be accessed efficiently. To maximize data access efficiency and caching, changes were made in the code generator to allocate data densely. To improve performance further, data items were allocated at quadword or longword aligned addresses.

An Alpha AXP code sequence executes more efficiently when instructions can be multi-issued and executed in parallel. This can be achieved by reordering the sequence of instructions while maintaining any chronological dependency between instructions. To take advantage of this Alpha AXP feature, BLISS macros were developed to reorder and interleave the instructions in a generated code sequence.

On Alpha AXP systems, backward branches in the code slow down the execution because of instruction stream invalidation.[4] Changes were made in the Rdb code generator to minimize backward branches. This change at times increased the size of the generated code but improved the code execution efficiency. Further, Boolean code generation algorithms were modified to incorporate branch prediction logic; code sequences with a smaller probability of execution were branched out of the main code stream. This technique maximized the effect of instruction stream caching.

*Rdb Precompilers* An Rdb precompiler preprocesses a user application program that includes Rdb statements and replaces these statements by standard RCI calls to the Rdb software.[2] The Rdb

statements embedded in the applications can be one of three types: structured query language (SQL), Rdb preprocessors language (RdbPRE), or relational data manipulation language (RDML). There are three different Rdb precompilers to support these languages.

The SQL precompiler, an industry-standard language interface to Rdb, is a strategic Rdb component. A long-term goal of this precompiler is flexibility in future developments and ease of maintenance. To meet this goal, the SQL precompiler was redesigned to use the GEM compiler on Alpha AXP systems to preprocess SQL application programs and produce Alpha AXP object code.

The RdbPRE precompiler is a proprietary language interface to Rdb. The long-term goal is no new functionality and minimal maintenance. So the main objective was to reduce the effort required to port this compiler. This was achieved by retaining the existing design and using the Alpha MACRO cross compiler to produce Alpha AXP objects from VAX MACRO-32 files.

The RDML precompiler is also a proprietary language interface to Rdb. Unlike the RdbPRE precompiler, this compiler does not produce VAX MACRO-32 files. So porting it was an easy and straightforward task.

*Rdb System Relations* Rdb uses system relations to record information about the user relations and the database. The system relations are stored on disk and loaded into memory on demand. Since they are frequently referenced during user request processing, efficient access to data in system relations is critical for performance. On Alpha AXP systems, accessing data from memory is efficient if it is located on either a longword or a quadword address boundary.[4] Therefore, changes were made to the in-memory system data structures to align each data field to at least a longword address boundary. Further, data fields that were a byte or a word were expanded to a longword.

The data in system relations was accessed by using RdbPRE statements embedded in Rdb source modules. Porting such Rdb modules posed a dilemma. To compile these modules, first the RdbPRE compiler had to be ported to the Alpha AXP platform. Vice versa, to port and test the RdbPRE precompiler, Rdb had to be ported and running on the Alpha AXP platform. Moreover, RdbPRE was no longer a strategic language interface. Therefore, new BLISS macros were designed that replaced the embedded RdbPRE statements.

## *Porting DBMS*

This section discusses some experiences of the DBMS porting group, namely those related to the Database Control System (DBCS) interface, the H_FLOAT data type support, and the use of the Alpha User-mode Debugging Environment (AUD).

*DBM$32, the Primary Interface to the DBMS* The DBCS for the DBMS software uses a single subroutine (DBM$32) as its primary entry point. This entry point is used by the DBMS precompilers (FDML, for Fortran, and DML, for other languages except COBOL), as well as other layered products, such as COBOL and DATATRIEVE.

After receiving control, DBM$32 performs some processing and then, using the CALLG mechanism, passes the entire argument list to lower-level routines for further processing. These lower-level routines, in turn, often pass on the argument list, sometimes as deep as five or six levels.

Because we found CALLG to be inefficient, we decided to change the primary entry point into the DBCS. Rather than passing up to 26 separate arguments, DBMS creates a vector of longwords; each longword contains an argument that would have been passed using a parameter. Once this vector is created (often during the compilation phase for the precompilers), DBM$32_VEC (the VECTOR version of DBM$32) is called with a single parameter: the address of the argument list. An example is shown in Figure 6.

Layered products using DBMS were advised of the new interface and were requested to use it as soon as possible. However, since the changed interface was incompatible with some existing products, the old interface was retained. DBM$32_VEC uses the new interface, and DBM$32 homes the argument list (thus creating the above vector) and then passes that, by reference, to DBM$32_VEC.

*Support of H_FLOAT Data Types* The H_FLOAT data type is fully supported on the VAX processor, but the Alpha AXP processor has no high-precision floating-point formats. Although facilities exist on Alpha AXP processors to read an H_FLOAT data type, no such facility exists to write an H_FLOAT data type.

As a result, DBMS customers are advised to eliminate any H_FLOAT data in databases before moving them to an Alpha AXP system. The DBMS Database Restructure Utility (DRU) can be used to change all H_FLOAT data to another common floating-point format.

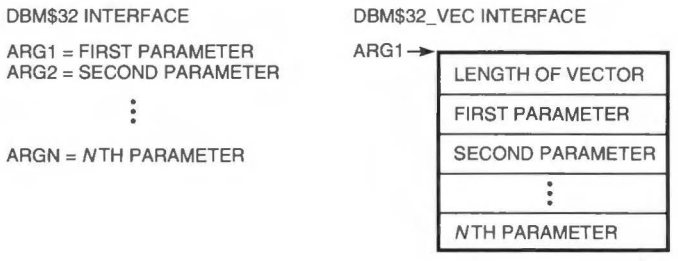



*Figure 6 DBCS Routine-calling Interface*

In preparation for mixed VAX and Alpha AXP VMScluster systems, DBMS was modified such that databases with H_FLOAT data can still be accessed. However, a run-time conversion error occurs if H_FLOAT data is accessed from an Alpha AXP system.

*Use of AUD* The Alpha User-mode Debugging Environment is a set of facilities that aids testing and debugging of native Alpha AXP code on any OpenVMS VAX system. AUD allowed as much Alpha AXP user-mode code as possible to be ported immediately to the Alpha AXP system and to be substantially debugged before Alpha AXP hardware was available. Early in the DBMS porting effort, we used AUD to verify our port and to ensure that our code was working correctly.

However, several issues hampered the success of using AUD in porting the DBMS software:

1. DBMS makes frequent use of signaled exceptions. AUD had difficulty in handling exceptions that cross the boundary between the Alpha AXP and VAX systems.
2. DBMS uses special stack manipulation code (stream code) to perform multithreading functions. AUD would become confused if the stack were to change unexpectedly.
3. At the time we were using AUD, the DBCS had been ported, but KODA (i.e., the low-level services used by the DBCS) had not. As a result, many variables needed to be defined as crossing the boundary between the Alpha AXP and VAX systems. The setup time to define this information was significant.
4. Since the code was still running on a VAX processor, many VAX dependencies were not caught by AUD. In particular, system services that changed in subtle ways would work as before because the operating system was still the OpenVMS system.
5. Most of the changes that we made in DBMS were not conditional, that is, the changes would affect both VAX and Alpha AXP systems. As a result, we were able to test our code on VAX systems with a fairly high degree of certainty that our code was correct, barring any operating system or compiler bugs.

We did eventually get an AUD version of DBMS working. However, since we spent a considerable amount of time accomplishing this, and we did not actually find any bugs in our code by using AUD, we decided not to use AUD in further areas of DBMS.

Shortly after using AUD, we received our Alpha Demonstration Unit (ADU) and could test our code on actual Alpha AXP hardware. The only problems we found, which were missed during our initial port, were VAX-style argument list assumptions. Some of our code assumed that routine arguments were contiguous in virtual memory; on Alpha AXP systems, this is not the case.

## Conclusion

To conclude the paper, we discuss our plans for performance testing and our reflections on the porting process.

### Performance

We have only begun our performance tests. Currently, we are running the TPC-B performance benchmark. We also plan to test against all TPC benchmarks (A, B, and C) and other benchmarks such as the Wisconsin benchmark. We are trying to minimize the amount of time spent in PALcode, decreasing the code path length, reducing the cycles per instruction, and optimizing internal algorithms.

Planned testing will also evaluate the effect of additional data alignment. As mentioned earlier, the ease-of-migration issue is paramount for our current customers. Consequently, we have not realigned the database pages because that action would

require too much downtime. Nevertheless, we do not want to preclude new customers, or current customers who need the performance boost, from utilizing a properly aligned database page. To test the potential performance improvement, we plan to create a test database that is completely aligned, in memory and on disk, and compare the TPC performance against the standard database.

### *Reflections*

At the beginning of the paper, we stated that our goal was for Digital to provide an easy migration path to the Alpha AXP platform for software products. Although we encountered some difficulties, we believe our Rdb and DBMS porting efforts attest to Digital's success in this endeavor.

As one example of how the experience influenced our approach to porting, we had to learn new methodologies, practices, and system behavior on the Alpha AXP machines. For instance, when stepping through a particular code sequence with the debugger, we would end up in an infinite loop; if we just ran the code, the sequence would work. Although this behavior was documented, we encountered the problem several times before we fully understood the ramifications and appropriately changed our development methods.

Overall, the porting effort had the following positive results:

- The port allowed us to clean up our code, even though we tried to avoid algorithm changes. Because we had to port and review every line of code, we managed to move the code to a more consistent coding convention.
- The port acted as a learning experience for most of the engineers. Most mature products contain some code that has not been modified in years. The port forced us to review and understand such code sequences. As a result, we ended up with more knowledgeable engineers.
- The port allowed us to transform the code into a more portable state. As we moved away from tight ties to VAX behavior, we simplified future tasks such as moving to the OSF/1 and Windows NT operating systems.
- Although overlapping current VAX development with the Alpha AXP port slowed down the porting process, the decision to use a common code base eliminated the future need to integrate two divergent source codes.
- Surprisingly, the code did not grow appreciably in size or complexity. One strength of the Rdb and DBMS software has been the ability to easily modify the code and to add new functionality. Even after the port, we find that the products are as malleable and as easy to modify as before.
- We found unreported bugs in our VAX products.

Virtually all the groups involved did a masterful job. The program team and various Alpha AXP committees anticipated potential issues and ensured that the program proceeded smoothly and predictably. The cross compilers from the language groups worked superbly. The OpenVMS AXP and hardware groups delivered their products on time, and when a user logs in to an Alpha AXP system, the OpenVMS AXP system is not only familiar but faster.

## *Acknowledgments*

The successful port of the Rdb and DBMS software to the OpenVMS AXP operating system was a result of the contributions made by many of the engineers in the Database Systems Group. The authors sincerely acknowledge the effort of each engineer in achieving the project goal, that is, to be able to quickly offer correct versions of Rdb and DBMS on the Alpha AXP platform. Finally, an unsung hero in the company-wide effort was Digital's VAX Notes communications facility. VAX Notes functioned as an excellent medium for identifying and sharing problems and solutions.

## *References*

1. T. Leonard, *VAX Architecture Reference Manual* (Bedford, MA: Digital Press, Order No. EY-3459E-DP, 1987).
2. *DSRI Handbook* (Maynard, MA: Digital Equipment Corporation, Order No. AA-GV71A-TE, 1986).
3. *OpenVMS Calling Standard* (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQY2A-TK, 1992).
4. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, MA: Digital Press, Order No. EY-L520E-DP, 1992).

*James V. Colombo*
*Pamela J. Rickard*
*Paul Benoit*

# *DECnet for OpenVMS AXP: A Case History*

*The DECnet for OpenVMS AXP networking software facilitates the integration of OpenVMS AXP systems into existing DECnet computing environments. This new software product supports application migration by providing the following networking capabilities: support of compatible libraries, consistent application programming interfaces, and the assurance of a common semantic operation with the OpenVMS VAX system. The team implemented a phased porting process and executed the project cooperatively. The effort resulted in a solid knowledge base with which to approach future porting undertakings. Using common code where possible and avoiding architecture-specific code were lessons learned during the project.*

The DECnet for OpenVMS AXP networking software product plays an important role in the integration of OpenVMS AXP systems into existing DECnet computing environments. The availability of DECnet software on the Alpha AXP hardware platform facilitates application migration. The networking capabilities needed to support this migration activity include support of compatible libraries, consistent application programming interfaces (APIs), and the assurance of a common semantic operation with the OpenVMS VAX system. The network features such as network file transfer, remote file access, remote login, downline load, and local and remote network management allow the OpenVMS AXP system to participate fully in a DECnet network.

The purpose of this paper is to describe the process of porting the DECnet-VAX product to the OpenVMS AXP operating system. The DECnet-VAX product consists of networking software written in the MACRO-32 and BLISS-32 programming languages. The software contains privileged system code, device drivers, and user-mode utilities.

This paper is divided into two major sections. The first section presents an overview of the project, including discussions about the DECnet features supported in the OpenVMS AXP operating system, the project schedule, and the major DECnet for OpenVMS AXP components. The second major section details the process of porting DECnet-VAX software to the OpenVMS AXP operating system, including testing and debugging. This section provides information on nonportable coding practices and identifies specific problem areas. It concludes with a summary of the lessons learned during the course of the project.

## *Project Overview*

In addition to presenting the DECnet for OpenVMS AXP features, this section details how we derived a project schedule and gives an overview of the software components.

### *Software Code Base*

Prior to the formation of a team to port a DECnet product from VAX to the Alpha AXP architecture, the DECnet-VAX development group completed a feasibility study of porting DECnet-VAX Phase IV to the Alpha AXP architecture. This effort was necessary because the DECnet-VAX software was not designed with porting in mind. The study concluded that it would take four engineers twelve months (i.e., 48 person-months) to port DECnet-VAX to the OpenVMS AXP operating system. After examining the proposal and investigating the alternatives, we decided that the best approach would be to start by porting DECnet-VAX V5-4.3, a Digital Network Architecture (DNA) Phase IV implementation.[1] One of the most important factors in making this decision was that this software version was in external field test and was nearly ready for shipment to customers. Another consideration was that some very important fixes had been made in that release, and we wanted to offer our customers

the highest quality possible in the first version of DECnet for OpenVMS AXP software. Since that time, we have continued to improve our DECnet software for the OpenVMS AXP operating system and have recently incorporated some fixes from DECnet for OpenVMS VAX V5.5-2.

## *DECnet for OpenVMS AXP Features*

The first release of the DECnet for OpenVMS AXP networking product is packaged with the OpenVMS AXP operating system. The initial offering includes the support of DECnet Phase IV protocols running over Ethernet or fiber distributed data interface (FDDI) local area networks. This release supports distributed task-to-task communications using the same set of documented programming interfaces supported in the DECnet-VAX environment. At this time, DECnet for OpenVMS AXP software does not support wide area communications devices and host-based routing. Future releases of DECnet for OpenVMS AXP may include symmetric multiprocessor (SMP) and cluster alias support.

## *Project Schedule*

The DECnet for OpenVMS AXP project schedule was primarily driven by the overall OpenVMS AXP operating system product schedule, with the DECnet component scheduled for delivery in November 1991. The DECnet-VAX porting project officially began in early January 1991, after the code base was selected.

*Porting Estimates* After analyzing the work required to achieve the port, we developed general porting guidelines and estimates based on a number of factors, including the language the software was written in, the amount of software to port, and the number of software component modules. We then combined these estimates to determine an overall project schedule. Table 1 presents the guidelines we used for the porting estimates.

We used two methods to estimate the amount of work required to complete the port. The Module Size Method takes into account the number of lines of code per software module. The Module Count Method uses the number of modules per software component to determine the workload. Both methods take into consideration the programming language used in each module. Table 2 presents details of the component module count and sizes. We further categorized the software being ported into three groups: privileged code, device driver, and user-mode utility. The software type was used to estimate the amount of time needed for linking. In general, we allocated more time for privileged code and device drivers.

**Table 1 Guidelines for Porting Estimates**

| Language | Lines of Code (Per week) | Module Count (Per week) |
|---|---|---|
| BLISS | 10,000 | 10 |
| MACRO | 3,000 | 5 |

The estimates were used to derive a first-pass schedule and to determine resource allocation. A number of other factors affected the final schedule. A major factor that we could not quickly estimate was the portability of the software. The software techniques encountered and described in this paper such as coroutines, up-level stack references, and condition code usage had a direct impact on the schedule. Also, during the first three months of the project, significant time was spent learning how to port code. During this learning period, we developed the skills, knowledge, and techniques used throughout the remainder of our porting work.

Once we established the estimation metrics, the data was compiled and time estimates calculated for each component. Tables 3 and 4 show the average amount of time required to port each DECnet for OpenVMS AXP component.

Based on these calculations, we estimated that it would take 13 person-months just to port the DECnet-VAX software. We then used project management software to plan the schedule. The schedule shown in Table 5 indicated that it would take 48 person-months to meet the OpenVMS AXP scheduled completion date of November 22, 1991. We made our first network connection on July 25, 1991, 20 person-months into the project. Although much work remained, we were well ahead of the November target date.

Since we were ahead of schedule, we assisted in the porting of other components, including RTPAD, CTDRIVER, RTTDRIVER, and REMACP, all discussed later in the paper. In addition, we were able to add support for FDDI.

*Milestones* The OpenVMS AXP project schedule consisted of a series of functional internal base levels numbered one to five. In terms of the whole OpenVMS AXP project schedule, DECnet for

**Table 2 Component Module Count and Sizes**

| Component | Software Type | Language | Module Count | Number of Lines | Average Number of Lines |
|---|---|---|---|---|---|
| DTR/DTS | User | MACRO | 14 | 1937 | 138.36 |
| EVL | Privileged | BLISS | 10 | 3821 | 382.10 |
| HLD | Privileged | MACRO | 9 | 715 | 79.44 |
| MIRROR | Privileged | MACRO | 1 | 131 | 131.00 |
| MOM | Privileged | BLISS | 15 | 5835 | 389.00 |
| | | MACRO | 7 | 1182 | 168.86 |
| Subtotal | | | 22 | 7017 | 318.95 |
| NCP | User | BLISS | 35 | 19371 | 553.46 |
| | | MACRO | 2 | 712 | 356.00 |
| Subtotal | | | 37 | 20083 | 542.78 |
| NETACP | Privileged | MACRO | 24 | 20871 | 869.63 |
| NETDRIVER* | Driver | MACRO | 4 | 6891 | 1722.75 |
| NICONFIG | User | BLISS | 7 | 2078 | 296.86 |
| NML† | Privileged | BLISS | 31 | 19889 | 641.58 |
| | | MACRO | 7 | 4997 | 713.86 |
| Subtotal | | | 38 | 24886 | 654.89 |
| NETSERVER | Privileged | BLISS | 3 | 303 | 101.00 |

Notes:
* Includes estimates for NDDRIVER
† Includes estimates for NMLSHR

**Table 3 Module Size Method**

| Component | BLISS | MACRO | Link | Total Time per Component |
|---|---|---|---|---|
| DTR/DTS | 0.00 | 0.65 | 2.00 | 2.65 |
| EVL | 0.38 | 0.00 | 2.00 | 2.38 |
| HLD | 0.00 | 0.24 | 2.00 | 2.24 |
| MIRROR | 0.00 | 0.04 | 2.00 | 2.04 |
| MOM | 0.58 | 0.39 | 4.00 | 4.98 |
| NCP | 1.94 | 0.24 | 4.00 | 6.17 |
| NETACP | 0.00 | 6.96 | 6.00 | 12.96 |
| NETDRIVER* | 0.00 | 2.30 | 6.00 | 8.30 |
| NICONFIG | 0.21 | 0.00 | 2.00 | 2.21 |
| NML† | 1.99 | 1.67 | 4.00 | 7.65 |
| NETSERVER | 0.03 | 0.00 | 2.00 | 2.03 |
| TOTAL | | | | |
| Weeks | 5.13 | 12.48 | 36.00 | 53.61 |
| Months | 1.18 | 2.88 | 8.31 | 12.37 |
| Years | 0.10 | 0.24 | 0.69 | 1.03 |

Notes:
* Includes estimates for NDDRIVER
† Includes estimates for NMLSHR

Note that the data presented is in weeks, unless otherwise specified. A week equals five working days, a month equals 4.33 weeks, and a year equals 12 months or 52 weeks.

**Table 4 Module Count Method**

| Component | BLISS | MACRO | Link | Total Time per Component |
|---|---|---|---|---|
| DTR/DTS | 0.00 | 2.80 | 2.00 | 4.80 |
| EVL | 1.00 | 0.00 | 2.00 | 3.00 |
| HLD | 0.00 | 1.80 | 2.00 | 3.80 |
| MIRROR | 0.00 | 0.20 | 2.00 | 2.20 |
| MOM | 1.50 | 1.40 | 4.00 | 6.90 |
| NCP | 3.50 | 0.40 | 4.00 | 7.90 |
| NETACP | 0.00 | 4.80 | 6.00 | 10.80 |
| NETDRIVER* | 0.00 | 0.80 | 6.00 | 6.80 |
| NICONFIG | 0.70 | 0.00 | 2.00 | 2.70 |
| NML† | 3.10 | 1.40 | 4.00 | 8.50 |
| NETSERVER | 0.30 | 0.00 | 2.00 | 2.30 |
| TOTALS | | | | |
| Weeks | 10.10 | 13.60 | 36.00 | 59.70 |
| Months | 2.33 | 3.14 | 8.31 | 13.78 |
| Years | 0.19 | 0.26 | 0.69 | 1.15 |

Notes:
* Includes estimates for NDDRIVER
† Includes estimates for NMLSHR

Note that the data presented is in weeks, unless otherwise specified. A week equals five working days, a month equals 4.33 weeks, and a year equals 12 months or 52 weeks.

**Table 5 Planned Project Schedule**

| Component | Port | Debug | Code Review | Test | Total Time per Component |
|---|---|---|---|---|---|
| DTR/DTS | 4.80 | 4.00 | 2.00 | 6.00 | 16.80 |
| EVL | 3.00 | 4.00 | 2.00 | 6.00 | 15.00 |
| HLD | 3.80 | 4.00 | 2.00 | 6.00 | 15.80 |
| MIRROR | 2.20 | 4.00 | 2.00 | 6.00 | 14.20 |
| MOM | 6.90 | 4.00 | 2.00 | 6.00 | 18.90 |
| NCP | 7.90 | 4.00 | 2.00 | 6.00 | 19.90 |
| NETACP | 10.80 | 8.00 | 6.00 | 6.00 | 30.80 |
| NETDRIVER* | 6.80 | 8.00 | 6.00 | 6.00 | 26.80 |
| NICONFIG | 2.70 | 4.00 | 2.00 | 6.00 | 14.70 |
| NML† | 8.50 | 4.00 | 2.00 | 6.00 | 20.50 |
| NETSERVER | 2.30 | 4.00 | 2.00 | 6.00 | 14.30 |
| TOTALS | | | | | |
| Weeks | 59.70 | 52.00 | 30.00 | 66.00 | 207.70 |
| Months | 13.78 | 12.00 | 6.92 | 15.23 | 47.93 |
| Years | 1.15 | 1.00 | 0.58 | 1.27 | 3.99 |

Notes:
* Includes estimates for NDDRIVER
† Includes estimates for NMLSHR

Note that the data presented is in weeks, unless otherwise specified. A week equals five working days, a month equals 4.33 weeks, and a year equals 12 months or 52 weeks.

OpenVMS AXP was targeted for base level five. However, it was highly desirable to provide file transfer and remote login capability over DECnet as early as possible. The project team worked closely with the OpenVMS AXP group to deliver this support prior to base level four.

### Common Code

One of the most important decisions that helped us deliver our software ahead of schedule was building common code for the VAX and Alpha AXP systems. During the course of porting code, we discovered two advantages to building common code. The first was having the ability to generate VAX and Alpha AXP images from a single set of source code. The second was being able to debug our ported changes in a stable OpenVMS VAX environment. We accomplished this by rewriting code that required change so that it worked on both platforms. We made architecture-specific code conditional on the platform on which it would execute. Our long-term goal is to incorporate common code into future DECnet for OpenVMS products.

### DECnet for OpenVMS AXP Components

This section describes the major DECnet for OpenVMS AXP components and lists the porting issues relevant to each.[2] Figure 1 shows the interconnection of the various components of the DECnet for OpenVMS AXP software. Detailed information for each porting issue is further discussed in this paper under the Porting Issues heading.

*NETDRIVER* NETDRIVER is a pseudo device driver, i.e., a device driver that does not directly control any hardware devices. It implements the routing, end communication, and session control layers of the Phase IV version of DNA.[1]

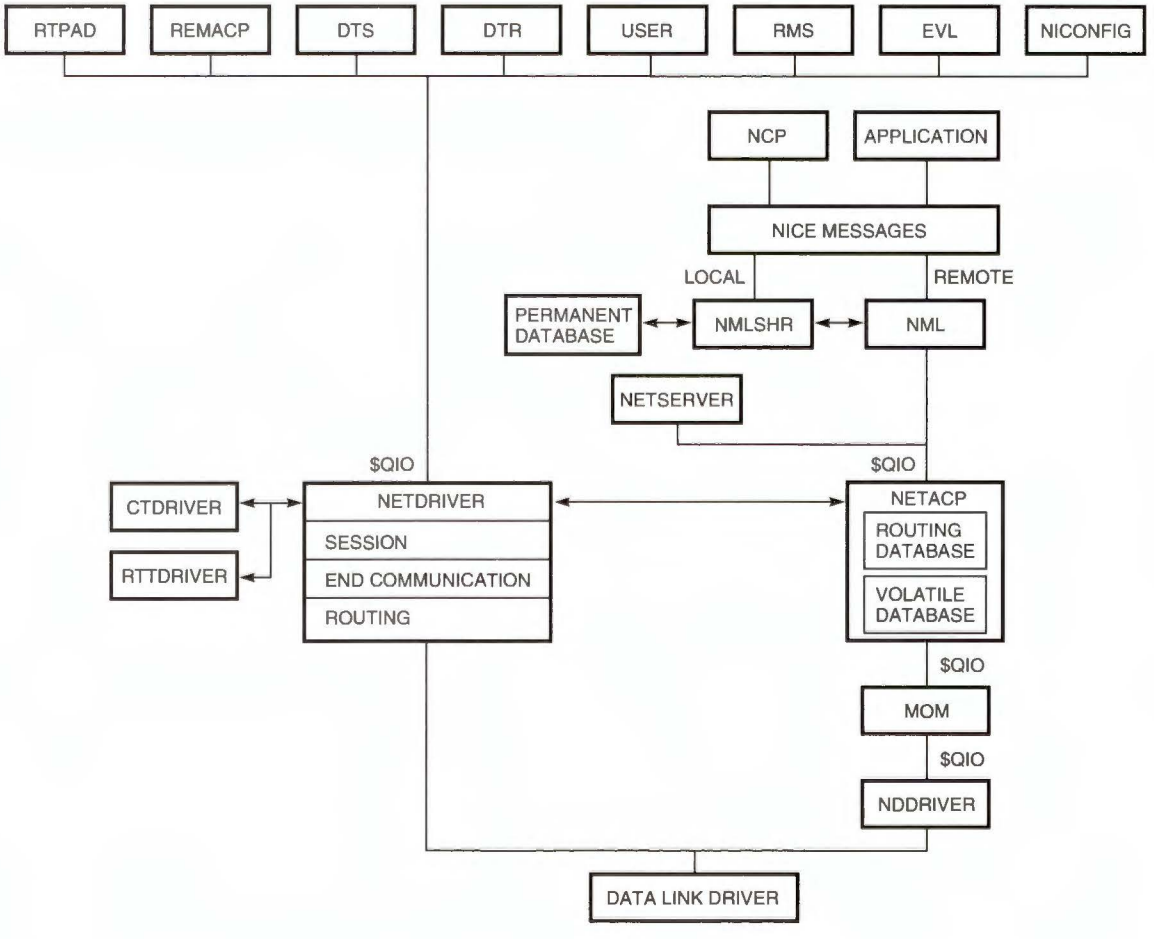



*Figure 1 DECnet for OpenVMS AXP Components*

The queue I/O request ($QIO) system service is the interface into the session control layer. The NETDRIVER routing layer communicates with other device drivers that implement the data link layer of DNA. NETDRIVER communicates with NETACP (another component discussed later in this section) to perform network management functions, to report state and network topology changes, and to perform operations that require process context.

NETDRIVER is written in MACRO-32 code and presented us with many porting issues, including device driver changes, coroutines, memory management changes, page size dependencies, atomicity and granularity problems, OpenVMS AXP operating system data structure changes, unaligned references, and up-level stack references.

*MOM* The maintenance operations module (MOM) image processes service operations defined by the maintenance operation protocol (MOP). One such service operation is downline loading remote systems. MOM uses NDDRIVER (described in the next subsection) to communicate with the remote system over a DECnet circuit. MOM communicates with NETACP to gather information about nodes requesting to be downline loaded. NETACP creates a process running the MOM image when a request for a service operation is received on a circuit enabled to perform service operations.

MOM is written primarily in BLISS-32 code. Porting issues included removing dependencies on the format of a VAX argument list, condition handling changes, and Alpha AXP image header changes.

*NDDRIVER* The pseudo device driver NDDRIVER implements an interface that allows MOM to use a DECnet circuit to perform service operations using DNA MOP. The MOM image uses the $QIO system service interface to send MOP messages to and receive MOP messages from NDDRIVER, which then communicates with the data link device drivers to transmit and receive these messages. NDDRIVER communicates with NETACP to perform tasks that require process context and to receive notification of state changes to circuits enabled for service operations.

NDDRIVER is written in MACRO-32 code. Porting issues included changes to device drivers, memory management, and OpenVMS AXP operating system data structures, as well as page size dependencies.

*CTDRIVER, RTTDRIVER, and REMACP* CTDRIVER is a pseudo device driver for remote terminals using the DNA command terminal (CTERM) protocol. CTDRIVER and RTTDRIVER perform similar functions with the exception that RTTDRIVER is used for interoperability with older implementations of remote terminal support. REMACP is an ancillary control process (ACP) that receives incoming requests for remote terminal support. After REMACP establishes a connection with the remote node, either CTDRIVER or RTTDRIVER communicates directly with NETDRIVER to send and receive remote terminal protocol messages.

CTDRIVER, RTTDRIVER, and REMACP are written in MACRO-32 code and presented the following porting issues: device driver changes, unaligned references, OpenVMS AXP operating system data structure changes, and for REMACP, changes in the interface with CTDRIVER.

*NETACP* NETACP runs as an ACP that assists NETDRIVER in performing network operations that require process context. Examples include creating processes for incoming logical links and assigning channels to data link devices. NETDRIVER and NETACP also work together to maintain information about the state of the network. Another major function performed by NETACP is the management of the network configuration parameters residing in virtual memory.

NETACP is written in MACRO-32 code. Porting issues included coroutines, usage of processor status longword (PSL) condition codes, memory management changes, page size dependencies, atomicity and granularity problems, OpenVMS AXP operating system data structure changes, and unaligned references. In addition, the use of "poor programmer's lockdown," a method of locking pages into a working set, required modification.

*NETSERVER* The NETSERVER image is run by server processes created to handle incoming logical link requests. NETSERVER invokes the image or command procedure associated with the network object specified by the incoming logical link. To avoid the overhead of process creation, a server process can be reused after the logical link it was servicing is terminated. Idle server processes register themselves with NETACP so that they may be reused for another logical link.

NETSERVER is written in BLISS-32 code. The only porting change necessary was the addition of the BLISS VOLATILE attribute to several data declarations.

*NCP* The network control program (NCP) is the user interface for network management. NCP communicates with other network management components using the network information and control exchange (NICE) protocol. NCP can be used to manage the local node as well as remote nodes. When managing the local node, NCP exchanges NICE protocol messages with the NMLSHR shareable image. For remote management, NCP creates a logical link to the network management listener (NML) object on the remote node and exchanges NICE protocol messages over this logical link.

NCP consists primarily of BLISS-32 modules. The major porting issue associated with NCP was changing the code to use LIB$TABLE_PARSE rather than LIB$TPARSE.

*NMLSHR* NMLSHR is a shareable image that processes NICE protocol network management messages on an OpenVMS system. NMLSHR decodes NICE messages received as input and performs the requested management operation. NMLSHR builds NICE protocol messages as a response to requests asking for network management information to be returned. NCP and NML both link with the NMLSHR image to call the routines that process the NICE protocol messages.

NMLSHR is written in BLISS-32 and MACRO-32. Porting issues included dependencies on the format of a VAX argument list and changes required to link shareable images.

*NML* The network management listener (NML) image is run when a remote node requests a connection to the NML object to perform remote network management operations. NML sends NICE protocol messages to and receives them from the remote node. NML passes NICE protocol messages received from the remote node to NMLSHR for decoding and receives messages from NMLSHR to send to the remote node.

NML is written in BLISS-32 code. The only porting change made to NML code was to add the BLISS VOLATILE attribute to one data declaration.

*EVL* The event logger (EVL) receives event messages from the various DNA layers. EVL can also act as an event sink for messages generated at a remote node. EVL is started by NETACP and declares itself as a network object so that remote nodes can connect to the EVL object and send event messages. EVL can log events to a file in binary form or format the messages into something readable by a network manager.

EVL is written in BLISS-32 code. Porting issues included adding the BLISS VOLATILE attribute to some data structure definitions and aligning data structure fields on natural boundaries.

*DTS and DTR* The DECnet test sender (DTS) and the DECnet test receiver (DTR) are cooperating programs that can be used to test the network connection between two nodes. DTS runs on the local node and communicates with DTR on the remote node. DTS and DTR can be used to test the throughput and reliability of a line over which DECnet is running.

DTS and DTR are written primarily in MACRO-32 code. The two major porting issues associated with DTS and DTR were changing the code to use LIB$TABLE_PARSE rather than LIB$TPARSE and adding some BLISS-32 code to support floating-point operations.

*RTPAD* RTPAD provides the connection between a local terminal and the remote terminal services of a remote node. RTPAD is invoked as the result of executing the SET HOST command of the Digital Command Language (DCL). RTPAD communicates with REMACP and CTDRIVER or RTTDRIVER on the remote system to provide remote terminal support. RTPAD accepts input from the local terminal (which could be another remote terminal) and sends this data over the network to the remote node. Output from the remote node is received by RTPAD and displayed on the local terminal.

RTPAD is written in MACRO-32 code. Porting issues included unaligned references and aligning data structure fields on natural boundaries.

*NICONFIG* NICONFIG is the Ethernet configurator that listens to the MOP system identification messages broadcast on Ethernet circuits and maintains a database of configuration information for all systems heard. NCP is used to manage and display the information maintained by NICONFIG. NICONFIG runs as a process created by NMLSHR and communicates with NMLSHR over a DECnet logical link using the NICE protocol.

NICONFIG is written in BLISS-32 code. The only porting change was to remove the module switch LANGUAGE.

*HLD* The host loader (HLD) communicates with the DECnet-RSX satellite loader to downline load

tasks to an RSX-11S node. HLD is written in MACRO-32 code. The only porting change was to update the structure definition language used to create one data structure.

*MIRROR* The loopback mirror participates in network services protocol and routing layer loopback testing. MIRROR is written in MACRO-32 code. No porting changes were required though changes were made to the link procedure.

## DECnet-VAX Port to the OpenVMS AXP Operating System

This section discusses the development environment, process, and issues related to porting the DECnet-VAX product to the OpenVMS operating system.

### DECnet for OpenVMS AXP Development Environment

DECnet for OpenVMS AXP is built with and integrated into the OpenVMS AXP operating system. Many changes were being made to system data structures that directly affected the DECnet software. These changes required the DECnet for OpenVMS AXP software to be built with and tested on many interim operating system base levels before the combined OpenVMS AXP operating system and DECnet for OpenVMS AXP kit was shipped for layered product development.

Because the development tools changed throughout the project, we used the same tools to port the DECnet-VAX software as were used to develop the operating system base levels. When we copied portions of an OpenVMS AXP base level, we also copied the tool directories associated with the system build. We used cross compilers for MACRO-32 and BLISS-32, which allowed us to develop Alpha AXP software on an OpenVMS VAX system.[3] In addition, we used the OpenVMS AXP linker, librarian, and system dump analyzer (SDA) cross tools on the VAX system.[4,5] We also used the OpenVMS AXP debugging tools Delta and XDelta on the Alpha AXP prototype hardware.[6]

Initial DECnet for OpenVMS AXP testing was accomplished on a VAX system. Such testing was possible because we designed a majority of the DECnet for OpenVMS AXP code to run on both VAX and Alpha AXP hardware platforms.

The Alpha AXP prototype system used for testing utilized a shared disk that contained the OpenVMS AXP operating system images. The images and test procedures were copied onto the disk from a AXP system. Each time new DECnet for OpenVMS AXP images or test procedures had to be added to the shared disk during a test or debug session, the Alpha AXP test system had to be stopped, the disk mounted on the VAX system, images copied, the disk dismounted, and the Alpha AXP system rebooted. Providing file transfer support by means of the DECnet for OpenVMS AXP software early in the Alpha AXP project provided increased productivity for anyone testing on Alpha AXP prototype systems.

### Porting Process

The process of porting the DECnet software from the VAX hardware platform to the Alpha AXP platform consisted of the following steps: code preparation, compilation, linking, code review, debug, and testing. We did not start the task of porting DECnet-VAX with a completely clear vision of the total process. As we progressed and learned more about the tools and porting process, we improved our porting techniques and, as a result, our productivity.

Our strategy was to begin by porting the drivers and privileged code. These components were the most complex; they were written completely in MACRO-32 code and had the greatest potential for change. We started with NETDRIVER and NETACP, assigning one engineer to work on each component. As our porting group grew in number, we began to port, in parallel, the BLISS modules that comprise NCP, NML, NMLSHR, EVL, and MOM.

The following is an overview of the process we used to port the DECnet-VAX software to the Alpha AXP platform. Later sections contain details of coding practices that had to change.

*Code Preparation* Our first task was to create procedures that we could use early in the porting process to compile single modules of a DECnet for OpenVMS AXP component. We also ported the component's build procedure to use the new Alpha AXP cross tools.

Next, we began to prepare the code for initial compilation. MACRO-32 code must have each entry point identified prior to the initial compile. Entry points are identified by a compiler directive such as .JSB_ENTRY and .CALL_ENTRY. Each directive accepts optional parameters that identify register usage. However, this information is not required at this point in the porting process. The Alpha AXP MACRO-32 compiler will provide register

usage hints during the compilation, if so directed. As the team became familiar with the porting process, we were able to combine these steps and include the register usage information when declaring entry points. Also, as our experience increased, we were able to make changes to nonportable coding practices prior to the initial compile of a module.

Our experience proved, as we expected, that BLISS code is far easier to port than MACRO-32 code. For the DECnet-VAX components containing BLISS modules, we began the port by running the component's build procedure. BLISS routines do not require that entry points be identified. The compiler can process each module, identify errors, and provide warning and informational messages.

*Compile Process* After we completed the initial code preparation and created customized build procedures, the real iterative process of porting began. At this point we compiled one or more modules, made additional modifications based on the compilation results, and recompiled until we were reasonably satisfied that all the errors were fixed.

The Alpha AXP cross compilers, the MACRO-32 compiler in particular, have the capability of providing a vast array of informational and warning messages. When compiling a module, we always requested all informational messages. The information assisted us in identifying the input and output registers as well as the registers that the compiler automatically preserved. Using this information, we verified the register usage in each routine and added the information to the entry-point directives. Other informational and warning messages directed us to coding techniques that required change. By working with one module at a time, we avoided making repetitive porting errors in multiple modules prior to our complete understanding of how to solve the more complex porting problems.

Some informational messages caution that certain coding techniques such as data alignment should be modified. We observed that attempting to make changes to align all data structure elements prior to completing preliminary debug and testing caused many debug problems. Therefore, we decided to establish a porting policy to change only as much code as was absolutely necessary prior to the initial debug and test of a DECnet for OpenVMS AXP software component. Adhering to this policy required careful consideration, since some atomicity and granularity problems that are not resolved/addressed might cause code failures during debug.[3]

NETDRIVER and NETACP contained architecture-specific code, including memory management, driver tables, and structure definitions, which had to be made conditional for the OpenVMS AXP and OpenVMS VAX systems. A prefix file was added to each MACRO-32 module during the Alpha AXP compilation step. This file contained an Alpha AXP declaration that allowed us to include the directives required for conditional compilation. To compile the ported code on a VAX system, it was necessary to provide a VAX declaration and macros for the various entry-point directives that when expanded contained no instructions. These were placed in a common library file and conditionally compiled. The library file is included in each module. Figure 2 is an example of a library file that contains a VAX declaration and macros.

BLISS architecture-specific code was made conditional using the %if %bliss(bliss32v) or %if %bliss(bliss32e) constructs for OpenVMS VAX and OpenVMS AXP, respectively.

After porting all the modules within a component, the component's build procedure was run to ensure that each module had been ported without error. This was typically the first attempt to link the component. We also ran the OpenVMS VAX procedure to ensure that the code would continue to compile and link on the OpenVMS VAX operating system.

*Linking* The process of linking was difficult at times. The DECnet for OpenVMS AXP software contains drivers, system images, and shareable images. Each component required changes to the link procedures. We made these procedures conditional for both the OpenVMS VAX and the OpenVMS AXP operating systems.

The process of linking the ported modules brought to light many unresolved references. In general, these references were to external routines that had changed for the OpenVMS AXP operating system. One of the most difficult aspects of the porting project was determining which changes to the OpenVMS operating system had an impact on our project. Determining these changes was difficult because DECnet for OpenVMS AXP is tightly integrated into the OpenVMS AXP operating system. The process of porting applications to the OpenVMS AXP environment should not be as difficult.

```
        .SUBTITLE $DECNETDEF
;
;       Define all those symbols that should precede all DECnet
;       macro modules.
;
.MACRO  $DECNETDEF
.IF NOT_DEFINED Alpha_AXP
;
;       These make Alpha AXP code compile on VAX builds by doing
;       nothing when encountered
;
        VAX=1
        ; .JSB_ENTRY
                .macro .jsb_entry, input, output, scratch, preserve
                .endm
        ; .JSB32_ENTRY
                .macro .jsb32_entry, scratch, preserve
                .endm
        ; .CALL_ENTRY
                .macro .call_entry, preserve, max_args=0,-
                       home_args=false, input, output, scratch
                .endm
.ENDC
.ENDM
/
```

*Figure 2 Library File That Contains a VAX Declaration and Macros*

*Code Review* When all the known porting problems found during the compile and link phases had been corrected, we began our code review process. The original VAX code, the ported code, and a difference listing were available to the porting team. One or more members of the team reviewed the changes made and pointed out any problems that were identified to the person responsible for the module being reviewed. We all had previously agreed that the reviews would be friendly and that egos would be left out of the process. We found that our successful code reviews were well worth the effort.

Initial reviews turned up differing philosophies regarding the porting process. We discussed these differences and reached a consensus. The reviews uncovered errors in the porting process, and correcting these problems reduced the amount of debugging required. The review process also allowed us to agree on and maintain coding standards.

*Debugging* Our approach to debugging the DECnet for OpenVMS AXP software was to build the common ported code for a VAX system and to replace the OpenVMS VAX images with our ported version on one of our workstations. We began by loading the ported NETDRIVER and NETACP components. Since many of the required changes were common to both OpenVMS AXP and OpenVMS VAX systems, we were able to debug much of this code before we had access to Alpha AXP hardware. We found and fixed a number of problems using this technique.

When we were reasonably confident that the ported code was working on the OpenVMS VAX operating system, we began testing on Alpha AXP prototype hardware, which fortunately had just become available. We completed the driver load and ACP initialization testing. The initial test uncovered some problems that required special workarounds to allow debug to continue. These problems were corrected in later versions of the tools. Since the user interface had not yet been ported, test code was written to start DECnet for OpenVMS AXP and begin debugging the $QIO interface to the driver.

Eventually NCP, NML, and NMLSHR were ported, and more comprehensive debugging began. We used the OpenVMS AXP XDelta and Delta tools to debug the DECnet for OpenVMS AXP code on our Alpha AXP prototype hardware. System failures were debugged using the SDA cross tool on a VAX system. We learned how to trace call chains by

studying the OpenVMS calling standard.[7] Understanding the format of linkage pairs, procedure descriptors, and register save areas made debugging much easier prior to the availability of these features in SDA. Debugging on an Alpha AXP system is more difficult than on a VAX system since most VAX instructions generate multiple Alpha AXP instructions whose positions are optimized by the compiler to take advantage of Alpha AXP architecture features. Thus, it is not always easy to follow the Alpha AXP code line by line because the generated Alpha AXP code from one language statement is interspersed with Alpha AXP code generated from another language statement.

*Testing* After solving the obvious problems during the debug process, we began to test the DECnet for OpenVMS AXP code. Early versions of the OpenVMS AXP file system, record management services (RMS), and the file access listener (FAL) were made available to us. We in turn provided the DECnet for OpenVMS AXP code to the group porting OpenVMS RMS and FAL for their use in debugging. We were then able to run test scripts that used a variety of DCL commands to perform loops of remote copies, differences, and directory listings of remote files. DECnet network management scripts tested the network management interface. DTS and DTR were used to perform data transfer testing. Since the DECnet for OpenVMS AXP software was available early, it was used by other Alpha AXP porting groups on Alpha AXP prototype hardware in various locations. As the code stabilized, a timesharing system was set up, which provided the opportunity for more testing.

## Porting Issues

When we began porting the DECnet-VAX software to the Alpha AXP hardware platform, we found many coding conventions could not be used. Most of these coding practices are called out by the cross compilers, which significantly helped the porting effort.[3]

The following is a discussion of some problems we encountered while porting and how we solved them.

*Entry Points* Approximately four months into the project, the porting team determined that using the .JSB_ENTRY directive in NETDRIVER was going to make porting difficult. The difficulty was due to the complexity of the code and the fact that some code paths contained more than a dozen layers of subroutine calls. We concluded that the code, which had existed for a long time, already saved and restored the correct registers. We decided that trying to communicate the correct list of input, output, pass-through, and preserve registers to the compiler could be an impossible task, especially given our schedule. We investigated the possibility of using the .JSB32_ENTRY directive. This directive allows the specification of registers that must be preserved but does not take any input, output, or scratch parameters. The OpenVMS AXP MACRO-32 cross compiler will not automatically preserve any registers when this directive is used. A great deal of care must be taken when using this entry-point directive.

Our decision to use .JSB32_ENTRY to declare entry points led to an interesting problem with asynchronously executing code that could interrupt other threads of execution. The DECnet-VAX code that we ported used PUSHR and POPR instructions to save and restore registers that were modified by DECnet-VAX code interrupting another thread of execution. When adding the .JSB32_ENTRY directives, we specified a register preserve parameter only on external entry points, assuming that the remainder of the original DECnet-VAX code was saving the proper registers. The preserve parameter ensures that all 64 bits of the registers specified are saved at routine entry and restored at routine exit. The PUSHR and POPR instructions preserve only the low-order 32 bits of the specified registers. However, if DECnet-VAX code in a routine without the .JSB32_ENTRY preserve parameter interrupts another thread of execution that makes use of the upper 32 bits of a register, these upper 32 bits would not be properly restored when control returned to the interrupted thread. The solution was to specify the register preserve parameter on the .JSB32_ENTRY directives used to declare the entry points of routines in DECnet for OpenVMS AXP that are capable of interrupting other threads of execution.

Whenever we changed the input or output parameters to an internal subroutine, we also changed the name of that subroutine. This effort helped identify all the internal calls made to subroutines whose interface had changed.

*Coroutines* A feature of the VAX architecture used throughout the NETACP and NETDRIVER components is called a coroutine. Coroutines used in MACRO-32 allow a subroutine to call code fragments in other subroutines. This technique uses the

jump-to-subroutine construct JSB @(SP)+ to jump between coroutines. The code example shown in Figure 3 demonstrates the use of the JSB construct.

The general flow of the example is for MAIN to call COROUTINE with R0 equal to 0 and R1 equal to 1. Usually, COROUTINE changes the value of R1 to 2 and calls back MAIN at address SAVE. If COROUTINE is entered with R1 not equal to 1, then R0 is set to 1 and the coroutine dialogue terminates. MAIN at address SAVE then tests R0 and exits. Under normal circumstances, MAIN at address SAVE continues, storing the returned value of R1 in DATA and calling back the coroutine at address FINAL. COROUTINE at address FINAL then changes the value of R1 to 3, sets the return status in R0 to 1, and returns to MAIN at address TERMINATE. TERMINATE then exits MAIN via the RSB instruction.

All entry points in MACRO-32 code on an OpenVMS AXP operating system must have an entry directive. Thus, it is not possible to use the JSB construct to jump to any random line of code, as the previous example demonstrates. To do so, the code shown in Figure 3 would have to be split into subroutines, each with a .JSB_ENTRY or .JSB32_ENTRY entry directive. Also, we had to change the implementation of coroutines. Rather than use the stack to pass return addresses, we passed each return address in a register.

Since some coroutines ported were more complex than the example shown in Figure 3, we developed a technique to port VAX coroutines to the Alpha AXP architecture. When a coroutine is split into multiple routines, some code, such as that testing returned values, may change relative location. In our example, the error processing at SAVE is no longer necessary. Instead, COROUTINE returns to MAIN if it detects an error, and MAIN simply returns to its caller with the status in R0. The VAX code example in Figure 3 was converted to Alpha AXP code using our technique. The resulting code is shown in Figure 4.

The use of coroutines on Alpha AXP systems should be discouraged because of the overhead associated with storing the return address in registers and the additional consumption of stack space. Rather than a simple return address on the stack, there will be a register save area on the stack for each subroutine that makes up the coroutine. Recursive coroutines can consume large quantities of stack space. We have since converted coroutines used in main code paths to straight in-line subroutine calls.

*Stack Usage* MACRO-32 code uses a number of common coding techniques that require knowledge of the state of the stack and that must be changed for the OpenVMS AXP operating system. One such technique, referred to as an up-level stack reference, occurs whenever a subroutine attempts to access information (address or data) stored on the stack by its caller. Parameter passing sometimes uses this technique. If a routine pushes arguments

```
MAIN:       MOVL #0, R0             ; Assume failure
            MOVL #1, R1             ; Set initial value
            JSB  COROUTINE          ; Open a coroutine dialogue

SAVE:       BLBS R0, TERMINATE      ; No change in value
            MOVL R1, DATA           ; Save the changed value
            JSB  @(SP)+             ; Continue coroutine dialogue

TERMINATE:  RSB                     ; Exit with status in R0

COROUTINE:  CMPL R1, #1             ; Should we change the value?
            BNEQ EXIT               ; If not, exit routine
            MOVL #2, R1             ; Change the value
            JSB  @(SP)+             ; Call back to coroutine

FINAL:      MOVL #3, R1             ; Final value

EXIT:       MOVL #1, R0             ; Signal success
            RSB                     ; Return
```

*Figure 3 VAX Code Example Showing the Use of the Construct JSB @ (SP)+ to Jump between Coroutines*

```
MAIN:    .JSB_ENTRY       OUTPUT=<R0,R1>,-
                          SCRATCH=<R2>
         MOVL     #0, R0           ; Assume failure
         MOVL     #1, R1           ; Set initial value
         MOVAB    SAVE,R2          ; Next coroutine address
         BSBW     COROUTINE        ; Open a coroutine dialogue
         RSB                       ; Return to caller

COROUTINE: .JSB_ENTRY     INPUT=<R1,R2>,-
                          OUTPUT=<R0,R1,R2>
         CMPL     R1, #1           ; Should we change the value?
         BNEQ     EXIT             ; If not, exit routine
         PUSHL    R2               ; Save next coroutine address
         MOVL     #2, R1           ; Change the value
         MOVAB    FINAL,R2         ; Coroutine address for SAVE to use
         JSB      @(SP)+           ; Continue at SAVE

EXIT:    MOVL     #1, R0           ; Set status
         RSB                       ; Return to MAIN

SAVE:    .JSB_ENTRY       INPUT=<R1,R2>,-
                          OUTPUT=<R0,R1>
         PUSHL    R2               ; Save next coroutine address - FINAL
         MOVL     R1, DATA         ; Save the changed value
         JSB      @(SP)+           ; Continue coroutine dialogue at FINAL
         RSB                       ; To COROUTINE


FINAL:   .JSB_ENTRY       OUTPUT=<R0,R1>
         MOVL     #3, R1           ; Final value
         RSB                       ; To SAVE
```

*Figure 4 Alpha AXP Code Example Showing the Use of the Construct JSB @(SP)+ to Jump between Coroutines*

onto the stack prior to jumping to a subroutine, the called subroutine does up-level stack references to retrieve the arguments. Other techniques include using the stack as a common data area or attempting to manipulate the caller's return address in order to alter the program flow.

All these techniques require re-coding. When we encountered code that passed parameters on the stack, we modified the code to pass parameters in registers. If a structure was being passed, separate memory was allocated and the address of the structure passed in a register. In one case, NETACP used coroutines to perform specific functions to update a common data area allocated on the stack. This code was redesigned to eliminate the coroutines and up-level stack references. Another alternative would have been to pass the address of the data area on the stack to the called routine.

Altering the program flow when error conditions were encountered was also a common technique used in the DECnet-VAX MACRO-32 code. Subroutines removed the return address from the stack and returned to the caller's caller. We modified the code to remove the up-level stack reference (the caller's return address) and return a flag in a register to signal the caller that a change in the program flow was desired.

*Condition Codes* The Alpha AXP architecture does not support global condition codes in the processor status word. Some routines set condition codes and returned to the caller, which proceeded to perform a conditional branch on the results of the called routine. All occurrences of this technique were changed; routines now pass the result of any conditional test to the caller in a register.

*Granularity and Atomicity Issues*[8] The NETACP and NETDRIVER components access shared data structures. Since NETDRIVER can interrupt NETACP, the DECnet-VAX code relies on the atomicity of VAX

instructions to provide synchronized access to shared fields in the data structures. The DECnet-VAX code also relies on byte (8-bit) and word (16-bit) granularity for memory writes. Since the granularity of Alpha AXP memory writes is either longword (32-bit) or quadword (64-bit), DECnet-VAX code that required atomic access to word fields had to be modified to protect against writes to neighboring byte and word fields sharing the same longword or quadword. In DECnet for OpenVMS AXP, word data structure fields shared by NETACP and NETDRIVER that required atomic access were moved to their own aligned quadwords to prevent interference from simultaneous writes to other byte and word fields sharing the same quadword. After the word fields were placed in their own aligned quadwords, the code generated by the MACRO-32 cross compiler for the ADAWI instruction was sufficient to provide atomic access to the word fields. We could also have used compiler directives to specify that VAX granularity and atomicity rules be preserved.

*BLISS-32 Code* The BLISS-32 code in the DECnet-VAX software was relatively simple to port. We made minor changes to add the VOLATILE parameter to data items that should not be cached in registers, to conditionally compile the exception handlers for VAX or Alpha AXP, and to remove unsupported built-ins. Other modifications were more extensive, such as the changes to accommodate the new LIB$TABLE_PARSE.

*LIB$TPARSE Changes* LIB$TPARSE and LIB$TABLE_PARSE are the interface routines to a general-purpose, table-driven parser for the OpenVMS VAX and OpenVMS AXP operating systems, respectively. The call to these routines was made conditional for the VAX and Alpha AXP architectures. Other changes were required because LIB$TPARSE and LIB$TABLE_PARSE differ in the way argument lists are passed. The method used by LIB$TPARSE to pass arguments is incompatible with the OpenVMS AXP calling standard. The LIB$TPARSE action routines required modification as a result of the required change to LIB$TABLE_PARSE for the OpenVMS AXP operating system. The LIB$TPARSE action routines received all or a subset of the argument block as parameters. LIB$TABLE_PARSE passes the address of the argument block to the action routines. The solution we used was to make the routine declaration conditional so that on the OpenVMS VAX operating system the action routines continued to receive the argument block parameters, and on the OpenVMS AXP operating system the action routines received the address of the argument block. Next, for the OpenVMS AXP operating system, the parameter names used by the common code were bound to the argument block. These changes are shown in Figure 5.

As a result of this relatively simple though repetitive change, no other changes had to be made in the action routines. If at some future time the OpenVMS VAX operating system uses LIB$TABLE_PARSE, there will be no need for conditionals.

```
%IF %BLISS(BLISS32V) %THEN
GLOBAL ROUTINE ACT$INV_COMMAND (OPT,STRCNT,STRPTR,TKNCNT,TKNPTR,CHR,
                                NUM,PARAM) = !
%ELSE
GLOBAL ROUTINE ACT$INV_COMMAND (PARSE_STATE : REF $BBLOCK) = ! %FI

%IF %BLISS(BLISS32E) %THEN
     BIND
          OPT       = PARSE_STATE[TPA$L_OPTIONS],
          STRCNT    = PARSE_STATE[TPA$L_STRINGCNT],
          STRPTR    = PARSE_STATE[TPA$L_STRINGPTR],
          TKNCNT    = PARSE_STATE[TPA$L_TOKENCNT],
          TKNPTR    = PARSE_STATE[TPA$L_TOKENPTR],
          CHR       = PARSE_STATE[TPA$B_CHAR],
          NUM       = PARSE_STATE[TPA$L_NUMBER],
          PARAM     = PARSE_STATE[TPA$L_PARAM]
          ;
%FI
```

*Figure 5 LIB$TPARSE Changes*

## Conclusion

This porting effort not only provided a solid base of knowledge with which to begin the port of the DECnet/OSI for OpenVMS VAX software and the associated products, but also gave us an appreciation of common code and the avoidance of architecture-specific code.

More and more software is being ported to new hardware platforms. The porting process is often carried out by individuals who did not develop the original software and who may not even be familiar with it. Our experience porting the DECnet-VAX code leads us to believe that new software development should take into account the possibility that the code will be ported to new hardware platforms at some future date. As we continue to port the DECnet/OSI for OpenVMS VAX software, it is becoming increasingly obvious that certain coding practices are difficult to port. As a general suggestion, if the code has knowledge of the architecture but can be written using system routines, system services, or run-time library functions, write the code in that manner. These system routines will be ported with the operating system, and in a majority of the cases, the application code will not require modification.

Also, if architecture-specific functions are required, provide only a minimum amount of code to perform these required functions and segregate the code. Document how the code works, why it had to be done that way, what the alternatives were, and why they were not taken. In addition to helping maintain the code, this information may provide valuable assistance to a person porting the code in the future.

If a routine is written in assembly language for the sole purpose of performance improvement, consider rewriting it in a high-level language. It is possible that the assembly language coding conventions that may have been optimal for one hardware platform will be slower on a different hardware platform. Using high-level language compilers, which generate optimized hardware-specific code, will eliminate additional porting effort and may very likely be the best performance solution.

As we discovered during the process of porting the DECnet-VAX software, MACRO-32 code is significantly more difficult to port than code written in higher-level languages. However, certain architecture-specific functions may have to be written in assembly language. We recommend that these functions be isolated. In addition, we recommend that any other code written in MACRO-32 be rewritten, over time, in a higher-level language.

We determined that the fastest approach to porting was to make the minimum number of changes required to get the DECnet for OpenVMS AXP software running. The porting process was accomplished in phases. The first phase included the initial port and addressed any errors that occurred until we successfully completed linking the image. This phase also included the initial debug, which was first performed on VAX systems because of our common code approach and, subsequently, done on Alpha AXP prototype hardware. When the product was stable, we proceeded to the second phase in which we began to methodically align data structures and fix granularity and atomicity problems. Small changes could then be made and tested, and any new problems were generally easy to identify.

Our team approach to the project worked extremely well. Each team member was initially responsible for porting specific portions of the code. As the project progressed, individuals worked on any part of the product that needed attention. This flexibility was also used when we began to debug the ported code and again when we began to respond to problem reports. Priorities were used to assign resources in order to solve problems as quickly as possible. Throughout the project, team members worked together to share knowledge and to solve problems efficiently. This effective teamwork allowed us to deliver the DECnet for OpenVMS AXP product ahead of schedule.

## Acknowledgments

The authors would like to thank the other members of the software development team, Ken Roberts, Cathy Wright, our manager John Heron, and the group engineering manager Morea Martocchio, whose hard work made this project a success. In addition, we would like to thank all the individuals of the Alpha AXP project who helped us along the way. In particular, we would like to recognize certain individuals for their important contributions to the success of this project: Paul Weiss, our porting consultant; Lenny Scubowitz, David Gagne, and Ben Thomas of the I/O team; Karen Noel and Mike Harvey of the executive group; and Steve Dipirro of the XDelta team.

The DECnet for OpenVMS AXP project was only part of the Alpha AXP team effort. We feel fortunate to have experienced the synergy that this team created.

## References


1. A. Lauck, D. Oran, and R. Perlman, "Digital Network Architecture Overview," *Digital Technical Journal,* vol. 1, no. 3 (September 1986): 10–24.

2. P. Beck and J. Krycka, "The DECnet-VAX Product—An Integrated Approach to Networking," *Digital Technical Journal,* vol. 1, no. 3 (September 1986): 88–99.

3. *Migrating to an OpenVMS Alpha System: Porting VAX MACRO Code* (Maynard: Digital Equipment Corporation, Order No. AA-PQYEA-TE, 1992).

4. *OpenVMS Linker Manual* (Maynard: Digital Equipment Corporation, Order No. AA-PQXYA-TK, 1992).

5. *OpenVMS Alpha System Dump Analyzer Utility Manual* (Maynard: Digital Equipment Corporation, Order No. AA-PQYCA-TE, 1992).

6. *OpenVMS Delta/XDelta Utility Manual* (Maynard: Digital Equipment Corporation, Order No. AA-PQYPA-TK, 1992).

7. *OpenVMS Calling Standard* (Maynard: Digital Equipment Corporation, Order No. AA-PQY2A-TK, 1992).

8. N. Kronenberg et al., "Porting OpenVMS from VAX to Alpha AXP," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 111–120.


## General References


*DECnet for OpenVMS Network Management Utilities* (Maynard: Digital Equipment Corporation, Order No. AA-PQYAA-TK, 1992).

*DECnet for OpenVMS Guide to Networking* (Maynard: Digital Equipment Corporation, Order No. AA-PQY8A-TK, 1992).

*DECnet for OpenVMS Networking Manual* (Maynard: Digital Equipment Corporation, Order No. AA-PQY9A-TK, 1992).

*Migrating to an OpenVMS Alpha System: Planning for Migration* (Maynard: Digital Equipment Corporation, Order No. AA-PQY7A-TE, 1992).

*George A. Darcy III*
*Ronald F. Brender*
*Stephen J. Morris*
*Michael V. Iles*

# *Using Simulation to Develop and Port Software*

*Among the tools developed to support Digital's Alpha AXP program were four software simulators. The Mannequin and ISP instruction set simulators were used to port the OpenVMS and OSF/1 operating systems to the Alpha AXP platform. The Alpha User-mode Debugging Environment (AUD) allowed Alpha AXP user-mode code to be debugged with support from the OpenVMS VAX run-time environment on VAX hardware. AUD was built from a combination of new and existing Digital software components. The Alpha User-mode Debugging Environment for Translated Images (AUDI) allowed translated images to be debugged on a simulator running on a VAX computer. With these debugging environments, user-mode applications and code components could be tested before Alpha AXP hardware and operating system software were available.*

Digital developed several software simulators to support its Alpha AXP program.[1] These tools enabled engineers to develop and port software for the 64-bit RISC Alpha AXP architecture concurrently with hardware development. The simulators were used for a variety of purposes including porting, testing, verification, and performance analysis. This paper discusses four Alpha AXP software simulators: Mannequin, ISP, AUD, and AUDI.

## *The Mannequin and ISP Simulators*

Two Alpha AXP instruction set simulators, Mannequin and ISP, were used to port operating systems to the Alpha AXP platform. The OpenVMS group used the Mannequin simulator to port the OpenVMS VAX system to the Alpha AXP platform. Likewise, the OSF/1 group used the ISP simulator in their port of the ULTRIX and OSF/1 operating systems to the Alpha AXP platform. Both simulators were also used for architectural and design verification, and for performance modeling.

The Mannequin simulator grew out of a simulator developed for an earlier RISC project at Digital. The ISP simulator was written anew by engineers closely associated with the Alpha AXP architecture.

The two development groups were requested to boot their respective operating systems on the simulators before attempting to boot on the Alpha Demonstration Unit (ADU), the Alpha AXP prototype hardware.[2] The simulators were so successful in tracking the Alpha AXP architecture and in rooting out software bugs that the OSF/1 group was able to boot the ULTRIX operating system on the hardware on the first attempt. The OpenVMS group had similar success and booted the OpenVMS AXP operating system in a few hours.

Note that the Alpha Demonstration Unit (ADU) is an Alpha AXP prototype hardware system and should not be confused with the Alpha User-mode Debugging Environment (AUD) or the Alpha User-mode Debugging Environment for Translated Images (AUDI), two software simulator facilities discussed later in the paper.

### *OpenVMS AXP Porting*

The OpenVMS group used Mannequin as their Alpha AXP instruction simulator in porting the OpenVMS VAX operating system to the Alpha AXP hardware. Never before had an OpenVMS porting effort been able to debug as much operating system code in advance of hardware. Prior porting efforts debugged only up to VMB, a primary boot stage in the OpenVMS operating system. Using Mannequin, operating system developers were able to boot the entire operating system on the simulator and actually log in and debug utilities.

Some developers used Mannequin's own windows interface and debugging facilities to debug their code. Others ran the XDelta utility on top of Mannequin.[3] XDelta is a low-level system debugger

used to debug the OpenVMS VAX kernel and drivers. However, the Mannequin interface was useful in initially debugging XDelta, since the Alpha AXP console allows neither breakpoints nor single stepping.

To debug their code before the full OpenVMS AXP operating system was available, other developers used Mannequin in conjunction with the Alpha primary boot (APB) code and a test harness. Mannequin was especially useful in finding alignment faults in the boot sequence, since the alignment tools are not operational until the OpenVMS AXP system is completely booted. Alignment faults occur when an attempt is made to access a unit of data located at an address that is not a multiple of the size of the data.

### *Microcode Speedup*

One main reason the OpenVMS team was able to debug a large part of the operating system in real time was the use of specially written microcode to speed up the simulator. Mannequin is capable of running with special user-written microcode on the VAX 8800 family of machines.[4] This microcode is an addition to the normal VAX microcode for the 8800 machines; the VAX microcode remains unchanged. With microcode support, a large subset of Alpha AXP instructions is executed in microcode and attains performance comparable to native VAX instructions. The Mannequin microcode occupies 93 percent of the total 1,024 words of the user-writable control store.

Using microcode assistance greatly speeds up Mannequin execution, yielding from 350 thousand Alpha AXP instructions per CPU second (KIPS) to a peak performance of 1 million Alpha AXP instructions per CPU second (MIPS) on a VAX 8800. Without microcode assistance, Mannequin performance is on the order of 10 KIPS. (For comparison, the ISP simulator operates at approximately 30 KIPS.) Code streams that execute completely in Mannequin microcode show much better performance than those that switch back and forth between microcode and the software simulator. With microcode assistance on an unloaded VAX 8800, it takes from 20 to 30 minutes to boot the OpenVMS AXP system and reach the Digital Command Language (DCL) prompt after login. Because of this microcode speedup, software engineers were able to simulate and debug a much larger part of the operating system and utilities than ever before.

### *OSF/1 AXP Porting*

The OSF/1 operating system group used the ISP simulator as an Alpha AXP instruction compute engine. The strategy was to connect the ISP simulator to dbx, a standard UNIX source-level debugger, via dbx's remote interface. An interface was added to the ISP to support the following low-level debugger commands:

- Instruction stream examine and deposit
- Data stream examine and deposit
- Register examine and deposit
- Single step
- Continue
- Boot

The dbx debugger was modified to work with the 64-bit Alpha AXP architecture. That is, addresses in the debugger were extended to 64 bits, and an Alpha AXP disassembler was provided. The ISP simulator and dbx debugger operated as separate processes communicating on the same machine by means of a socket. A socket is a protocol-independent connection point for interprocess communications.

Historically, the OSF/1 group used the ISP-dbx combination to port the ULTRIX operating system to the Alpha AXP platform as an advanced development effort. When the group began to port the OSF/1 system, Alpha AXP prototype hardware (ADUs) and field-test compilers were available. Thus, the OSF/1 group used the ISP in its ADU mode, where the ISP simulator operated as a console to the ADU hardware system. The ADU consists of an Alpha AXP DECchip 21064 processor, memory, disks, Ethernet, and a DECstation 5000 workstation, which acted as the console interface. Instructions that normally execute on the simulator were transferred to the ADU for execution. However, the entire symbolic debugging environment remained unchanged.

### *Simulator Specifics*

The ISP simulator was written entirely in portable C. The Mannequin simulator was a hybrid of the C++ and C languages. ISP consisted of approximately 25,000 lines of code, Mannequin 31,800 lines. The two simulators shared common code: the ISP simulator provided Mannequin with floating-point routines and a comprehensive instruction

test program; Mannequin provided ISP with I/O device routines. Thus, the simulators verified the Alpha AXP architecture as well as each other.

The Mannequin and ISP simulators tracked and supported changes in the evolving Alpha AXP architecture and in PALcode. PALcode is special machine-dependent software that provides support for many low-level operating system services such as faults and exceptions. PALcode also provides instructions not in the base Alpha AXP hardware.

The two simulators have features common to many simulators, including support for loading and running programs, setting breakpoints and watchpoints, accessing memory, and saving and restoring machine state. Also supported are many machine-specific features, such as I/O devices, interval timers, and configurable translation lookaside buffers. Besides a command line interface, the Mannequin simulator has a graphical windows interface that allowed users to see most machine resources in a windows-based format, as shown in Figure 1.

The Mannequin and ISP simulators support three basic devices:

- A console device used for terminal I/O
- A disk device used to boot the operating system
- An interval timer used for interrupts

The disk device on the simulators can be either a file or a physical disk device. The OpenVMS group used a shared disk so that developers could boot from a common disk while running on the simulator.

The simulators provide 16 megabytes (MB) of physical memory with a default page size of 8 kilobytes (kB). The physical memory of the simulators may be increased to the practical limit of available virtual memory on a VAX system (minus a small amount for the actual simulator code).

Both simulators have configurable instruction stream (I-stream) and data stream (D-stream) translation lookaside buffers (TLBs). A TLB is a small cache that holds recent virtual-to-physical address translation and protection information. The simulator TLBs can have a variable number of entries in each of the four granularity hint block sizes. Granularity hints indicate to the translation buffer implementations that a block of pages can be treated as a single, larger page. In essence, there are four minitranslation buffers. The ISP simulator supports selectable TLB replacement algorithms, whereas Mannequin supports only the not-last-used (NLU) algorithm. The configurable TLBs allowed the operating system and chip design groups to analyze and finely tune the translation lookaside buffers for optimum performance.

### *Performance Analysis and Benchmarking*

The Mannequin and ISP simulators also support execution of user-mode, stand-alone programs, i.e., those with little or no operating system run-time support, by providing program loaders for several formats. These formats include two UNIX object formats (COFF and a.out), an OpenVMS AXP image format, and a system (raw data) image format.

Programs were compiled with early field-test Alpha AXP compilers. Program execution was especially useful for hardware designers and compiler writers for performance analysis and benchmarking purposes. Note that applications requiring full operating system support used the AUD facility, described in a later section.

The simulators can generate trace files in a standard trace file format. This commonality enabled the two facilities to share the same trace analysis tools. The trace files generated by Mannequin and ISP were also used as input to the Alpha Performance Model, another simulator that generated detailed performance data.

EVILIST and ALPHA$REPORT were two tools frequently used to analyze trace files and generate statistics concerning machine resources used during program execution. The types of data generated by ALPHA$REPORT include the following:

- Instruction distribution by opcode, class, and format
- Instruction and floating-point register utilization summary
- Distribution of code block run lengths
- Opcode pair distribution by class
- Control/branch instruction flow summary

An actual trace analysis report generated by ALPHA$REPORT is shown in Figure 2. This example comes from a scaled version of FPPPP (one of the 14 benchmarks in the SPECfp92 floating-point test suite), with the constant NATOMS set equal to 2. Figure 2 displays a report listing instruction distribution by opcode.

Alpha AXP operating system developers and compiler writers relied heavily on the trace reports for

```
                EV1:STATUS                    Mannequin Alpha Simulator V3.10-0

   microcode  UNAVAILABLE     unaligned_traps  ON        cycle_count  0000000588
      update  0               arithmetic_traps  ON        run status  STOPPED
       trace  OFF                    pal_debug  ON               srm  4
                                      halt_pal  ON               dtb  EV3/FIX=(4,0,0,32)
                                                                 itb  /FIX=(0,0,0,8)
  trace file  NONE
 console log  NONE

                                                EV1:GPR

 R00 00000000 00000008   R08 00000000 00000000   R16 00000000 00000000   R24 00000000 00000000
 R01 FFFFFFFF FFFFE6DF   R09 00000000 00000000   R17 00000000 00000000   R25 00000000 00000000
 R02 00000000 00000200   R10 00000000 00000000   R18 00000000 00000000   R26 00000000 00000000
 R03 00000000 00000000   R11 00000000 00000000   R19 00000000 000501D8   R27 00000000 00000000
 R04 00000000 00000000   R12 00000000 00040000   R20 FFFFFFFF FFFFFFC0   R28 00000000 00000000
 R05 00000000 00000000   R13 00000000 00000090   R21 00000000 00000000   R29 00000000 00000002
 R06 00000000 00000000   R14 00000000 00FF2800   R22 80000000 00000000   R30 00000000 01000000
 R07 00000000 00000000   R15 00000000 00000000   R23 00000000 00000080   R31 00000000 00000000

                EV1:INST                                                EV1:MEMORY

    P00000000 000501C0: A2C70000 LDL      R22,0(R7)          V00000000 00000000: 203F0000 201F0D04
    P00000000 000501C4: 42C03017 ADDL     R22,#1,R23         V00000000 00000008: 205FE000 24210001
    P00000000 000501C8: B2E70000 STL      R23,0(R7)          V00000000 00000010: 209F0000 24420001
    P00000000 000501CC: 951EFFD8 STD      R8,FFD8(R30)       V00000000 00000018: 40201403 24840000
    P00000000 000501D0: A11EFFD8 LDL      R8,FFD8(R30)       V00000000 00000020: 00000000 6BE30000
    P00000000 000501D4: C3FFFFAC BR       1FFFAC             V00000000 00000028: 00000000 00000000
P->P00000000 000501D8: 45C07019 AND      R14,#3,R25         V00000000 00000030: 00000000 00000000
    P00000000 000501DC: 45C0710E BIC      R14,#3,R14         V00000000 00000038: 00000000 00000000
    P00000000 000501E0: 21CEFFD0 LDA      R14,FFD0(R14)      V00000000 00000040: 00000000 00000000
    P00000000 000501E4: B10E002C STL      R8,2C(R14)         V00000000 00000048: 00000000 00000000
    P00000000 000501E8: B0EE0028 STL      R7,28(R14)         V00000000 00000050: 00000000 00000000
    P00000000 000501EC: B0CE0024 STL      R6,24(R14)         V00000000 00000058: 00000000 00000000

EV1> LOAD SQUARE_ROOT
  %MQN: loaded vms file (EMT_00:[DARCY]SQUARE_ROOT.EXE;1)
  %MQN: 22 blocks; entry at 00000000 000501D8
EV1>
```

*Figure 1 Mannequin Simulator Windows*

```
ALPHA Instruction Statistics Report  6-MAY-1992
FPPPP -- Quantum chemistry calculation of a two-electron integral
derivative


  Instruction Distribution by Opcode
  (Ranked from highest to lowest)

  Instruction                                          Cumulative
Class  Mnemonic     Occurrence      Percent            Percent
6      LDT            2321155        25.41                 20
8      MULT           1732928        18.97                 40
8      ADDT           1433798        15.70                 60
6      STT             998446        10.93                 70
1      LDQ             544385         5.96
1      LDL             241142         2.64
1      STL             178828         1.96                 80
4      BIS             151120         1.65
3      ADDL            126321         1.38
8      SUBT             95045         1.04
```

*Figure 2 Mannequin/ISP Trace Output*

help in designing critical sections of code. For example, the register usage distribution report helped determine how many registers should be preserved by a call and how many should be scratch (usable by a called routine without being preserved).

## The AUD Facility

Whereas the Mannequin and ISP simulators were suitable for initial debugging of low-level software such as operating systems, direct use of these tools for user-mode applications, i.e., layered products, is a different matter. Porting and debugging Alpha AXP user-mode code is at best difficult without the full run-time support of an operating system. User-mode applications typically take advantage of a wide variety of run-time libraries, including compiled code support (such as the Fortran run-time library), mathematical routines, graphics I/O services, and database software (such as Rdb for OpenVMS). Even if all this software were immediately available for Alpha AXP systems, running it under simulation would be prohibitively slow.

Therefore, Digital developed a mixed-execution debugging environment. This Alpha User-mode Debugging Environment (AUD) was built from a combination of new and existing Digital software components. In the AUD environment, user-mode code being developed for or ported to the Alpha AXP platform could be compiled and executed as Alpha AXP code using simulation on VAX hardware. At the same time, OpenVMS VAX run-time services called by the code could be executed as native VAX instructions. Thus, modules could be ported and debugged one at a time, until almost the entire application consisted of bug-free Alpha AXP code.

During the design of the AUD environment, two key technical issues were

- How to efficiently detect calls made by executing VAX code to a routine in Alpha AXP code that could be "executed" only by simulation, and conversely, how to detect calls made by Alpha AXP code being simulated to native VAX code.
- How to effect the transformation of parameters, both location and representation, from that provided by the caller in one domain into the locations and representations expected by the called routine in the other domain. Although there existed well-defined and widely followed calling standards for both domains, a variety of special-purpose, high-performance calling conventions were used in many situations.

This mixed-execution environment was expected to have a relatively short lifetime, because it would become obsolete as soon as significant numbers of real Alpha AXP hardware systems became available. Consequently, AUD itself had to be simple and inexpensive enough to be created quickly and put into use. The development effort met this requirement.

The elapsed time from initial concept to first use was about eight months; the total development effort for AUD over its lifetime was between three and four man-years.

## *AUD Components*

Despite the desire for simplicity, AUD consists of a number of cooperating components:

- Callable Mannequin Alpha Simulator
- AUD debugger
- AUD linker
- Alpha AXP native services
- VAX jacketing services
- AUD Linkage Analyzer (ALA)
- Selected VAX jackets

*Callable Mannequin Alpha Simulator* Callable Mannequin, the Alpha AXP instruction set simulator, is essentially a subset of the Mannequin simulator described earlier. In particular, Callable Mannequin omits the user interface and Alpha AXP machine state. Instead, the AUD debugger supplies the user interface. Also, storage for the Alpha AXP machine state is separately linked into the AUD environment to make this information globally accessible. Callable Mannequin does retain the microcode-assist feature.

*AUD Debugger* The AUD debugger is a modified version of DEBUG-32, the user-mode debug utility on the OpenVMS VAX operating system. The AUD debugger provides most of the same features of DEBUG-32. A configuration option allows the DEBUG-32 utility to use an internal, low-level remote debugger interface to interface with a foreign target. (This capability was originally developed for use in other products such as VAXELN Ada.) We developed new code to join DEBUG-32 and Mannequin using this interface. As a result, the AUD debugger works directly with VAX code, in the usual manner, and works with Alpha AXP code by passing commands to the Callable Mannequin simulator to set breakpoints, examine instructions, execute code, etc.

*AUD Linker* The AUD linker is a variant of the Alpha AXP cross linker that reads Alpha AXP object modules as input and produces an OpenVMS VAX format image as output. The standard VAX linker can therefore reference locations in the Alpha AXP image in the normal way, and the standard OpenVMS image activator can be used to load the Alpha AXP image for execution. However, to minimize complexity, we did constrain the Alpha AXP image to be linked as an absolute image (i.e., a based image, in OpenVMS jargon). This restriction eliminated the problem of how to relocate Alpha AXP instructions using the OpenVMS image activator. As mentioned previously, the Alpha AXP image also includes a global storage area to hold the simulated Alpha AXP machine state.

*Alpha AXP Native Services* Alpha AXP native services is a special operating system shell, part of which executes as Alpha AXP code (under simulation) and part of which is included in the AUD jacketing services. The native services provide the lowest-level support for hardware exception handling and the OpenVMS condition-handling facility. While AUD ultimately supported frame-based condition handling within the Alpha AXP image, interoperation of application exceptions between the Alpha AXP and VAX domains was not supported.

*VAX Jacketing Services* VAX jacketing services is VAX code that supports the ability to write jackets that pass control back and forth between VAX and Alpha AXP code. The mechanics for accomplishing this are discussed in the Jacketing section.

*AUD Linkage Analyzer* The ALA is a specialized compiler that reads a specialized jacket description language. This language describes how calls in one domain are to be transformed into calls in the other domain on a routine-by-routine, parameter-by-parameter basis. The output from the ALA is an Alpha AXP object module and a linker options control file, both used to link the Alpha AXP image, and a VAX object module. The Alpha AXP object module provides a transfer vector into the Alpha AXP procedures. The linker options control file provides symbol definitions in an encoded form to manage calls from the Alpha AXP image to the main VAX image, which is linked later. The VAX object module contains a table that encodes the jacketing description.

*Selected VAX Jackets* Selected VAX jackets are ALA jacketing files (in both source and compiled forms) for calling common VAX facilities from Alpha AXP

code. Jackets are provided for OpenVMS system services, the C run-time library, and some parts of the general-purpose, run-time library (LIBRTL). The DECwindows group also supplied jacket definition files for use by other groups. AUD users are able to supplement these files as needed by creating and compiling their own jacketing descriptions for other VAX facilities.

Figure 3 shows the main steps in building an AUD environment. The uppermost sequence shows the compilation and linking of the Alpha AXP components, which results in the creation of the Alpha AXP image. The central sequence shows the compilation of the jacket descriptions, which results in the creation of components that are included in both the Alpha AXP and the VAX images. The lower rows of Figure 3 show the compilation of the VAX part of an application and its linking with the AUD manager to create the VAX main image. When the mixed VAX and Alpha AXP application is executed, these images are combined in memory with Callable Mannequin, the AUD debugger, and other shareable images. This relationship is illustrated in Figure 4.

## *Jacketing*

Jacketing is the key feature that allows VAX and Alpha AXP interoperability, i.e., gives a processor the appearance of being able to execute both VAX and Alpha AXP instructions. Although the details of jacketing are intricate, the result is simple and elegant. Calls can be made freely back and forth between VAX compiled code and Alpha AXP compiled code, without any special compilation modes

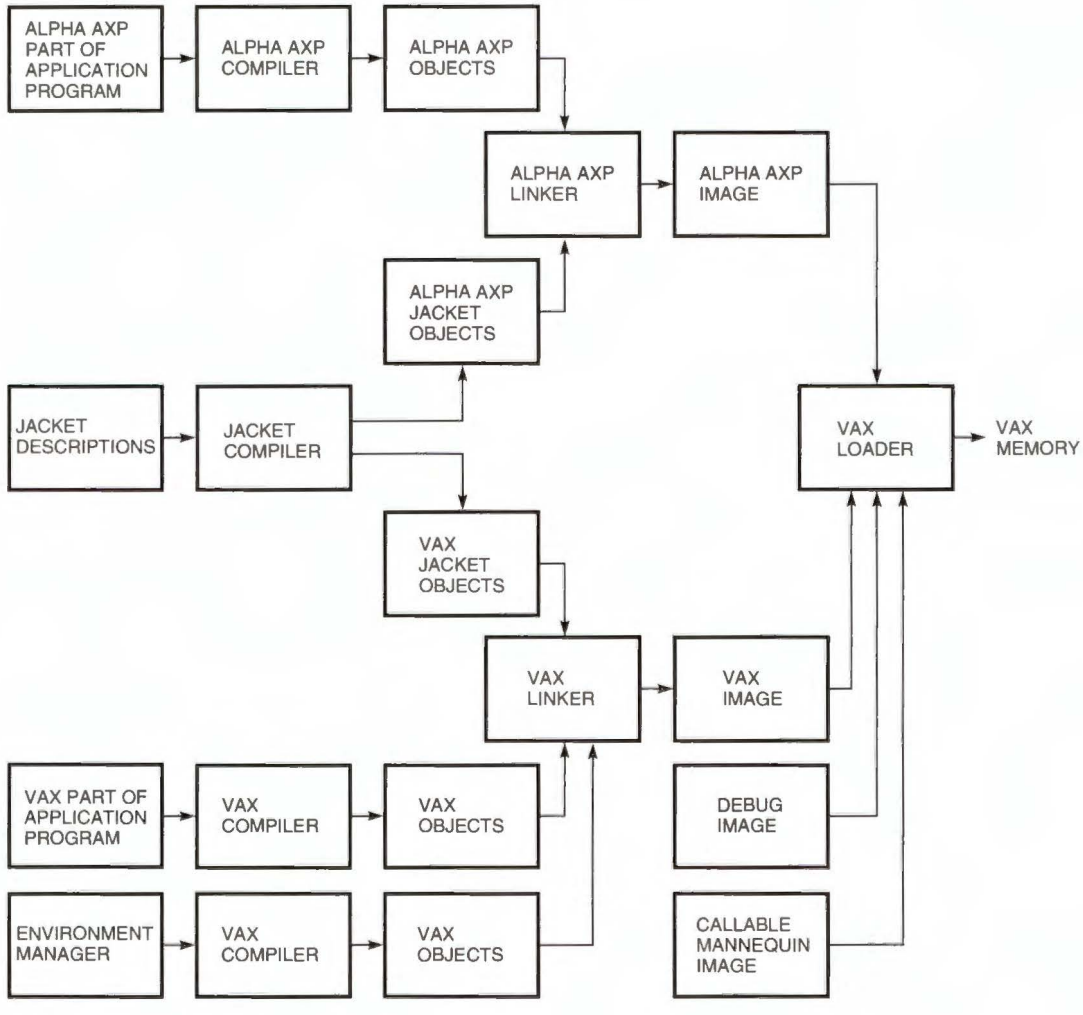



*Figure 3 Creating an AUD Application*

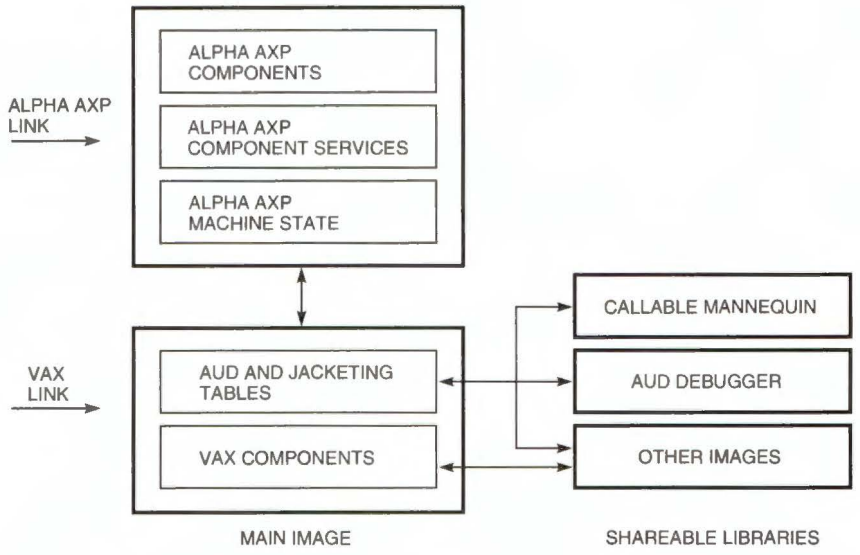



*Figure 4 AUD Process Components*

on either side. The AUD support is fully recursive and reentrant.

Static calls from VAX to Alpha AXP code are directed to dummy entry points in the object module produced by the ALA compiler. Each entry point is simply an instruction that loads a pointer to the jacketing description table for the target Alpha AXP procedure, followed by a transfer into common jacket interpretation code.

Calls from Alpha AXP code to VAX code use the fact that the Callable Mannequin component stops and returns control to the AUD environment when it detects an instruction that transfers control out of the Alpha AXP image. In this case, the apparent address is an encoded integer (created by the ALA), whose high four bits make it look like an illegal address (in the VAX reserved S1 space) and whose remaining bits are a two-level index (i.e., 12 bits of facility code and 16 bits of offset) into the jacket description table for the target VAX procedure. This two-level scheme was chosen to allow jacket descriptions for different shared library facilities to be prepared and compiled independently. The facility code is a number normally already associated with that facility by software convention for other purposes.

When a routine is called using a dynamically determined address, such as an address given in a function variable, a property of the VAX and Alpha AXP architectures is exploited to determine dynamically whether the target routine is a VAX routine or an Alpha AXP routine. According to the VAX architecture, the first 16 bits of a routine comprise a mask that encodes the registers to be preserved as part of the call. Bits 12 and 13 of this mask are unused and required to be 0; if one of these bits is set at the time of a call, then a hardware exception results. According to the OpenVMS AXP software architecture, an Alpha AXP procedure address is actually the address of a procedure descriptor, which is a data structure and not the actual Alpha AXP code. By design, bits 12 and 13 of this data structure must be set to 1.

VAX execution of a VAX CALL instruction that attempts to transfer to an Alpha AXP procedure results in an exception. A special AUD exception handler intercepts the exception, determines if the illegal entry mask is caused by a reference into an Alpha AXP image, and if so, calls into the AUD jacketing routines to reformat the call frame. This mechanism also works for handling asynchronous system traps (ASTs) from the OpenVMS VAX operating system to Alpha AXP code.

For computed calls from Alpha AXP code, compiled code calls an Alpha AXP run-time library routine to perform the comparable bit 13 test (under simulation). If bit 13 of the target location is set to 1, then simulated execution continues and an Alpha-to-Alpha call is carried out. Otherwise, control transfers to a special VAX code entry point in AUD, which terminates simulation and performs jacketing back to the VAX target procedure.

### Basic Operation

To start executing a mixed application, the AUD environment first performs several initialization steps. In particular, AUD scans all the images loaded in process memory to identify the Alpha AXP image (only one was allowed and supported).

Some AUD options are set through the use of OpenVMS logical names, which are interrogated during image initialization. These options include

- Selecting Alpha AXP stack size
- Enabling delivery of ASTs to Alpha AXP routines
- Disabling the normal Alpha AXP stack consistency checks
- Disabling unaligned memory reference messages
- Enabling AUD initialization tracing
- Disabling integer overflow checking

Debugging combined VAX and Alpha AXP code under the AUD environment is similar to debugging normal VAX code under the DEBUG-32 OpenVMS debugger. Basically, if the address involved in a debug command is within an Alpha AXP image, then the debugger calls the Mannequin simulator to perform the command for the Alpha AXP code. Otherwise, the DEBUG-32 debugger itself performs the command for the VAX code, as usual. Alpha AXP machine state is kept in static global storage by Mannequin and thus is visible to the AUD debugger.

In the DEBUG symbol table (DST) representation, variables that are allocated in the Alpha AXP registers are described as being allocated in the corresponding global state locations. This "trick" allowed AUD to handle the 64 Alpha AXP registers using the VAX DST representation, which can encode only the 16 VAX registers.

Once simulation begins, Mannequin continues to simulate Alpha AXP instructions until it either detects an instruction that would transfer control outside of the Alpha AXP image, completes a single-step request, or detects an error condition. Upon returning to the AUD environment, Mannequin supplies status information that indicates the reason simulation ended.

For a transfer of control from Alpha AXP to VAX code, AUD must first determine whether the transfer is a return from Alpha AXP code as a result of a prior VAX call or a new call from Alpha AXP code to VAX code. AUD is fully reentrant, so AUD cannot make this determination from global state. If the target address is a distinguished address that AUD supplies when it sets up a VAX-to-Alpha call (i.e., an address in the reserved S1 part of the VAX address space), the address is interpreted as a return transfer. Otherwise, AUD initiates a new Alpha-to-VAX call.

For a return operation, the AUD code copies the return value or values from the Alpha AXP registers and passes them back to the VAX code. A VAX return instruction is then executed to resume execution of the calling VAX code.

For a call operation, the VAX code fetches the Alpha AXP parameters and builds a VAX argument list, which is then used to call the target VAX routine. When the VAX routine returns, the contents of the result registers are copied to the corresponding Alpha AXP machine state locations, and Mannequin is restarted to resume executing Alpha AXP code.

Despite some limitations (e.g., only one Alpha image and no exception handling across the VAX to Alpha AXP domains), AUD greatly aided the OpenVMS AXP porting effort. The simulator provided software groups with a pseudo-Alpha AXP environment in which to debug their Alpha AXP code, well before either Alpha AXP hardware or the OpenVMS AXP operating system was available. Many OpenVMS AXP groups successfully used AUD to facilitate their porting, including the Record Management Services (RMS), DECwindows, Forms Management System (FMS), various OpenVMS command utilities, text processing utility (TPU), DEBUG, and GEM compiler back-end groups.

## The AUDI Facility

The VAX Environment Software Translator (VEST) is an important part of the initial OpenVMS AXP offering.[5] VEST translates an OpenVMS VAX executable or shareable image into an OpenVMS AXP image that can then be executed with support on an OpenVMS AXP system. As for other user-mode layer software components, it was desirable to test VEST and images translated by VEST as early as possible in a simulation environment such as AUD. However, AUD could not be used directly to test translated images for two reasons:

- VEST directly creates an Alpha AXP image. In effect, VEST is a combined compiler and linker. Thus, the symbol mapping protocols used by AUD were extraneous, and the linking protocols had to be completely replaced.
- Actual execution of a translated image on an OpenVMS AXP system makes use of the

Translated Image Environment (TIE).[5] The TIE is a shareable library that contains support routines for translated images. In particular, TIE provides support for VAX complex instruction processing, VAX-to-Alpha address mapping, and OpenVMS VAX exception handling. Creating a VAX version of the TIE to use with AUD required intimate interfaces with the OpenVMS VAX operating system as well as compatibility with AUD.

Thus, the need to debug translated images led to the creation of the Alpha User-mode Debugging Environment for Translated Images (AUDI). Just as Callable Mannequin provided a key building block for AUD, AUD in turn provided a key building block for AUDI. Alpha AXP software teams and porting centers used AUDI to port both Digital and third-party translated applications prior to the arrival of Alpha AXP hardware. The porting process was as follows: a VAX application was translated to Alpha AXP code by means of the VEST translator; this code was then run on a VAX system using the AUDI simulator.

The AUDI process components shown in Figure 5 include the

- Callable Mannequin Alpha simulator
- AUD debugger
- VAX version of the TIE
- Translated VAX code (Alpha AXP code)

### *AUDI Environment*

Emulated VAX state in AUDI is maintained in a global context block. Emulated VAX registers R0 through R14 are used exactly as their VAX counterparts. The correspondence between a translated and equivalent VAX program counter (PC) is not directly available during execution, since translated code occupies different address space than the original VAX code. Thus, register R15 is used instead as an in-image index register.

The user-mode VAX stack is split into a VAX stack and an Alpha and emulated VAX stack. The VAX stack services both the AUDI environment and any VAX system services or run-time library routines that the translated image may call. The Alpha and emulated VAX stack services Alpha AXP and translated code.

Translated images contain calls to the TIE as necessary to simulate VAX complex instructions and procedure calls. Complex instruction routines are used to simulate VAX instructions that would otherwise expand into excessive Alpha AXP code. However, since AUDI is running on VAX hardware, complex instructions can be executed native on the VAX hardware.

To initialize the AUDI environment, the translated image calls an initialization routine in the TIE by means of an initialization program section (PSECT). This routine determines the address range of the Alpha AXP code and the location of the VAX-to-Alpha address mapping structure, saves the current Alpha AXP register state, and calls Mannequin to begin executing translated code at the appropriate entry point. Translated code uses the address mapping structure to find computed branch destinations on the fly. Callable Mannequin then executes translated code until it encounters some instruction that would transfer control out of translated code. The cause of this transfer would be either a TIE-based procedure or complex instruction call, or calls to native VAX routines.

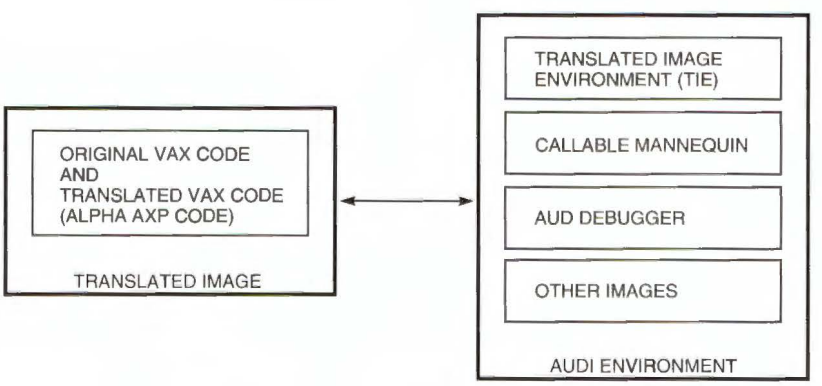



*Figure 5 AUDI Process Components*

Like AUD, AUDI allows interoperation between translated VAX code (Alpha AXP code) and VAX code. Translated code can use existing VAX system services and run-time libraries. AUDI does not use the jacketing language described in the section The AUD Facility. Instead, AUDI automatically jackets procedure calls between the translated VAX code and the native VAX code. Autojacketing includes building proper parameter lists and call frames for the destination calling standard.

The fact that AUDI does not use a jacketing language leads to some procedure call limitations. However, note that these limitations do not appear when running translated code on actual Alpha AXP hardware. For incoming calls (VAX code to translated VAX code), all AST delivery and condition handlers execute as VAX code rather than as translated VAX code. Thus, translated programs may not function properly. For outgoing calls (translated VAX code to VAX code), routines in which a callee modifies its caller's stack frame argument list or return address may produce unpredictable results, since the autojacketing may be altered or disconnected.

### *AUDI Example*

Figure 6 shows the execution of a translated image under AUDI. Note that both the BASIC image (HELLO_WORLD) and the BASIC run-time library (BASRTL) are translated. Run-time libraries that are used by the AUDI environment cannot be translated under AUDI. Translating run-time libraries that AUDI itself uses causes a "circularity in activation" and incorrect or no execution.

In the HELLO_WORLD example, there are 28 calls to VAX routines, most likely those to LIBRTL and

```
$ RUN HELLO_WORLD_TV
Hello World from VAX BASIC

AUDI V3.0 Runtime Statistics:

  8085 Alpha AXP instructions were executed.

                  TIE Lookups:      CALLx     JSB      JMP
  ========================================================
  Stayed in Alpha AXP routines:       4        5        0
          Went to VAX routines:      28        0        0
  --------------------------------------------------------
                         Total:      32        5        0

  28 VAX returns used (28 RET, 0 RSB) to resume Alpha AXP code.
  There were no Fault-On-Execute conditions converted to Lookups.
  21 CALLx Context Blocks were allocated - which were reused 7 times.

  There were 19 TIE-based 'complex instructions' executed.
    Instruction INSQUE   (0E)   : 2
    Instruction MOVC3    (28)   : 8
    Instruction MOVC5    (2C)   : 8
    Instruction MOVTUC   (2F)   : 1

  There was 1 VAX routine call to Alpha AXP code.

  There were 2 images containing Alpha AXP code:
    HELLO_WORLD_TV X0.0 from BL3.3 VEST of Mar 30 1992 09:27:02
    BASRTL_TV X0.0 from BL3.3 VEST of Mar 30 1992 09:14:10

  Execution depended on these images:
    LIBRTL_TV                DECW$XLIBSHR             LIBRTL2
    MTHRTL_TV                DECW$TRANSPORT_COMMON    LIBRTL
    TIE$SHARE                VAXCRTL                  DBGSSISHR
    MQN$SHARE                MTHRTL
    DECW$DWTLIBSHR           CONVSHR
    LBRSHR                   SORTSHR
```

*Figure 6 AUDI Example—Translated Hello World Image*

OpenVMS system services. There are 21 unique CALLx contexts and 7 reused ones. In addition, the example uses four different complex instructions.

## Summary

The software simulators Mannequin, ISP, AUD, and AUDI greatly aided Alpha AXP software porting and development efforts. Substantial parts of both system and application software were simulated and verified concurrently with hardware development. When Alpha AXP hardware became available, most software could be plugged in simply and ran exactly as expected. The use of these simulation tools saved a year or more from the overall Alpha AXP schedule.

## Acknowledgments

Many people throughout Digital contributed to the success of the Alpha AXP simulators. Homayoon Akhiani, Ray Lanza, Stephan Meier, Steve Morris, Andrew Payne, and Jon Reeves worked on the ISP model. George Darcy, Mark Herdeg, Kevin Koch, Eric Rasmussen, and Scott Robinson contributed to the Mannequin simulator. The AUD effort included several groups from across Digital. Their primary contributors were Walter Arbo, Ronald Brender, Henry Grieb, Mark Herdeg, Michael Iles, James Johnson, Robert Landau, Maurice Marks, Dennis Murphy, Scott Robinson, Larry Woodman, and James Wooldridge. Finally, much of the AUDI information in this article is taken from work originally done by Scott Robinson. Other AUDI contributors include George Darcy, Mark Herdeg, Matthew Kirk, Naghmeh Mirghafori, and Murari Srinivasan.

## References

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

2. C. Thacker, D. Conroy, and L. Stewart, "The Alpha Demonstration Unit: A High-performance Multiprocessor for Software and Chip Development," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 51–65.

3. *OpenVMS Delta/XDelta Utility Manual* (Maynard: Digital Equipment Corporation, Order No. AA-PQYPA-TK, 1992).

4. S. Mishra, "The VAX 8800 Microarchitecture," *Digital Technical Journal,* vol. 1, no. 4 (February 1987): 20–33.

5. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal,* vol. 4, no. 4 (1992, this issue): 137–152.

*Peter F. Conklin*

# *Enrollment Management, Managing the Alpha AXP Program*

*Digital's multiyear Alpha AXP program has involved more than two thousand engineers across many disciplines. Innovative management styles and techniques were required to deliver this high-quality program on an aggressive schedule. The Alpha AXP Program Office used a four-point methodology for management: (1) establish an appropriately large shared vision; (2) delegate completely and elicit specific commitments; (3) inspect rigorously, providing supportive feedback; (4) acknowledge every advance, learning as the program progresses. We consciously used each project event to propel progress and gain momentum. Digital delivered the Alpha AXP program on schedule with industry-leadership capabilities.*

## *Introduction*

The program to develop the Alpha AXP systems has been the largest in Digital's history and one of the largest in the computer industry. During the course of the program, the Alpha AXP Program Office developed a model that provided the tools necessary to manage the program. At times, this paper may seem to imply that the program team developed the tools and then used them in a pure form. In practice, the team developed these approaches based on many years of experience and on the management theories of experts; we also learned and applied these lessons as we managed the program.

Although the positive effects of timely delivery and high quality are particularly noticeable results of such a large program, Digital has also used the tools to good effect on smaller projects. Moreover, teams within the Alpha AXP program used the tools recursively, project by project. The author's experience is that this management model is applicable to projects of nearly any size.

The discussion that follows briefly defines the scope of the program and explains why traditional methods were inappropriate for managing the development of such a complex product set in a short time period. The Enrollment Management Model and the concept of cusps—a key element of the model—are then defined and clarified through discussion of the model's evolution during the Alpha AXP Program.

### *Size of the Alpha AXP Program*

Digital's Alpha AXP program encompassed the design of a world-leadership microprocessor chip, a new 64-bit system architecture, multiple hardware systems (from personal computers to mainframes), multiple operating systems, and hundreds of software products layered on these systems. The development of the first-generation products extended over several years and involved more than two thousand hardware, software, and systems engineers at its peak. Digital managed the overall development program from a Program Office staffed by eight professionals.

Across Digital worldwide, the Alpha AXP program development spanned more than 22 software engineering groups and 10 hardware engineering groups. The hardware effort included the semiconductor design group and groups for each of the hardware systems platforms. The software efforts encompassed four operating systems groups, and groups designing migration tools, network systems, compilers, databases, integration frameworks, and applications. Some groups peaked at more than 150 development engineers plus supporting staff. Many also contracted with suppliers both within and outside Digital.

### *Inappropriate Organizational Approaches*

Implementing such a broad, complex program presented not only technological challenges but a management challenge as well. The Program Office therefore considered and rejected a number of traditional organizational approaches.[1]

In the classic organizational model, a hierarchical, or line, organization is formed, containing all the primary implementers. The problem with this approach to large programs is that it takes too long to form the organization. Staffing the teams and establishing operational procedures take longer than the market window and available technology allow. The result is grand visions and projects delivered years behind schedule. Further, "temporary" organizations must be folded back into the mainstream at the end of the program. The management goal of the Alpha AXP program was to keep expertise concentrated to achieve synergy across many projects within a particular discipline.[2]

An alternative approach is to form small entrepreneurial teams and challenge them to work long hours to achieve the goals. This works well in small projects suitable for "skunk works." The original design work was conducted in this fashion. However, when this approach is applied to large programs, the result is that team members burn out without achieving the aggressive schedules demanded. Management then becomes frustrated and tries again with different teams, but the results are no better.

The Program Office established the Alpha AXP program as an integration of project teams that would remain within the existing line organizations. Thus, for example, each hardware and software project resided within its functional group (semiconductors, servers, OpenVMS, UNIX, compilers, database, CPU development, networks, etc.). The Program Office integrated the work of the individual project teams, which provided the additional advantage of program resilience in the face of functional group reorganizations.

## *The Enrollment Management Model*

The Enrollment Management Model (Figure 1) for the Alpha AXP program comprises four stages.

Vision–Enrollment

Commitment–Delegation

Inspection–Support

Acknowledgment–Learning

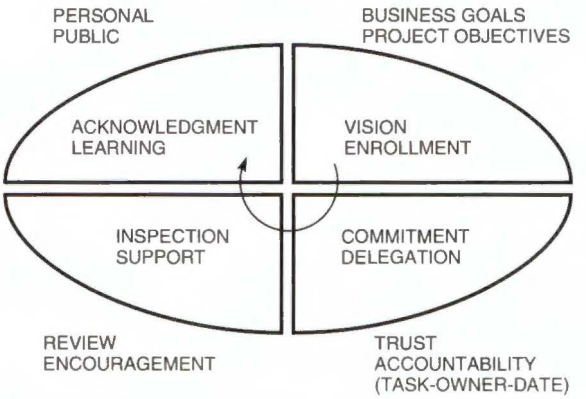



*Figure 1 Enrollment Management Model*

The model in this form was developed by the author. Some elements are derived from management seminars and from consultants' recommendations. The particular forms used for vision, commitment, and acknowledgment emerged during the Alpha AXP program. The inspection-support stage was developed by the author during many years of project management and reviews.

Two concepts are key to implementing this model for large programs. First, the Program Office, which has already been mentioned, provides the necessary cohesion, program vision, and inspection structures, while allowing the skills and resources to remain within their natural organizations. Moreover, the office lends consistency across the program and encourages each contributing group to hold to its commitments. The small Alpha AXP Program Office, made up of a diverse group of product and operations managers, had no formal authority (not even budget authority); so it exerted influence only through rigorous enrollment and delegation outlined by the management model.

The second key concept is the project "cusp," which is a critical event that propels change. Cusps are further defined in the sections Inspection–Support and Using Project Cusps below.

### *Vision–Enrollment*

The Program Office uses vision to enroll the related groups in the goals of the program. For example, the vision can be the top-level business goals and customer needs. For subordinate projects, the vision can be the objectives of the larger project. In all cases, the enrollment happens only when the goals are set in the context of the audience (the project team). In particular, the Program Office is most effective when it expresses the program's

vision in the terms and language of the group being enrolled. The vision has to be large enough to encompass all the required commitments and the ultimate results.

### Commitment-Delegation

As the manager of a project develops plans, he or she delegates the tasks to sub-groups and solicits specific commitments to content and schedule.[3] Since these commitments are made within the context of the larger vision, the subordinate commitments become quite strong for sub-project members. A key element of the delegation process is the explicit specification of the results such that they are measurable and identified with an individual owner. The owner is a single individual empowered by the committing group and held accountable for the deliverable.[4] An important point here is that the term "owner" does not necessarily refer to the person who actually does the work. The owner is responsible and therefore accountable for getting the work done on time. In our particular program, the Program Office had to clarify and reinforce this distinction carefully as part of the enrollment stage.

### Inspection-Support

The project manager trusts in the commitments made and continually inspects the project to ensure delivery on schedule. This inspection strictly takes the form of supportive feedback, thereby encouraging people to disclose risks before they become problems. Whenever the projected results are at risk of falling short of the commitment, the project manager declares a project "cusp."

The term "cusp" is adapted here from Gleick to describe the potential turning points, or critical events, in a project.[5] (Other terms in conventional parlance include "gotchas," setbacks, crises, turning points, project breakdowns, and "calls to action." The managers used these terms during the program. For our purposes, we adopt the term cusp as an emotionally neutral term. It is important that at any point in the project the term used be one that gives an opening for the possibility of making a difference and for moving the project forward.) At the point of a cusp, everyone is ready to embrace change because it furthers the overall program objectives.

The management team collaborated to take advantage of cusps to propel project momentum toward the established goal. Examples of cusps in the Alpha AXP program are presented throughout this paper to demonstrate their integral value in the application of the Enrollment Management Model and the role they played in the creation of the model itself.

### Acknowledgment-Learning

At each step of the project, the Program Office acknowledges progress both personally and publicly. For each event, the management team repeatedly asks what was learned and how managers and the team can do even better next time. Teams are frequently coached to improve their methods for better results.

### Using the Model

In principle, the Program Office used the Enrollment Management Model in each component project. Of course in practice, not all groups used this methodology. Early in the program, only a few groups signed up. As the Alpha AXP Program Office began organizing the overall program, we started formalizing the methodology. As noted above, we learned extensively as events progressed. We found few useful manuals applicable to running such a large program effectively. Instead, the Program Office developed many of the tools "on the job," learning as the project unfolded.[6] This paper exaggerates a pure model rather than presenting its incremental development. To balance the picture, we show some of the pitfalls and side paths.

Most project managers followed the Enrollment Management Model either by instinct (experience) or by example. In several instances, they formally reached outside for training in running projects of this complexity. Depending on the size of the project or sub-project, managers used the model with varying degrees of rigor. For example, the larger projects and the program overall used formal inspection meetings and reviews. Subordinate projects were free to use formal or informal inspection processes. The program team inspected each group's inspection processes to ensure that there would not be any unfortunate management surprises.

### Using Project Cusps

As described earlier, cusps are critical project events, or crises. Conventional project management concentrates on rigorous planning to avoid such crises. The Program Office took the opposite approach: We strove to understand the critical events and milestones and used these cusps to increase project momentum, as Figure 2 illustrates. As the project approached each cusp, the Program

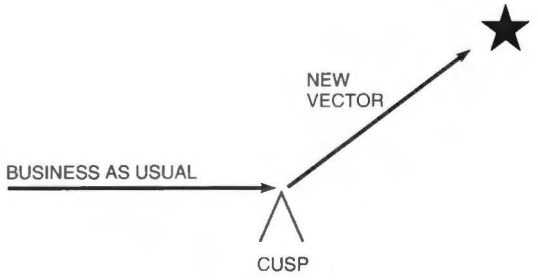



*Figure 2 Cusps as a Way to Change Directions*

Office dealt with the event promptly to ensure that the project continued to move toward the overarching goal. In other words, the managers did not develop a plan just to follow the plan. Instead, they developed a plan to understand the overall project flow and used the milestones and other events as opportunities to adjust the project velocity to keep moving toward the goal.[7] In many cases, we generated a cusp to propel the necessary change (for example, by creating a schedule crisis). In other cases, we took advantage of a cusp to make a necessary change.

As the management team became comfortable with using project cusps constructively, the Program Office actively solicited more of them. These increased the velocity and resulting momentum of the program, thereby achieving a "slingshot" effect. The Program Office used each cusp to acknowledge progress. As the team acknowledged more and more progress, the program's momentum moved from very low to break-even, and finally into high gear.

## *Vision–Enrollment Stage: Magnitude of the Program's Vision*

The vision for a program or project becomes the ultimate goal or deliverable. Thus, the Alpha AXP Program Manager's first task was to establish a vision shared by all groups that would contribute to the program. This vision had to be large enough to encompass all the work.

### *Alpha AXP Systems as Fifth-generation Computing*

The Alpha AXP family is at the confluence of five major trends in computing.

1. Nineteen ninety-two is the first year in which it is feasible to achieve 64-bit computing on a single microprocessor.
2. Nineteen ninety-two is the first year in which microprocessors have achieved over 100 MIPS (million instructions per second) of computing.
3. It is now cost-effective to place more than 4 gigabytes of main memory on a system; hence 32-bit addressing is insufficient.
4. Networking technology now allows the construction of networks with over 100-megabit throughput.
5. Cost-effective storage systems now exceed the many-gigabyte range and are approaching terabytes.

These computing systems will include large amounts of parallelism as compared with classical designs. The levels of performance and connectivity finally allow computing to realize greater human productivity: *mobile, highly interactive computing that supports group work with algorithms that intelligently analyze, simulate, and synthesize in support of a wide variety of human endeavors.* The application of this technology qualifies as the fifth generation of computing.[8,9]

The program vision for Alpha AXP systems, as shown in Figure 3, is to be the first family of systems to implement the technology and applications for the fifth generation of computing. This family is fully compatible across all members now and will be into future generations, ensuring that application binary programs will run unchanged. With no compromise to future performance, the initial family members also maintain a high degree of compatibility with current systems to allow easy migration for customers as they begin to require this technology. Delivering a family of high-quality systems in a timely fashion reestablishes Digital's reputation for technology and systems leadership.

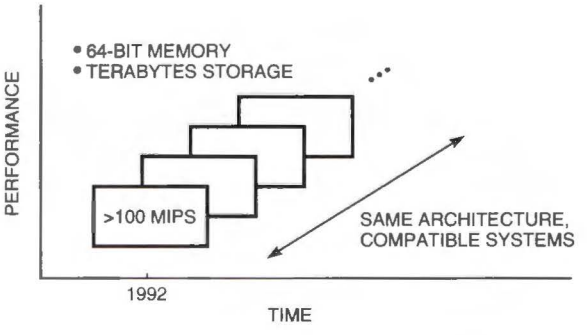



*Figure 3 Alpha AXP Vision*

### Getting Started

The Alpha AXP program grew out of research on computing, specifically the technology and benefits of different RISC (reduced instruction set computing) architectures, and from advanced development in compiler designs, VLSI (very large-scale integration) design, and semiconductor fabrication. In 1988, Digital's Executive Committee challenged Engineering to develop a system that would meet the customers' needs for competitive performance in all of Digital's computing environments. Engineering formed a cross-disciplinary task force that developed the requisite systems architecture and designs and produced the above vision and hence the Alpha AXP program. Digital's Executive Committee approved the Alpha AXP program in October 1989.[10]

### First Cusp: Executive Challenge to Accelerate Schedule

By the end of 1989, Digital had completed the advanced developments and signed off on the architecture and primary design documents. In a major review during January 1990, upper management challenged the program to improve the schedules to capture the market window for this new technology. The project managers understood the rationale for this demand but could see no way to meet the aggressive schedule. The result was a loss of rapport between management and the technical staff, with comments such as "Don't talk to me about crazy schedules" and "This is just going to be a lot of hard work."

The Program Office viewed this cusp as an opportunity rather than the crisis that it appeared to be. The office enrolled key project managers in the overall vision, i.e., in the business value of a timely execution. For some projects, it was sufficient to focus on the classic "time-to-market." However, for many, the ship date proved an insufficient motivator. Therefore the Program Office framed the vision differently, as follows. A program becomes profitable when it reaches break-even (i.e., cumulative revenues meet and then exceed cumulative expenses).

The time taken to achieve this point is known as the "time-to-profit."[11] The Program Office estimated that the program's schedule would affect Digital's revenue at the rate of $1 million per hour. That is, for each hour that the project could improve (lower) the time-to-profit, Digital would achieve an additional $1 million of revenue. The Program Office pointed out to the project managers that this revenue could translate to approximately $0.01 on the stock price for each hour of schedule improvement. With this concrete business metric in mind, the key project managers were willing to consider new ways to tackle the program's challenge.

Once the Alpha AXP program was approved, the Program Office began holding Alpha AXP quarterly review meetings. At these forums, groups reported plans and progress to a wide, cross-disciplinary audience. Initially, the audience was composed of engineering, manufacturing, and service groups. As the program gained momentum, other disciplines such as marketing and sales joined and began to report on their own progress. These forums helped generate belief and solidify enrollment. They also helped the Program Office identify problem areas before they became crises.

### First Cusp Result

We established a program-wide understanding of the importance of volume deliveries in 1992.

## Commitment–Delegation Stage: Delegating and Eliciting Commitment

With the key project managers sharing a common vision, the next step was to establish a work plan and to ensure that each group committed to deliver on its parts.

It had been 15 years since Digital attempted to change simultaneously its architecture, hardware, operating systems, compilers, and other layered products. Since the introduction of the VAX systems in the fall of 1977, each component had progressed relatively independent of the development schedules of the others. Fewer than half a dozen project team members had participated in the VAX design. For most participants, the system had always been in existence, and hence the developer of each subsystem could invoke and depend on the existence of all the other subsystems.

The need for the simultaneous development of multiple hardware and software systems complicated the coordination task. The Program Office addressed this complex coordination in two dimensions: technical and project management. In the technical dimension, the office formed a team of technical leaders from the core engineering groups, known as the EJST, shown in Figure 4. (EJST is an acronym for EVAX Joint Systems Team. EVAX was an early name for the Alpha AXP program. An earlier forum, the EVAX Technical Team, merged into the

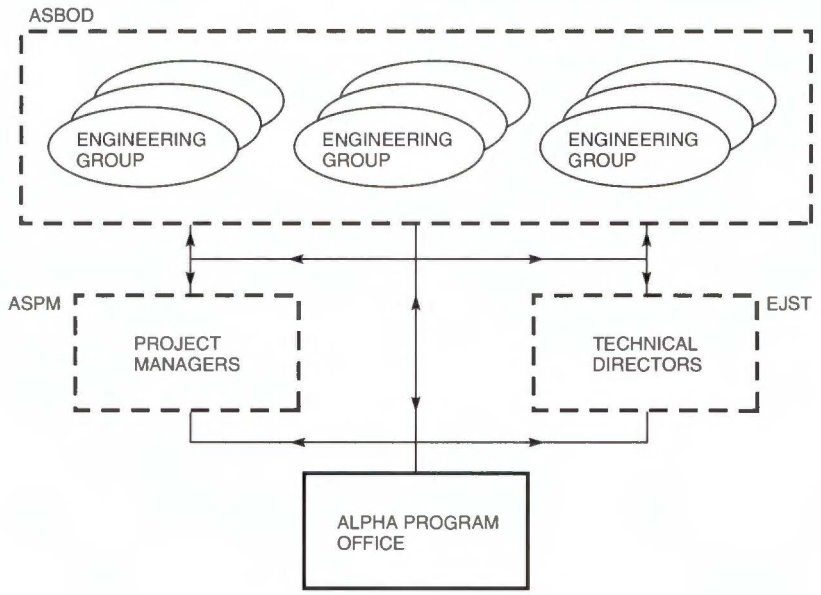



*Figure 4 The Alpha AXP Virtual Organization*

EJST process over time.) This group met weekly to set directions for important cross-group technical design and strategy issues. Since the group's charter was to resolve problems and ensure that solutions "stuck," the EJST became a group to which others brought technical problems for resolution.

In the project management dimension, the program manager formed a team of project managers. Members of this team were empowered by their contributing engineering development groups to make commitments and to be accountable for deliverables. This team was known as the ASPM (Alpha AXP System Project Managers). Given the magnitude of the overall task and the complexity of fully understanding the interdependencies, the project managers tended to view the project as impossibly complex. At the program level, the challenge then became to establish the Alpha AXP master plan. A master plan, however, evolved much more slowly than expected.

### *Second Cusp: Cannot Choose the Order of the Work*

Management's inability to provide an overall plan induced a crisis of disbelief. The project managers threatened to revolt (or move to other projects). The technical leaders were generating ever-larger design documents. The engineering development group managers declared that the Program Office had a crisis on its hands: We had to establish a program-wide work plan that showed the order in which each sub-project must deliver its contribution.

How does one coordinate without a plan? The Alpha AXP program manager kept asking the individual groups for their plans. What do you depend on? How long will it take? What resources do you need? What are your milestones or metrics of progress? The repeated answer was "I don't know because I don't know what everyone else is doing and when they will be done with their piece." At this time, we had already established the cross-functional ASPM team of experienced project managers representing most of the development groups. This team was unable to develop the component plans because they lacked a master plan.

### *Choosing the Strategy*

The engineering development group managers met in a full-day meeting to establish the overall parameters of the Alpha AXP program's plan. First, they established the business goals and examined the various technical constraints. The group tested the inclusion of each component with the question "Is this component critical to the overall business success of the Alpha AXP program?" This process established solid reasons for the contents of the master plan and kept the responsibility for the inclusion or exclusion of a component with the responsible development group. The

group then determined the organizational implications of such a work plan. Members of the group balanced the dimensions of business, technology, and organization to establish the priorities and work order. We institutionalized this group into the Alpha AXP System Board of Directors (ASBOD).

### *Representing the Plan*

With the major program priorities and constraints established, the Alpha AXP program manager then set off to establish the master plan. For all groups to see their contributions, he held the master plan to a single page. He established the content during an intense period in which he asked contributors to describe their assumptions and tasks and to show where on the overall plan their pieces would fall. The single-page format forced the management team to keep the plan to a high-level view and allowed contributors to see their pieces without adding the complexity of their own group's details. Further, in review meetings it was easy for everyone in the room to view the same picture so that the results could be seen, debated, and agreed upon.

Once the management team had outlined the plan, it was recommended by the project managers (ASPM) and approved by the engineering development group managers (ASBOD). Thus team members knew their goals would not change without clearly stated reasons. Further, others could start building their plans based on a consistent set of assumptions. The resulting single page also became a reference, which we called the "straw horse," to establish and reinforce constancy of purpose. Figure 5 is an example of the Straw Horse Plan. (We later upgraded the name to be the "tin horse" to connote the increasing degree of solidity of the underlying plans and commitments.)

### *Second Cusp Result*

We agreed on the overall single-page plan upon which teams could build their own plans.

### *Enrollment and Delegation: Value of Each Contribution*

With the master plan outlined (the straw horse reviewed and approved), the next step was to obtain the commitment of each contributing group. To address continuing skepticism about the necessity of each component and its schedule, the program manager walked each group through the overall program and the economic value of its urgency. The group was then asked to contribute to the overall system's value. A key prerequisite to this conversation was to establish a full-time project manager for each component group, who became the coordination point and who was held accountable for each deliverable.

### *Decide What to Do before How to Do It*

The Program Office found that each group went through a disbelief process similar to the one seen earlier for the program. The program manager urged each group to first focus on the "what" of their deliverable, before trying to decide the "how." The program manager ensured that the group grounded its overall estimates in reality. For example, a software group might count the number of modules to port and estimate the person-days per module. This kind of high-level, quantifiable estimate allowed the project manager to make an overall estimate without needing to understand the order of the specific tasks.

### *Third Cusp: Need for Project Management Expertise*

Members of several of the larger projects determined that they did not have sufficient project management experience. Previously, this realization would have resulted in replanning to move out the target schedule, perhaps repeatedly. Instead, given the group's commitment to the larger result, we found a much more aggressive behavior. For example, the OpenVMS AXP group publicly committed to their target schedule and stated, "We don't know how to achieve this, but we commit to finding a way." The next day they went to a project management consultant for training on how to build an aggressive, attainable schedule. This consultant conducted the seminar many times throughout the project for various groups.[12]

### *Third Cusp Result*

Groups introduced education and rigor into project management.

## *Inspection–Support Stage: Inspection with Supportive Feedback*

One of our vice presidents in the early 1980s had an aphorism: You get what you inspect, not what you expect. In other words, a common failing is that managers obtain someone's promise and expect that the results will be what they expected. Unfortunately, despite everyone's best intentions, circumstances and unexpected requests can easily

```
                        Straw Horse Plan
                                                        Aug 1990
Stage 1:  Technical Development System
          Fortran & C performance system; Single hardware platform
          English only

          Fortran, C, Bliss, Assembler, Debug, License Mgmt Facility,
          CASE tools (TPU, Code Mgmt System, Module Mgmt System,
          Performance Code Analyzer, Language Sensitive Editor,
          Digital Test Mgr), Compound Document Architecture,
          DECnet Phase IV task-to-task, DECwindows client via LAT
          SortMerge

Stage 2:  Commercial Development System
          Second hardware platform;
          International versions follow 3 months later

          COBOL, PASCAL, C++, ADA, CDD/Repository, Rdb,
          threads-rtl, RPC, GKS, PHIGS, Forms Mgmt System, DECforms,
          File Cache, VAXset, Distr Servers (name, time, file, queuing),
          Remote System Manager, ALL-IN-1 base, CDA suite,
          DECnet IV end node CTERM and DECwindows,
          TCP/IP, PATHWORKS, LATmaster, ABSS extensions

Stage 3:  Technical Robust System
          Open System; Symmetric Multi-Processing

          LISP, PL/I, user-written drivers, POSIX, disk shadowing,
          DBMS, ACMS, DAS or equivalent, full NAS, DECnet V end node,
          X.25 access, ALL-IN-1 fully supported

Stage 4:  Commercial Robust System
          Alpha Clusters; International versions released
          simultaneously

          New Batch/Print, all System Integrated Products,
          DECnet V routing node, SNA access

Stage 5:  Transaction System

          Transaction Monitor, Exec threads
```

*Figure 5 The Single-page Plan: An Extract from the Straw Horse Plan*

divert the promiser away from fulfilling the promise. Thus, managers learn to inspect regularly the progress of groups on whose commitments they depend.

The model, therefore, incorporates this traditional, essential project management practice. Its inclusion was prompted by another project crisis, described below.

## *Fourth Cusp: Project Slips Motivate Formal Operational Inspection*

The Program Office knew that it was working with highly motivated teams. On the basis of the earlier planning work, we assumed that they were all tightly focused on the objectives of the Alpha AXP program and shared our sense of schedule urgency. Suddenly, we were shocked by a memo stating that a critical project's schedule had slipped several months. Since virtually every other project depended upon it, this schedule slip could easily have led to a program disaster. Instead, we used the event to institute a regular operational inspection. Often, instituting such regular reviews is difficult and generally resisted by the reviewees. In this case, every group could see the danger of continuing without regular inspections and readily agreed to this new process.

The Program Office adopted the term "inspection," rather than "review," because we have found this term to be neutral or positive. In the past,

reviews had been imposed by line management and tended to encourage the reviewees to cover up issues until it was too late to recover. In contrast, the program manager, operating under the Program Office model, had no line authority and set up the monthly operational inspections in an open and supportive environment. The presenters were the designated project managers from each development group. The Program Office encouraged all presenters to bring in their risks and problems before it was too late to address them effectively. We used the single-page format again, as shown in Figure 6. Note that the simple, visual history of all milestones is at the top, so one can readily see any repetitive slips. The emphasis is on critical path events completed last month and those coming up next month. At the bottom are listed those issues that have been resolved and issues being opened, with clearly indicated ownership and due dates.

### Operational Excellence

To ensure that every project implemented the strategies, the Program Office established the principle of operational excellence across the Alpha AXP program. The office consistently recognized teams that accomplished their results on time and

```
PROJECT:    ALPHA/VMS
DATE:       April 8, 1992

SCHEDULE:
|  Q4 1991  |  Q1 1992  |  Q2 1992  |  Q3 1992  |  Q4 1992
|Oct Nov Dec|Jan Feb Mar|Apr May Jun|Jul Aug Sep|Oct Nov Dec
[---[---[---[---[---[---[---[---[---[---[---[---[---[---[---
B     4    5           I       E                              |Sep 91
 B     4   5      6     I           E                         |Nov 91
 B     4    5     6     I           E           S             |Jan 92
 B     4    5        6   I    U     E           S             |Mar 92
 B     4    5        6   I    U     E           S             |Apr 92

Milestones
B -- Base Level 3B (Editor, debugger, TIE, base DECnet)       -- DONE
4 -- Base Level 4 (More DECnet, utilities, and DW clients)    -- DONE
5 -- Base Level 5 (EV4 support, TFF, performance)             -- DONE
6 -- Base Level 6 (Performance & Tapes)                       -- DONE
I -- Internal field test & Pilot Porting Activity - FT1
U -- Internal field test update - FT2
E -- External field test & Early Support Program - FT3
S -- V1.0 submit to SSB

CRITICAL PATH EVENTS PAST MONTH:
      Shipped BL6 on March 12 - stable on ADU, Ruby, Cobra, Flamingo
      Shipped BL6 AMC porting toolkit
      Achieved FT1 (PPA) code freeze
      Received 2 Flamingo systems in Varese, Italy, for POSIX development
      With SPE (CSSE), delivered worldwide field test support training
      FT1 stabilization continuing

ACTIVITIES ALONG THE CRITICAL PATH (NEXT MONTH):
      Ship FT1; revised target is Apr 10
      Ship FT1 AMC porting toolkit
      Complete PPA Readiness Review
      Begin FT2 stabilization

ISSUES / DEPENDENCIES RESOLVED:
      Flamingo SFB graphics support formally accelerated into V1.0

ISSUES / DEPENDENCIES NOT RESOLVED:
      GEM BL24 compilers needed for ESP integration: D.L., May 15
      Rollout support staffing is not in anyone's plan: J.S., May 29
```

*Figure 6 The Single-Page Review*

predictably. We also used the monthly program-wide inspections to maintain a published record of progress. Thus, each project was encouraged to excel operationally and to learn from the experiences and presentations of the others.

### *Fourth Cusp Result*

The Program Office established monthly inspections using a consistent single-page document to record pertinent information.

## *Acknowledgment–Learning Stage: Building Momentum*

Developing the vision and plan resulted in a general sense of euphoria. Shortly afterwards, the reality of the work ahead descended like a cloud of despair. At this point, the primary challenge was to start building momentum in the program. In the Enrollment Management Model, building momentum—the acknowledgment-learning stage—is tightly intertwined with the inspection stage; that is, events reported during inspections were used to build momentum. The Program Office reinforced the vision and used momentum building to minimize the time period during which the team felt despair about the work ahead.

### *Fifth Cusp: Despair*

Since the overall program had such a formidable goal, many of the contributing teams became stalled with the magnitude of the task ahead of them. This manifested itself in the form of comments about the large amount of work, the resulting potential for schedule delays, and a fear of overtime demands. This syndrome is common in any large project, especially when commitments are made that involve taking large risks. The approach the program team took was to start recognizing each element of progress. As we distributed announcements of progress widely (using Digital's worldwide electronic mail network), we began to build momentum around the Alpha AXP program. Other groups picked up on this momentum and contributed to it themselves. This effect cascaded throughout the entire program—more groups perceived their tasks ahead as achievable; rapidly each group wanted its own progress acknowledged; and momentum increased.

The Program Office found that the members of a project appreciated and were motivated by the simple "thank you" represented by the public acknowledgment of their work. This contrasts with the conventional management wisdom that it is necessary to give frequent monetary rewards to motivate people. Although everyone appreciates the financial rewards, the biggest motivator is the professional recognition that the contributor did a good and necessary job!

The second benefit of the acknowledgment was its effect in creating a sense of momentum throughout all the project teams. Repeatedly, peer managers would comment that the Alpha AXP team was making significant progress. This in turn gave us a sense of accomplishment as well. So the program realized a double benefit from the original acknowledgment and a further slingshot effect with recognition coming back to the Program Office.

### *Fifth Cusp Result*

Program-wide, managers established the norm of frequent acknowledgment of progress.

As the Alpha AXP program made further progress, the Program Office actively solicited third-party and customer involvement. This involvement provided good feedback on progress and had the effect of reinforcing the fact that the program was on track to meet customer needs. In some cases, the project teams altered the Alpha AXP plans to better help our customers address their business needs. This further contributed to the credibility and momentum of the program as well as the sense of accomplishment.

### *Sixth Cusp: Broken Dependencies and Replanning*

Like any project, not every assumption and dependency proves to be correct or totally accurate. At one point, one of the major Alpha AXP hardware systems slipped its schedule for delivery of prototypes to software. After considering a number of alternatives, the operating system group proposed an alternate plan using a different hardware system and a changed order of events. They said in their management presentation at the time, "The question is not one of blame. Instead our goal is to preserve the ultimate schedule goal of the program, specifically its volume availability date."

### *Sixth Cusp Result*

Program-wide, team members established the principle of focusing on the desired state of time-to-profit rather than on blaming others for failures.

At another point, one group was at risk because it needed a critical skill for a week. A (historically) competing hardware group responded by asking what sort of resource, and then freely supplied the resource despite its own very tight schedule. In the past, these groups would compete for the same resource without collaborating for the common good.

### *Seventh Cusp: Incomplete Assumptions and the Need for the Performance Team*

Less than half way through the Alpha AXP program, the program team realized that some projects' assumptions were incomplete. RISC systems are notorious for requiring careful design and tuning to meet aggressive performance goals. Evidence from a related program at Digital suggested that some of our system performance homework was weak. The Program Office quietly asked the appropriate teams to assign some resources to measure key components and subsystems of the design. This confirmed the program team's concerns that the current plans were incomplete. Quickly, we pulled together a cross-disciplinary task force to analyze the information rigorously and to make recommendations. These analyses resulted in changes in the architecture, the chip design, the systems designs, and the software. The changes have proved to increase performance substantially.

### *Seventh Cusp Result*

The program established a performance team to change the design and plans as needed.

### *Eighth Cusp: Prototype Allocation Process*

As manufacturing started to deliver prototypes, the Program Office found that the early manufacturing build rate was lower than planned. This was the result of normal start-up problems. At the same time, initial demand had increased substantially. Nevertheless, the project administrators continued to ship the systems to engineering and applications groups in the original order. If this had continued, dependent software would have been delivered progressively later because of inadequate testing cycles. Our impact analysis indicated that the Alpha AXP volume availability would slip by three months.

The review team highlighted this problem in an early program readiness review. Traditionally, Digital uses readiness reviews to establish manufacturing's readiness to build systems. The Alpha AXP Program Office broadened this process and asked for a program-wide readiness review to identify the "showstopper" risks. As a result, the Program Office centralized the allocation process so that we could maintain the prototype allocations in real time. The result was to reestablish sufficient software test time and maintain momentum with minimal program impact.

### *Eighth Cusp Result*

The program teams decided that prototypes would be delivered based on program priorities, not solely on existing plans.

### *Ninth Cusp: Need for Quality Metrics*

Each group in the Alpha AXP program adopted very high standards for the quality of its work. The management team repeatedly found reinforcement of Phil Crosby's dictum: "Quality is free."[13] Results in group after group showed that early and continuous attention to quality resulted in held or improved schedules.

However, the program team noticed that we were not inspecting and measuring progress in quality at the total systems level; customers care about only the quality of the total result. As the projects started integrating into a total system, the Program Office established an independent group to measure overall quality levels. The classic reaction to such independently derived quality metrics is that they are meaningless. Instead, since the program established the metrics at the moment when everyone saw the need, the reaction has been to focus on the total system's quality without dropping attention on the individual component metrics.

### *Ninth Cusp Result*

The program formalized system-wide quality metrics.

## *Results and Lessons Learned*

Digital met exactly the program's overall schedule to the month (i.e., date for high-volume shipments), despite numerous setbacks along the way. The Alpha AXP system is meeting the original performance goals, and quality is excellent. Digital's Board of Directors has approved the full Alpha AXP program business plan and the investments necessary to capitalize on the Alpha AXP family's early

successes. Initial reactions from customers have been favorable. Third parties have committed Alpha AXP support for their products in record numbers.

### *What Worked Well*

The Program Office in conjunction with the Enrollment Management Model has worked well. If the management team had followed traditional approaches, we would still be getting organized. Using the model, each group has been able to bring its full capabilities to bear as problems have arisen. The project teams have accepted the introduction of multiple levels of inspection, and other programs within Digital are copying this aspect of the model. Further, the notion of using project cusps creatively has been an effective tool to build momentum. Finally, a common schedule and inspection discipline allowed the schedule to become an opportunity to reinforce a shared vision. This positive view contrasts with the conventional interpretation of a schedule as a burden.

As a result, most groups met very aggressive goals on schedule. Several groups accelerated their deliverables despite having the most complex tasks. For example, the OpenVMS AXP system group not only met its original schedule but also accelerated numerous deliverables into earlier base levels or releases. Figure 7 shows the OpenVMS schedule and actual dates of availability; footnotes indicate functional accelerations. The networks group delivered DECnet on the AXP system an entire base level early. The database systems group accelerated its schedule by several months and demonstrated products four months early at Digital's DECWORLD '92 trade show.

Clearly one of the major lessons was to establish a constancy of purpose and hold to it while continually learning and updating the detailed plans. The single-page representation of the goals and master plan is a key element in maintaining this constancy.

### *What We Would Do Differently*

Looking back, we would have approached the program differently in two areas. First, project teams would have benefited from broader early education on project methodology. Several projects had significant slips, causing undue hardship on other groups. The Program Office should have

ALPHA/VMS SCHEDULE RESULTS

| MILESTONE | ORIGINAL | ACTUAL |
|---|---|---|
| Phase 0 closure | Aug 30, 1990 | Aug 30 |
| Alpha VMS minimal login | Jun 17, 1991 | Mar 20 |
| BL1 ship - minimal login | Jul 15, 1991 | May 31 |
| BL2 ship - RTLs, DW(1) & LAT | Aug 26, 1991 | Jul 12 |
| BL3A ship - ISAM, linker | n/a | Aug 23 |
| BL3B ship - prog devel & TIE(2), DECnet(3) | Oct 7, 1991 | Oct 10 |
| BL4 ship | Nov 18, 1991 | Nov 15 |
| BL5 ship - functionally complete(4) | Dec 30, 1991 | Jan 10 |
| BL6 ship - Ruby complete(5) | Feb 21, 1992 | Mar 6 |
| FT1/PPA | Apr 3, 1992 | Apr 10 |
| Phase 1 | May 1992 | May 20 |
| FT2/PPA | n/a 1992 | May 22 |
| FT3/ESP(6) | Jul 2, 1992 | Jul 8 |
| FT4/ESP | n/a 1992 | Aug 14 |
| V1.0 SSB submission (LRS) | Oct 2, 1992 | Oct 26 |

Notes:

(1) DECwindows
(2) Translated Image Environment (RTL for translated images)
(3) DECnet accelerated from BL4 to BL3B
(4) Full graphics support accelerated from next version to V1.0
(5) Support for multiple hardware platforms accelerated from next version to V1.0
(6) FDDI support accelerated from next version to V1.0

*Figure 7 Original OpenVMS Milestone and Delivery Dates*

introduced Ron LaFleur's project methodology sooner and pervasively. Instead, we waited until each group saw the need and then tried to introduce it. For groups such as the OpenVMS AXP system group, the methodology was introduced early. However, other groups needed (and still need) this discipline.

Second, the office would have conducted more pervasive project inspections. Several groups were very late in producing schedules and plans that the Program Office could understand. The office was unable to obtain their cooperation to hold detailed and frequent inspections. Eventually, the Program Office was invited to set up and participate in appropriate inspections of schedule, process, etc. However, we should have insisted on these much sooner.

## Summary

The Alpha AXP program is the most complex program in Digital's history and has been delivered on schedule with high quality. The Alpha AXP Program Office used a rigorous management methodology to build the program-level teamwork necessary to accomplish this breakthrough. The office proved the effectiveness of the Enrollment Management Model: vision–enrollment, commitment–delegation, inspection–support, and acknowledgment–learning. Integral to this model and empowering to the team is to take each cusp head-on and to use them to increase momentum. The management team has been learning as the program progressed and has identified areas needing strengthening for future programs.

## Acknowledgments

The author thanks the following senior managers for demonstrating the importance of good management: Gordon Bell for architecture and a clear strategy; Ken Olsen for demanding simple, single-page plans; Jeff Kalb for operational excellence; David Stone for the model of focusing on the desired state; Bob Supnik for the paradigm of the Program Office.

The author also thanks key members of the Alpha AXP Program Office for their contributions in managing the program and developing the Enrollment Management Methodology: Al Avery for systems integration and significant help preparing this paper; Scott Gordon for competitive benchmarking; Ellen Salisbury for planning; and Ken Schultz for operations and inspection.

## References and Note

1. R. Waterman, T. Peters, and J. Phillips, "Structure is Not Organization," *Business Horizons,* no. 80302 (June 1980).
2. C. Savage, *Fifth Generation Management* (Burlington, MA: Digital Press, 1990).
3. W. Oncken and D. Wass, "Management Time: Who's got the monkey," *Harvard Business Review,* vol. 18, no. 6 (November 1974): 75–79.
4. M. McMaster and J. Grinder, *PRECISION: A New Approach to Communication* (Bonny Doon, CA: Precision Models, 1980).
5. J. Gleick, *CHAOS: Making a New Science* (New York: Penguin Books, 1987).
6. P. Senge, *The Fifth Discipline: The Art and Practice of the Learning Organization* (New York: Doubleday, 1990).
7. A. Scherr, "Managing for Breakthroughs in Productivity," *Human Resource Management,* vol. 28, no. 3 (Fall 1989): 403–424.
8. L. Tesler, "Networked Computing in the 1990s," *Scientific American* (September 1991): 86–93.
9. The five generations of computing are as follows: 1950s, standalone; 1960s, batch; 1970s, timesharing; 1980s, personal; 1990s, mobile distributed.
10. R. Comerford, "How DEC Developed Alpha," *IEEE Spectrum* (July 1992): 26–31.
11. C. House and R. Price, "The Return Map: Tracking Product Teams," *Harvard Business Review,* vol. 69, no. 1 (January 1991): 92–100.
12. R. LaFleur, "A Seminar in Project Management" (Scituate, MA: Project Management Assistance Co., 1990).
13. P. Crosby, *Quality Is Free: The Art of Making Quality Certain* (New York: McGraw-Hill, 1979).

## General References

F. Brooks, *The Mythical Man-month: Essays on Software Engineering* (Reading, MA: Addison-Wesley, 1975).

R. Neustadt and E. May, *Thinking In Time: The uses of history for decision makers* (New York: The Free Press, 1986).

# 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. Digital engineers who would like to contribute a paper to the* Journal *should contact the editor at RDVAX::BLAKE.*

Topics covered in previous issues of the *Digital Technical Journal* are as follows:

**NVAX-microprocessor VAX Systems**
*Vol. 4, No. 3, Summer 1992,* EY-J884E-DP

**Semiconductor Technologies**
*Vol. 4, No. 2, Spring 1992,* EY-L521E-DP

**PATHWORKS: PC Integration Software**
*Vol. 4, No. 1, Winter 1992,* EY-J825E-DP

**Image Processing, Video Terminals, and Printer Technologies**
*Vol. 3, No. 4, Fall 1991,* EY-H889E-DP

**Availability in VAXcluster Systems/ Network Performance and Adapters**
*Vol. 3, No. 3, Summer 1991,* EY-H890E-DP

**Fiber Distributed Data Interface**
*Vol. 3, No. 2, Spring 1991,* EY-H876E-DP

**Transaction Processing, Databases, and Fault-tolerant Systems**
*Vol. 3, No. 1, Winter 1991,* EY-F588E-DP

**VAX 9000 Series**
*Vol. 2, No. 4, Fall 1990,* EY-E762E-DP

**DECwindows Program**
*Vol. 2, No. 3, Summer 1990,* EY-E756E-DP

**VAX 6000 Model 400 System**
*Vol. 2, No. 2, Spring 1990,* EY-C197E-DP

**Compound Document Architecture**
*Vol. 2, No. 1, Winter 1990,* EY-C196E-DP

**Distributed Systems**
*Vol. 1, No. 9, June 1989,* EY-C179E-DP

**Storage Technology**
*Vol. 1, No. 8, February 1989,* EY-C166E-DP

**CVAX-based Systems**
*Vol. 1, No. 7, August 1988,* EY-6742E-DP

**Software Productivity Tools**
*Vol. 1, No. 6, February 1988,* EY-8259E-DP

**VAXcluster Systems**
*Vol. 1, No. 5, September 1987,* EY-8258E-DP

**VAX 8800 Family**
*Vol. 1, No. 4, February 1987,* EY-6711E-DP

**Networking Products**
*Vol. 1, No. 3, September 1986,* EY-6715E-DP

**MicroVAX II System**
*Vol. 1, No. 2, March 1986,* EY-3474E-DP

**VAX 8600 Processor**
*Vol. 1, No. 1, August 1985,* EY-3435E-DP

Subscriptions to the *Digital Technical Journal* are available on a prepaid basis. The subscription rate is $40.00 for four issues and $75.00 for eight issues. Orders should be sent to Cathy Phillips, Digital Equipment Corporation, MLO1-3/B68, 146 Main Street, Maynard, MA 01754-2571, U.S.A., Telephone: (508) 493-2894, FAX: (508) 493-0637. Inquiries can be sent electronically to DTJ@CRL.DEC.COM. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

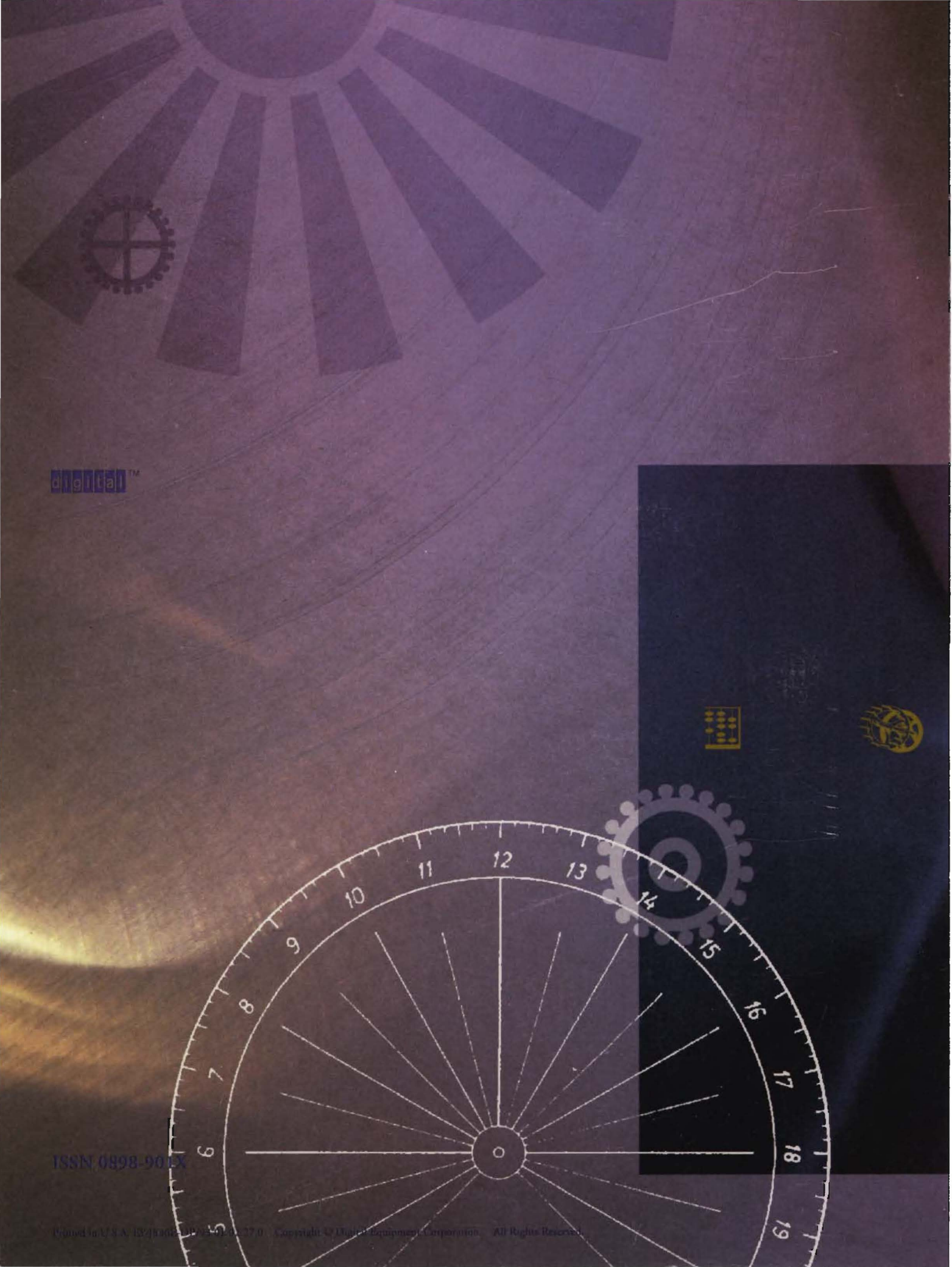
Single copies and past issues of the *Digital Technical Journal* are available for $16.00 each from Digital Press, Department EEB, 1 Burlington Woods Drive, Burlington, MA 01830-4597. Single issues can also be ordered by calling DECdirect at 1-800-DIGITAL (1-800-344-4825).

# Recent Digital U.S. Patents

*The following patents were recently issued to Digital Equipment Corporation. Titles and names supplied to us by the U.S. Patent and Trademark Office are reproduced exactly as they appear on the original published patent.*

| | | |
|---|---|---|
| D327,261 | K. L. Korellis and R. T. Faranda | Front Face Panel Portion for Enclosure Doors for a Computer |
| D327,477 | K. L. Korellis | Front Panel for an Integrated Storage Assembly for Computer Storage Units |
| 5,092,631 | M. G. M. Masnik and R. C. Martel | Safety Enclosure for Gas Line Fittings |
| 5,093,628 | I. T. Chan | Current-Pulse Integrating Circuit and Phase-Locked Loop |
| 5,093,775 | W. R. Grundmann, R. F. Boucher, and T. Fossum | Microcode Control System for Digital Data Processing System |
| 5,094,980 | A. Shepela | Method for Providing a Metal-Semiconductor Contact |
| 5,095,441 | D. F. Hopper, E. G. Fortmiller, S. Kundu, and D. F. Wall | Rule Inference and Localization during Synthesis of Logic Circuit Designs |
| 5,095,460 | T. L. Rodeheffer | Rotating Priority Encoder Operating by Selectively Masking Input Signals to a Fixed Priority Encoder |
| 5,095,471 | M. D. Sidman | Velocity Estimator in a Disk Drive Positioning System |
| 5,095,613 | K. R. Hussinger and M. L. Mallary | Thin Film Head Slider Fabrication Process |
| 5,097,370 | Y. Hsia | Subambient Pressure Air Bearing Slider for Disk Drive |
| 5,097,387 | J. L. Griffith | Circuit Chip Package Employing Low Melting Point Solder for Heat Transfer |
| 5,097,411 | P. L. Doyle, J. P. Ellenberger, E. O. Jones, D. C. Carver, S. D. Dipirro, B. J. Gerovac, W. P. Armstrong, E. S. Gibson, R. E. Shapiro, K. C. Rushforth, and W. C. Roach | Graphics Workstation for Creating Graphics Data Structure Which Are Stored Retrieved and Displayed by a Graphics Subsystem for Competing Programs |
| 5,097,436 | J. H. Zurawski | High Performance Adder Using Carry Prediction |
| 5,097,468 | E. Earlie | Testing Asynchronous Processes |
| 5,099,367 | M. D. Sidman | Method of Automatic Gain Control Basis Selection and Method of Half-Track Servoing |
| 5,099,484 | D. W. Smelser | Multiple Bit Error Detection and Correction System Employing a Modified Reed-Solomon Code Incorporating Address Parity and Catastrophic Failure Detection |
| 5,099,485 | W. F. Bruckert, T. D. Bissett, D. Mazur, J. Munzer, F. Bernaby, and J. H. Bhatia | Fault Tolerant Computer Systems with Fault Isolation and Repair |
| 5,099,517 | A. Gupta, W. R. Hawe, M. F. Kempf, and C. S. Lee | Frame Status Encoding for Communication Networks |
| 5,101,106 | E. E. Cox, Jr. and M. P. Rolla | Resonant Technique and Apparatus for Thermal Capacitor Screening |
| 5,101,362 | E. Simoudis | Modular Blackboard-Based Expert System |
| 5,101,402 | D. Chiu and R. Sudama | Apparatus and Method for Realtime Monitoring of Network Sessions in a Local Area Network |
| 5,101,485 | F. L. Perazzoli, Jr. | Virtual Memory Page Table Paging Apparatus and Method |
| 5,101,493 | R. L. Travis and W. R. Laurune | Digital Computer Using Data Structure Including External Reference Arrangement |
| 5,103,352 | W. Y. Moon and R. Y. Noguchi | Phased Series Tuned Equalizer |

| | | |
|---|---|---|
| 5,103,393 | J. P. Harris, D. Leibholz, and B. Miller | Method of Dynamically Allocating Processors in a Massively Parallel Processing System |
| 5,103,553 | M. Mallary | Method of Making a Magnetic Recording Head |
| 5,105,055 | W. C. Mooney, J. R. Santandreu, and K. Kshonze | Tunnelled Multiconductor System and Method |
| 5,105,183 | K. O. Beckman | System for Displaying Video from a Plurality of Sources on a Display |
| 5,105,322 | E. L. Steltzer | Transverse Positioner for Read/Write Head |
| 5,105,408 | N. K. S. Lee, J. W. Howard, P. K. Tan, and W. Hrytsay | Optical Head with Flying Lens |
| 5,107,398 | D. A. Bailey | Cooling System for Computers |
| 5,107,462 | W. R. Grundmann, V. R. Hay, L. O. Herman, and D. M. Litwinetz | Self Timed Register File Having Bit Storage Cells with Emitter-Coupled Output Selectors for Common Bits Sharing a Common Pull-Up Resistor and a Common Current Sink |
| 5,107,503 | C. M. Riggle, L. Weng, and P. N. Hui | High Bandwidth Reed-Solomon Encoding, Decoding and Error Correcting Circuit |
| 5,107,506 | L. J. Weng and B. A. Leshay | Error Trapping Decoding Method and Apparatus |
| 5,108,837 | M. L. Mallary | Laminated Poles for Recording Heads |
| 5,109,307 | M. Sidman | Continuous-Plus-Embedded Servo Data Position Control System for Magnetic Disk Device |
| 5,109,495 | D. B. Fite, T. Fossum, W. R. Grundmann, D. P. Manley, F. X. McKeen, J. E. Murray, R. M. Salett, E. Samberg, and D. P. Stirling | Method and Apparatus Using a Source Operand List and a Source Operand Pointer Queue between the Execution Unit and the Instruction Decoding and Operand Processing Units of a Pipelined Data Processor |
| 5,111,352 | S. C. Das and M. L. Mallary | Three-Pole Magnetic Head with Reduced Flux Leakage |
| 5,111,424 | D. D. Donaldson and R. B. Gillett, Jr. | Lookahead Bus Arbitration System with Override of Conditional Access Grants by Bus Cycle Extensions for Multicycle Data Transfer |
| 5,111,465 | B. C. Edem, R. P. Helliwell, J. T. Johnston, and R. F. Lary | Data Integrity Features for a Sort Accelerator |
| 5,112,142 | F. Titcomb and J. Cordova | Hydrodynamic Bearing |
| 5,112,662 | Q. Y. Ng | Method for Providing a Lubricant Coating on the Surface of a Magneto-Optical Disk and Resulting Optical Disk |
| 5,113,352 | J. L. Finnerty | Integrating the Logical and Physical Design of Electronically Linked Objects |
| 5,113,515 | D. B. Fite, R. C. Hetherington, M. M. McKeon, D. P. Manley, and J. E. Murray | Virtual Instruction Cache System Using Length Responsive Decoded Instruction Shifting and Merging with Prefetch Buffer Outputs to Fill Instruction Buffer |
| 5,113,521 | F. X. McKeen, T. Fossum, D. P. Bhandarkar, and C. A. Wiecek | Method and Apparatus for Handling Faults of Vector Instructions Causing Memory Management Exceptions |
| 5,115,359 | M. D. Sidman | Fault Tolerant Frame, Guardband and Index Detection Methods |
| 5,115,360 | M. D. Sidman | Embedded Burst Demodulation and Tracking Error Generation |
| 5,115,455 | W. A. Samaras, D. T. Vaughan, and A. D. Ingraham | Method and Apparatus for Stabilized Data Transmission |
| 5,115,858 | J. S. Fitch and W. R. Hamburgen | Micro-Channel Wafer Cooling Chuck |
| 5,117,351 | S. Miller | Object Identifier Generator for Distributed Computer System |




ISSN 0898-901X